Project 1 : Basic Sorting Algorithms and Hybrid Optimization

Course: CS20009.04 Data Structure
Name: Yijia Chen
Student Number: 24300240127
Date: September 26, 2025

Introduction

Sorting is widely used in many programs as an intermediate step. Therefore, it's necessary to analyze the efficiency of different kinds of sorting algorithms, and choose an appropriate type in application.

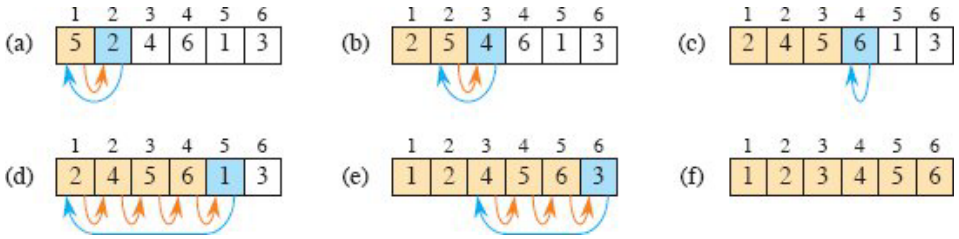
This project aims to:

- Implement insertion sort, mergesort, and quicksort in C++.
- Analyze and compare their efficiency, both in theory and by experiment.
- Combine the advantages of these sorting algorithms to optimize the performance.

