**CONTENT**

[Sorting Problem 2](#_Toc27303827)

[Computation Model 2](#_Toc27303828)

[Background 2](#_Toc27303829)

[1. Traditional quick sorting algorithm 2](#_Toc27303830)

[1.1 Basic idea of quick sorting 2](#_Toc27303831)

[1.2 Three steps for quick sorting 2](#_Toc27303832)

[2. Quick sorting optimization strategy 3](#_Toc27303833)

[2.1 Optimize small array sorting 3](#_Toc27303834)

[2.2 Pivot element selection 3](#_Toc27303835)

[3. Experimental details 3](#_Toc27303836)

[3.1 Array to be sorted 3](#_Toc27303837)

[3.2 Sorting Algorithm 3](#_Toc27303838)

[Result 3](#_Toc27303839)

[Runtime statistics 4](#_Toc27303840)

[Code 5](#_Toc27303841)

[Traditional non-recursive quick sort algorithm 5](#_Toc27303842)

[Optimized non-recursive quick sort algorithm 6](#_Toc27303843)

[Analysis 8](#_Toc27303844)

[Time complexity 8](#_Toc27303845)

# Sorting Problem

The sorting algorithm reorders one or more sets of data according to a certain pattern through a specific algorithm. This new sequence follows certain rules and reflects certain laws. The processed data is easy to filter and calculate, which greatly improves the calculation efficiency.

This article will study the use and optimization of the quick sort algorithm in array sorting problems above 10,000 levels.

# Computation Model

This paper selects the RAM computing model to implement the sorting algorithm and optimize the algorithm. Because the quick sort algorithm uses the in-place sort method, there is no need to create additional storage space, so the memory space is sufficient to run the sort algorithm.

# Background

## 1. Traditional quick sorting algorithm

### 1.1 Basic idea of quick sorting

Quick sorting uses the idea of divide-and-conquer. The sequence to be sorted is divided into two parts by one-pass sorting. The keywords recorded in one part are smaller than the keywords recorded in the other part. Afterwards, the two parts of records are sorted separately to achieve the purpose of ordering the entire sequence.

### 1.2 Three steps for quick sorting

(1) Choose a benchmark: In the to-be-scheduled sequence, pick an element in a certain way as a "pivot"

(2) Segmentation operation: The sequence is divided into two subsequences based on the actual position of the reference in the sequence. At this point, the elements to the left of the datum are smaller than the datum, and the elements to the right of the datum are larger than the datum.

(3) Recursively sort two sequences recursively until the sequence is empty or has only one element.

## 2. Quick sorting optimization strategy

### 2.1 Optimize small array sorting

In the small-scale array sorting problem, the execution efficiency of the insert sort algorithm is higher than that of the traditional fast sort algorithm. Therefore, by adding the insert sort algorithm to the fast sort algorithm, the sort efficiency can be effectively improved. In the optimization algorithm of the fast sorting algorithm studied in this article, a small-scale array refers to a sub-array with less than 10 elements.

### 2.2 Pivot element selection

In traditional sorting algorithms, pivot elements are fixed. When processing completely out-of-order arrays, the sorting efficiency can reach O(nlogn), but when processing basically ordered arrays, the number of recursions reaches O(n^2). Therefore, this article uses ‘take middle from three numbers’ to determine the pivot elements and optimize the ranking algorithm.

## 3. Experimental details

### 3.1 Array to be sorted

Number of elements: In order to compare the operating efficiency of different quick sorting algorithms in array sorting problems of different sizes, this article sets up arrays of 1000, 10000, 100000, 1000000, and 10000000 elements, and executes different quick sorting algorithms to count the execution time.