Implementation of Sorting Algorithms

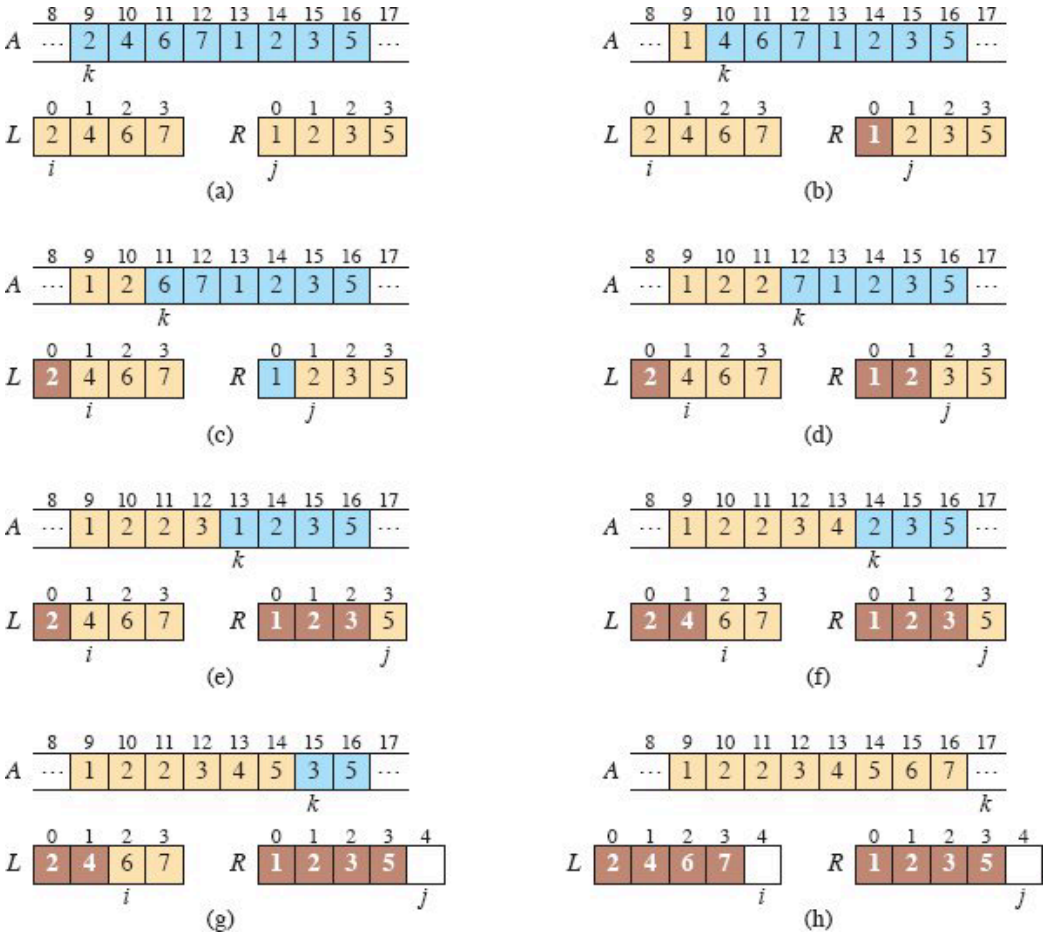
Insertion Sort

Insertion sort maintains a sorted prefix `a[1..j-1]`, and insert the current element `a[j]` into it by comparing and exchanging it with its previous element repeatedly until `a[j] > a[j-1]`.



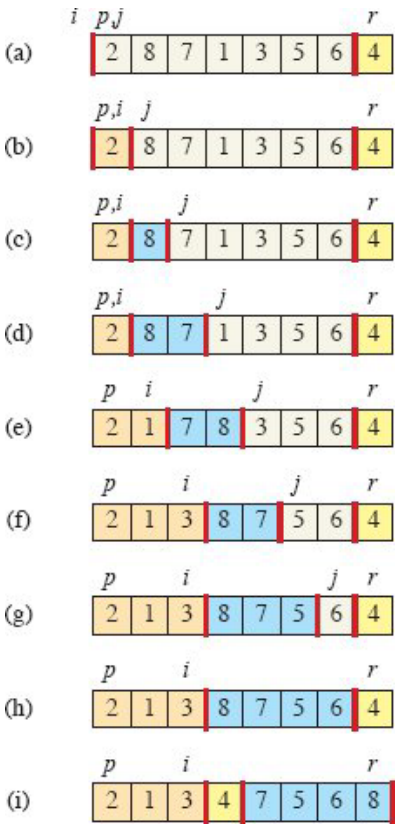
Mergesort

Mergesort follows the *divide-and-conquer* method: divide the array into two subarrays, conquer them recursively using `mergeSort`, and combine these two sorted subarrays using the auxiliary procedure `merge`.



Randomized Quicksort

`partition` divides the array into two parts: all elements in one part is less than `pivot` , and all elements in the other part is greater than `pivot` . Then it recursively calls `quickSort` on the two subarrays.



```cpp
#pragma once
#include <vector>
#include <algorithm>

using std::vector;

// insertion sort

template <typename T>
void insertionSort(vector<T>& a) {
    int n = a.size();
    for (int j = 1; j < n; ++j) {
        T key = a[j];
        // insert a[j] into the sorted sequence a[1..j-1]
        int i = j-1;
        while (i >= 0 && a[i] > key) {
            a[i+1] = a[i];
            --i;
        }
        a[i+1] = key;
    }
}

// merge sort

template <typename T>
void merge(vector<T>& a, int l, int m, int r) {
    // a[l..m] & a[m+1..r] are sorted
    int n1 = m-l+1, n2 = r-m; // lengths of 2 subarrays
    vector<T> L, R; // two sorted subarrays
    copy(a.begin() + l, a.begin() + m + 1, std::back_inserter(L));
    copy(a.begin() + m + 1, a.begin() + r + 1, std::back_inserter(R));

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) a[k++] = L[i++]; // copy the smaller one
        else a[k++] = R[j++];
    }
    // when one of L and R is empty, copy the remaining elements
    while (i < n1) a[k++] = L[i++];
    while (j < n2) a[k++] = R[j++];
}

template <typename T>
void mergeSort(vector<T>& a, int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(a, l, m);
        mergeSort(a, m+1, r);
        merge(a, l, m, r);
    }
}
```

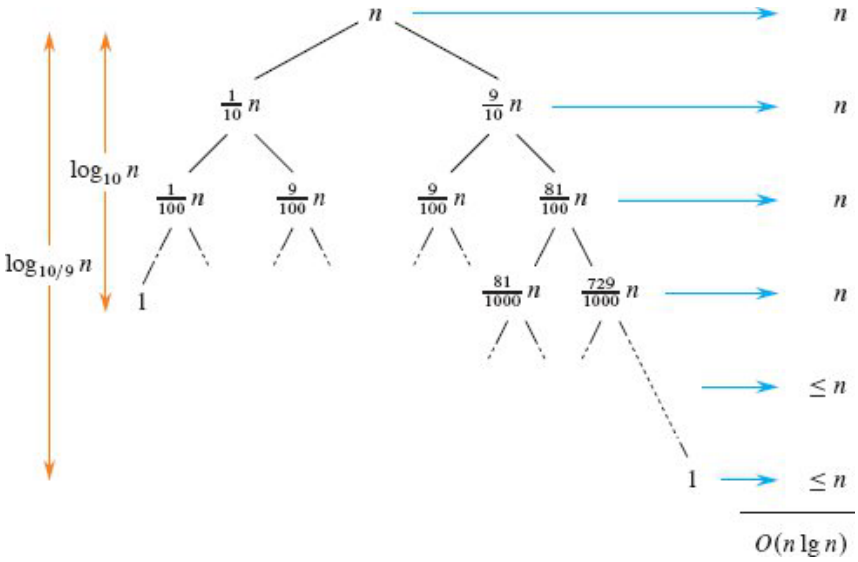
```
// randomized quick sort

template <typename T>
int partition(vector<T>& a, int l, int r) {
    int pivotIndex = rand() % (r - l + 1);
    std::swap(a[pivotIndex], a[r]);

    T pivot = a[r];
    int i = l - 1;
    for (int j = l; j < r; ++j) {
        if (a[j] <= pivot) {
        std::swap(a[++i], a[j]);
        }
    }
    std::swap(a[i+1], a[r]);
    return i+1;
}

template <typename T>
void quickSort(vector<T>& a, int l, int r) {
    if (l < r) {
        int m = partition(a, l, r);
        quickSort(a, l, m-1);
        quickSort(a, m+1, r);
    }
}
```

Theoretical Analysis

Running Time of the Hybrid Algorithm

Algorithm	Time $O(\cdot)$
`insertionSort`	n^2
`mergeSort`	$n \lg n$
`quickSort`	$n \lg n$

In the hybrid algorithm, `quickSort` is called recursively on subarrays with more than k elements. Subarrays with fewer than k elements, which are left unsorted by the recursive calls of `quickSort`, are subsequently sorted by a single call to `insertionSort` directly. This takes the advantage of speediness `insertionSort` when input scale is relatively small.



Consider the recursive tree of `quickSort` .

Suppose the `partition` procedure always splits input into subarrays of sizes in a fixed 9:1 ratio, as shown in the diagram. On every level, the sum of subarray sizes $\leq n$. In particular, if none of the subproblems reaches the base case and the recursive branch terminates, then the subarrays form a partition of the original array, and their sizes add up to exactly n, corresponding to the top $\log_{10} n$ levels in this example.

Therefore, recursive function calls on every level takes $O(n)$ time. In the hybrid algorithm, `quickSort` only applies to $\log_k n = \lg(n/k)$ levels from the top, whose expected running time is $O(n \lg(n/k))$.

When `quickSort` halts, the array is partitioned into approximately n/k unsorted subarrays, each sized $\leq k$. The operations of `insertionSort` on the whole array is equivalent to calling `insertionSort` on the n/k arrays separately, because comparison between any two elements in different subarrays is unnecessary after `partition` . Let l_i be the length of the i^{th} subarray, where $0 \leq l_i \leq k$ and $\sum_i l_i = n$. Using `insertionSort` to sort these subarrays takes

$$\sum_i O(l_i^2) \leq \sum_i O(k \cdot l_i) = O(\sum_i k \cdot l_i) = O(nk).$$

Combine the running time of `quickSort` and `insertionSort` above, Therefore, the hybrid sorting algorithm runs in

$$O(n \lg(n/k)) + O(nk) = O(nk + n \lg(n/k))$$

expected time.