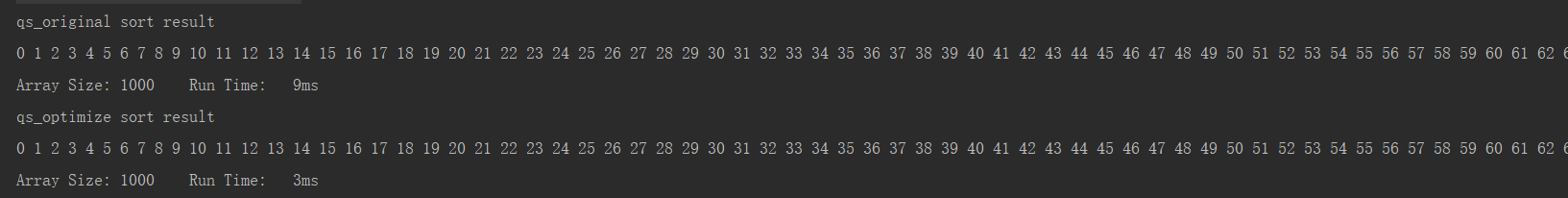
Array type: Because the fast sort algorithm has the worst execution efficiency when dealing with fully ordered array sorting problems, the arrays designed in this article are divided into completely out of order arrays and completely ordered arrays.

### 3.2 Sorting Algorithm

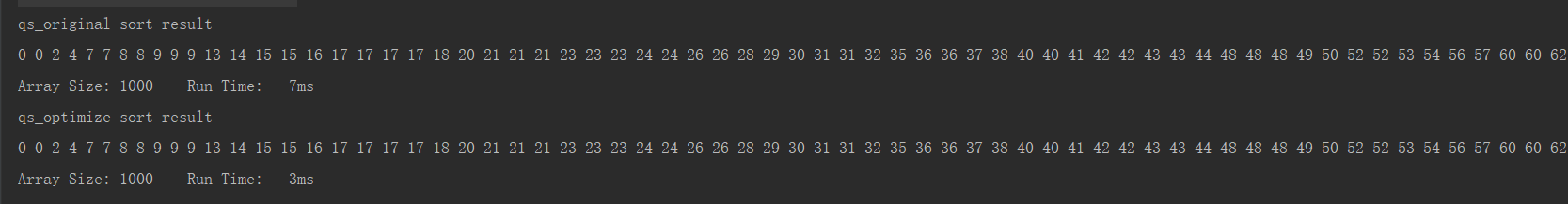
* qs\_original：Non-Recursive Quick Sorting Algorithm
* qs\_optimize：Non-Recursive Quick Sorting Algorithm that including insert sorting algorithms and optimizes pivot elements

# Result

* Sort results of a completely ordered array (take 1000 elements array as an example)



* Sort results of a completely random array (take 1000 elements array as an example)



## Runtime statistics

Out-of-order array sort time statistics：

Figure 1 Out-of-order array sort time statistics

In the out-of-order array sorting experiment, it was found that the optimized fast sorting algorithm is significantly higher than the traditional fast sorting algorithm in processing sorting problems with a large amount of data.

Ordered array sort time statistics：

(The traditional fast sort algorithm is too slow when sorting a fully ordered array, so the sort experiment of 10000000 elements is omitted)

Figure 2 Ordered array sort time statistics

In the fully ordered array sorting experiment, it is found that the traditional quick sorting processing is completely inefficient. The reason is that fixed pivot elements lead to unreasonable subarray partitioning, which leads to excessive recursion depth.

The optimized quick sorting algorithm uses the middle of three numbers to determine the pivot element, which effectively reduces the number of algorithm recursions and improves the execution efficiency.

# Code

## Traditional non-recursive quick sort algorithm

/\*\*

\* Traditional non-recursive quick sort algorithm

\* \*/

public static void qs\_original(int[] arr, int low ,int high){

int pivot;

if (low >= high)

return;

Stack<Integer> stack = new Stack<Integer>();

stack.push(low);

stack.push(high);

while (!stack.empty()) {

// pop high first,then low

high = stack.pop();

low = stack.pop();

pivot = partition(arr,low,high);

// push low first,then high

if (low < pivot - 1) {

stack.push(low);

stack.push(pivot - 1);

}

if (pivot + 1 < high) {

stack.push(pivot + 1);

stack.push(high);