Naive Deduction on k

Let $f(k) = nk + n \log_2(n/k)$, where n is treated as a constant, and k is a continuous positive variable.

$$f'(k) = n + n \cdot (-\frac{1}{k \ln 2}) = n \cdot (1 - \frac{1}{k \ln 2})$$

Set $f'(k) = 0$. The critical point $k^* = \frac{1}{\ln 2} \approx 1.4427$.

Since

$$f''(k) = \frac{1}{k^2 \ln 2} > 0, \text{ for } k > 0,$$

the critical point k^* is a strict local minimum.

Because the input size k of `insertionSort` must be a positive integer, this simplified model suggests $k = 1$ or $k = 2$ as candidates.

However, this result is quite misleading, mainly because the big-O notation $O(nk + n \lg(n/k))$ ignores constant factors and discards lower-order terms, which dominates when k is small. Recursive function calls in `quickSort` introduce overhead, making its constant coefficients much greater than the simple iterative structure of `insertionSort`. Thus, the asymptotic analysis does not make sense to determine the threshold for the hybrid algorithm.

Experimental Analysis

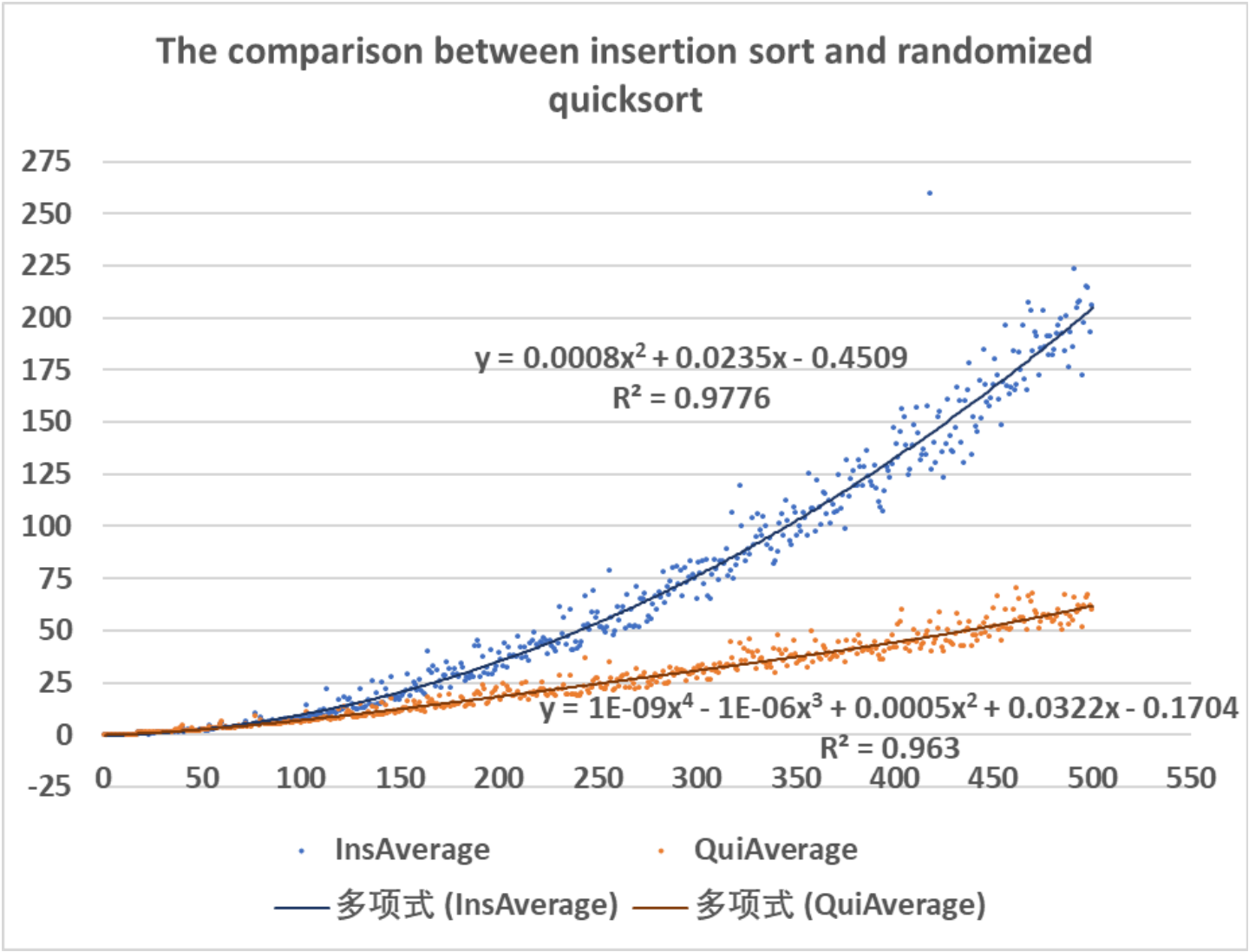
To choose the value of k, the objective is to identify the input size at which randomized quicksort begins to outperform insertion sort.

A benchmarking program is implemented to compare insertion sort and randomized quicksort over input sizes ranging from 1 to 500. The test is run 500 times, and the results are automatically saved in "results.csv".

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <chrono>
#include <algorithm>
#include <random>
#include "sort.h"

using std::vector;
using namespace std::chrono;

vector<int> generateRandomArray(int n) {
    static std::mt19937 gen(std::random_device{}()); // Mersenne Twister engine
    std::uniform_int_distribution<int> dist(0, 10000);

    vector<int> arr(n);
    for (auto &elem : arr) {
        elem = dist(gen);  // high-quality random integer
    }
    return arr;
}

int main() {
    srand(time(0));

    std::ofstream fout("results.csv");
    if (!fout.is_open()) {
        std::cerr << "Error: could not open results.csv\n";
        return 1;
    }

    int totalRuns = 500;

    // Write header row
    fout << "InputSize";
    for (int run = 1; run <= totalRuns; ++run) {
        fout << ",InsertionSort_Run" << run;
    }
    for (int run = 1; run <= totalRuns; ++run) {
        fout << ",QuickSort_Run" << run;
    }
    fout << "\n";

    vector<int> sizes(500);
    for (int i = 0; i < 500; ++i) sizes[i] = i + 1;

    // For each input size, run totalRuns times
    for (int n : sizes) {
        fout << n;

        // Collect insertion sort results first
        vector<long long> insertionTimes;
```

```
        insertionTimes.reserve(totalRuns);

        vector<long long> quickTimes;
        quickTimes.reserve(totalRuns);

        for (int run = 1; run <= totalRuns; ++run) {
            vector<int> arr1 = generateRandomArray(n);
            vector<int> arr2 = arr1;

            auto start = high_resolution_clock::now();
            insertionSort(arr1);
            auto end = high_resolution_clock::now();
            insertionTimes.push_back(duration_cast<microseconds>(end - start).count());

            start = high_resolution_clock::now();
            quickSort(arr2, 0, arr2.size() - 1);
            end = high_resolution_clock::now();
            quickTimes.push_back(duration_cast<microseconds>(end - start).count());
        }