}

}

}

/\*\*

\* Traditional quick sorting algorithm for array partitioning

\* \*/

public static int partition(int[] a, int low, int high) {

int pivotKey = a[low]; // Use the first element as the Pivot element

while (low < high) { // Scan from both sides to the middle

while (low < high && a[high] >= pivotKey)

high--;

a[low] = a[high];

while (low < high && a[low] <= pivotKey)

low++;

a[high] = a[low];

}

a[low] = pivotKey; // Put the reference value back in the middle

return low; // Returns the position of the pivot element

}

## Optimized non-recursive quick sort algorithm

/\*\*

\* Optimized non-recursive quick sort algorithm

\* Optimization Strategy：

\* 1. Use insert sort to handle small array sort

\* 2. Use partition\_optimaze as partition method

\* \*/

public static void qs\_optimize(int[] arr,int low,int high){

if(low>high){

return;

}

//use insert sort when array size is less than 10

if(high-low<10){

insertsort(arr,low,high);

return;

}

int pivot;

if (low >= high)

return;

Stack<Integer> stack = new Stack<Integer>();

stack.push(low);

stack.push(high);

while (!stack.empty()) {

// pop high first,then low

high = stack.pop();

low = stack.pop();

pivot = partition\_optimaze(arr,low,high);

// push low first,then high

if (low < pivot - 1) {

stack.push(low);

stack.push(pivot - 1);

}

if (pivot + 1 < high) {

stack.push(pivot + 1);

stack.push(high);

}

}

}

/\*\*

\* Optimized quick sort algorithm for array partitioning

\* \*/

public static int partition\_optimaze(int[] a, int low, int high) {

// Take three numbers and place the middle element in the first position

if (a[low] > a[high])

swap(a, low, high);

if (a[(low + high) / 2] > a[high])

swap(a, (low + high) / 2, high);

if (a[low] < a[(low + high) / 2])

swap(a, (low + high) / 2, low);

int pivotKey = a[low]; // Use the first element as the Pivot element

while (low < high) { // Scan from both sides to the middle

while (low < high && a[high] >= pivotKey)

high--;

a[low] = a[high];

while (low < high && a[low] <= pivotKey)

low++;

a[high] = a[low];

}

a[low] = pivotKey; // Put the reference value back in the middle

return low; // Returns the position of the pivot element

}

/\*\*

\* swap elements in a

\* \*/

public static void swap(int[] a, int i, int j) {

int temp;

temp = a[j];

a[j] = a[i];

a[i] = temp;

}

/\*\*

\* Insert Sort,used when size of array is less than 10

\* \*/

public static void insertsort(int[] arr,int left,int right){

if(left>right)

return;

int min = Integer.MAX\_VALUE;

int index=-1;

int tmp=0;

for(int i=left;i<=right;i++){

for(int j=i;j<=right;j++){

if(arr[j]<min){

min=arr[j];

index = j;

}

}

tmp = arr[i];

arr[i]=min;

arr[index]=tmp;

min = Integer.MAX\_VALUE;

}

}

# Analysis

## Time complexity

The best case for quick sorting is that each time the element is taken, the entire array is evenly divided. The time complexity formula at this time is: T [n] = 2T[n/2] + f(n), T[n/2] is the time complexity of the sub-array after the bisect, and f[n] is the time it takes to bisect this array. In the optimal state, the time complexity is O(nlogn).

The worst case of traditional quick sorting is to select the smallest or largest element as the pivot element, and the execution efficiency is the same as the bubble sort: O(n^2).

This article optimizes the traditional fast sort algorithm from two perspectives:

1. Small array sorting uses direct insert sorting instead of quick sorting;

2. Optimize the selection of pivot elements. Instead of using fixed positions to select pivot elements, use ‘middle of three numbers’ to ensure that the pivot elements can divide the array into sub-arrays of similar size, reducing the number of recursions.

The optimized fast sort algorithm designed in this paper can basically reach O (nlogn) in the best and worst cases.