        // Write insertion sort results
        for (auto t : insertionTimes) fout << "," << t;
        // Write quicksort results
        for (auto t : quickTimes) fout << "," << t;

        fout << "\n";
    }

    fout.close();
    std::cout << "Results saved to results.csv\n";
    return 0;
}
```

To visualize the results, I used Excel to compute the averages and generate a scattered plot to gain insight into the data.



From the diagram, we conclude that the running time of insertion sort curves into

$$y = 0.008x^2 + 0.0235x - 0.4509,$$

while randomized quicksort is

$$y = 1 \times 10^{-9}x^4 - 1 \times 10^{-6}x^3 + 0.0005x^2 + 0.0322x - 0.1704.$$

The formula of quicksort does not exactly fit into the pattern of $n \lg n$, primarily due to limitations of Excel's curve-fitting functions. Fortunately, $R^2 = 0.963$, which indicates that the curve is relatively accurate.

Solve the equation

$$0.008x^2 + 0.0235x - 0.4509 = 1 \times 10^{-9}x^4 - 1 \times 10^{-6}x^3 + 0.0005x^2 + 0.0322x - 0.1704,$$

and we obtain the intersection point

$$x = 44.0459.$$

Insertion sort is faster when $x < 44.0459$, and quicksort is faster when $x > 44.0459$. It can be inferred that a possible choice of k in the hybrid sorting algorithm is around 44.

Hybrid Quicksort with Insertion Sort

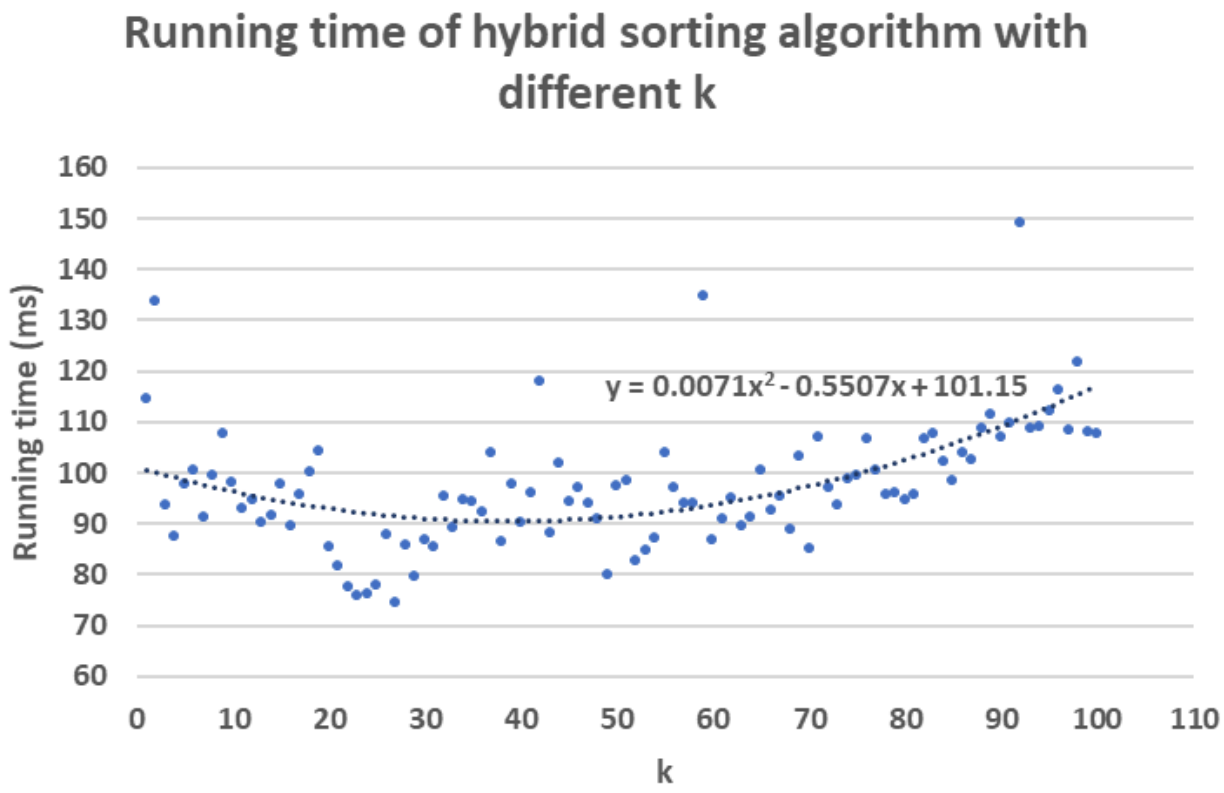
The Choice of k by experiment

To determine the value of k that yields the highest efficiency, the following experiment was conducted. It runs the test 1000 times with a fixed input size 1000 and k ranging from 1 to 100.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <chrono>
#include <algorithm>
#include <random>

using std::vector;
using namespace std::chrono;

template <typename T>
void insertionSort(vector<T>& a, int l, int r) {
    int n = r - l + 1;
    for (int j = l; j < r+1; ++j) {
        T key = a[j];
        // insert a[j] into the sorted sequence a[l..j-1]
        int i = j-1;
        while (i >= 0 && a[i] > key) {
            a[i+1] = a[i];
            --i;
        }
        a[i+1] = key;
    }
}

template <typename T>
int partition(vector<T>& a, int l, int r) {
    int pivotIndex = l + rand() % (r - l + 1);
    std::swap(a[pivotIndex], a[r]);

    T pivot = a[r];
    int i = l - 1;
    for (int j = l; j < r; ++j) {
        if (a[j] <= pivot) {
        std::swap(a[++i], a[j]);
        }
    }
    std::swap(a[i+1], a[r]);
    return i+1;
}

template<typename T>
void hybridQuickSortTest(vector<T>& a, int l, int r, int k) {
    if (l < r) {
        if (r - l + 1< k) return;
        else {
            int m = partition(a, l, r);
            hybridQuickSortTest(a, l, m-1, k);
            hybridQuickSortTest(a, m+1, r, k);
        }
    }
}
```

```
template <typename T>
void hybridSortTest(vector<T>& a, int k) {
    hybridQuickSortTest(a, 0, a.size() - 1, k);
    insertionSort(a, 0, a.size() - 1);
}

vector<int> generateRandomArray(int n) {
    static std::mt19937 gen(std::random_device{}()); // Mersenne Twister engine
    std::uniform_int_distribution<int> dist(0, 10000);

    vector<int> arr(n);
    for (auto &elem : arr) {
        elem = dist(gen);  // high-quality random integer
    }
    return arr;
}

int main() {
    std::ofstream fout("resultsK.csv");
    if (!fout.is_open()) {
        std::cerr << "Error: could not open results.csv\n";
        return 1;
    }

    int totalRuns = 1000;

    // Write header row
    fout << "InputSize";
    for (int run = 1; run <= totalRuns; ++run) {
        fout << ",HybridSort_Run" << run;
    }
    fout << "\n";

    vector<int> ks(100);
    for (int i = 0; i < 100; ++i) ks[i] = i + 1;

    // For each input size, run totalRuns times
    for (int k : ks) {
        fout << k;

        // Collect insertion sort results first
        vector<long long> insertionTimes;
        insertionTimes.reserve(totalRuns);

        vector<long long> quickTimes;
        quickTimes.reserve(totalRuns);

        for (int run = 1; run <= totalRuns; ++run) {
            vector<int> arr = generateRandomArray(1000);

            auto start = high_resolution_clock::now();
            hybridSortTest(arr, k);
```

```
            auto end = high_resolution_clock::now();
            insertionTimes.push_back(duration_cast<microseconds>(end - start).count());
        }

        // Write insertion sort results
        for (auto t : insertionTimes) fout << "," << t;
        // Write quicksort results
        for (auto t : quickTimes) fout << "," << t;

        fout << "\n";
    }

    fout.close();
    std::cout << "Results saved to resultsK.csv\n";
    return 0;
}
```



According to the scattered plot, where each point represents the average running time for the given k, we obtain the formula

$$y = 0.0071x^2 - 0.5507x + 101.15 = 0.0071(x - 38.7817)^2 + 90.4715.$$

The function has a strict minimum value of 90.4715 at $x = 38.7817$. The optimal integer choice of k is 38 or 39.

Implementation of the Hybrid Algorithm

```
#pragma once
#include <vector>
#include <algorithm>

using std::vector;

#define k 36

template <typename T>
void insertionSort(vector<T>& a, int l, int r) {
    int n = r - l + 1;
    for (int j = l; j < r+1; ++j) {
        T key = a[j];
        // insert a[j] into the sorted sequence a[l..j-1]
        int i = j-1;
        while (i >= 0 && a[i] > key) {
            a[i+1] = a[i];
            --i;
        }
        a[i+1] = key;
    }
}

template <typename T>
int partition(vector<T>& a, int l, int r) {
    int pivotIndex = l + rand() % (r - l + 1);
    std::swap(a[pivotIndex], a[r]);

    T pivot = a[r];
    int i = l - 1;
    for (int j = l; j < r; ++j) {
        if (a[j] <= pivot) {
        std::swap(a[++i], a[j]);
        }
    }
    std::swap(a[i+1], a[r]);
    return i+1;
}

template<typename T>
void hybridQuickSort(vector<T>& a, int l, int r) {
    if (l < r) {
        if (r - l + 1< k) return;
        else {
            int m = partition(a, l, r);
            hybridQuickSort(a, l, m-1);
            hybridQuickSort(a, m+1, r);
        }
    }
}

template <typename T>
void hybridSort(vector<T>& a) {
```

```
    hybridQuickSort(a, 0, a.size() - 1, k);
    insertionSort(a, 0, a.size() - 1);
}
```

Discussion

Despite the result, the project is subject to several limitations.

First, the scale of the test is limited. For feasibility, each configuration was only tested 1000 times, which is insufficient to get convincing results. The test was only implemented on a single machine, which limits the generalizability of the results.

In addition, the experiment lacks a precise methodology of measuring the performance of an algorithm. In this experiment, the performance of insertion sort and randomized quicksort is depicted by the running time recorded in millisecond. On a sophisticated operating system like Windows 11, background processes may introduce noise that undermines the precision of millisecond-level timing. The optimization and scheduling of the testing program by the operating system is hardly known from a programmer's perspective. In other words, the performance of the algorithms presented by the experiment merely indicates an ambiguous trend.

Due to these uncontrolled factors, improved statistical analysis is necessary. For example, the outliers need to be removed from the data, instead of being taken into account in the average calculation.

Algorithm analysis puts emphasis on the number of operations that a program makes, independent of the particular machine, compiler optimization and OS scheduling. In the implementation of this project, these factors are scarcely taken into account.

Last but not least, in real-life application, the sequence to sort usually follows a certain pattern. For example, insertion sort is fast when the input is nearly sorted. All experiments in this project were conducted using uniformly distributed random integers.

Conclusion

In the hybrid sorting algorithm, the choice of k is approximately 39, which aligns closely with practical implementations that select k between 10 and 30. A practical approach is to combine the strengths of different basic algorithms to obtain an algorithm that adapts efficiently to diverse input characteristics.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022).
Introduction to Algorithms (4th ed.). MIT Press.

Acknowledgement

ChatGPT-5 provided valuable suggestions about generating random numbers, measuring the running time in C++ and correcting spelling mistakes in my writing.

> https://chatgpt.com/s/t_68d1f697dc9c8191aab672cd50b08c5f
> https://chatgpt.com/s/t_68d1f613548c81918a8bf7a13fd3881b

Appendix : Project specification

1. Write code for insertion sort. (10 points)
2. Write code for mergesort. (10 points)
3. Write code for quicksort. (10 points)
4. The running time of quicksort can be improved in practice by taking advantage of the fast running time of insertion sort when its input is “nearly” sorted. When quicksort is called on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process.
   Argue that this sorting algorithm runs in $O(nk + n\lg(n/k))$ expected time. How should k be picked, both in theory and in practice by experiments? (30 points)
5. Write code for improved version of sorting algorithm which combines quicksort with insertion sort. (20 points)
6. All document for the answers of the above questions. (20 points)