数据结构

第六次上机作业实验文档

学号2017211123

班级 2017211301

序号 17

姓名 褚逸豪

## **问题描述**

输入若干组长度各异的待排序列，分别用快速排序算法和改进的枢轴元素三者取中算法对待排序列进行排序，当待排子序列长度已小于20时，改用直接插入排序，利用时间函数验证三者取中算法在效率上的提高。

## 算法思路

对普通的快速排序进行适当改进即可

## 算法描述

1. 普通的快速排序（略）
2. 改进过的快速排序
   1. 函数 ts（参数1 待排序列， 参数2 起始位置， 参数3 结束位置）：
      1. 如果结束位置-开始位置<20，对当前序列进行插入排序，并退出当前函数
      2. 找到待排序列开始位置元素，结束位置元素和序列正中的元素三者的中位数为t
      3. 以t为参考元素调整序列，小于t的元素在左侧，大于等于t的元素在右侧
      4. 将t放到序列中间
      5. 对t左侧的序列和t右侧的序列分别递归调用函数ts

## 源程序及驱动程序

编译命令：g++ <文件名.cpp> -std=c++11

#include <cstdio>

#include <cstring>

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <chrono>

#include <algorithm>

typedef std::chrono::high\_resolution\_clock Clock;

using namespace std;

namespace Qsort {

int tarr[27];

void partion\_sort(int seq[], int l, int r) { // 每次选择区间右端点作为参照

if (l >= r) return;

int p = r; // 定基准

int tl = l, tr = r;

while (tl < tr) {

while (tl < tr && seq[tl] < seq[p]) ++tl;

while (tl < tr && seq[tr] >= seq[p]) --tr;

swap(seq[tl], seq[tr]);

}

swap(seq[p], seq[tl]);

partion\_sort(seq, l, tl - 1);

partion\_sort(seq, tl + 1, r);

}

void tripivot\_sort(int seq[], int l, int r) { // 枢轴三者取中方法，序列长度低于20直接插排

if (l >= r) return;

if (r - l < 20) { // 插排部分

memset(tarr, 0x7f, sizeof(tarr));

for (int i = l, j = 0; i <= r; ++i) {

for (j = 0; j < i - l; ++j) {

if (tarr[j] >= seq[i]) break;

}

for (int t = i - l; t > j; --t)

tarr[t] = tarr[t - 1];

tarr[j] = seq[i];

}

for (int i = l; i <= r; ++i)

seq[i] = tarr[i - l];

} else { // 先找中值

int M = l, m = l;

if (seq[r] < seq[m]) m = r;

if (seq[(l + r) >> 1] < seq[m]) m = (l + r) >> 1;

if (seq[r] > seq[M]) M = r;

if (seq[(l + r) >> 1] > seq[M]) M = (l + r) >> 1;

int p = (l + r) >> 1;

if (m != l && M != l) p = l;

if (m != r && M != r) p = r;

swap(seq[p], seq[r]);

p = r;

int tl = l, tr = r;

while (tl < tr) {

while (tl < tr && seq[tl] < seq[p]) ++tl;

while (tl < tr && seq[tr] >= seq[p]) --tr;

swap(seq[tl], seq[tr]);

}

swap(seq[p], seq[tl]);

tripivot\_sort(seq, l, tl - 1);

tripivot\_sort(seq, tl + 1, r);

}

}

}

int tmp[2][100007];

int main() {

int n, seed;

scanf("%d %d", &n, &seed);

srand(seed);

for (int i = 0; i < n; ++i)

tmp[0][i] = tmp[1][i] = rand();

auto t1 = Clock::now();

decltype(t1) t2, t3;

Qsort::partion\_sort(tmp[0], 0, n - 1); // 测量随机序列

t2 = Clock::now();

Qsort::tripivot\_sort(tmp[1], 0, n - 1);

t3 = Clock::now();

printf("Normal qsort costs :%ld ns\nTripivot sort costs :%ld ns\n", chrono::duration\_cast<std::chrono::nanoseconds>(t2 - t1).count(), chrono::duration\_cast<std::chrono::nanoseconds>(t3 - t2).count());

reverse(tmp[0], tmp[0] + n);

reverse(tmp[1], tmp[1] + n);

t1 = Clock::now();

Qsort::partion\_sort(tmp[0], 0, n - 1); // 测量递减序列

t2 = Clock::now();

Qsort::tripivot\_sort(tmp[1], 0, n - 1);

t3 = Clock::now();

printf("Normal qsort costs(decreasing sequence) :%ld ns\nTripivot sort costs(decreasing sequence) :%ld ns\n", chrono::duration\_cast<std::chrono::nanoseconds>(t2 - t1).count(), chrono::duration\_cast<std::chrono::nanoseconds>(t3 - t2).count());

}

## 测试数据

### Input

100000 2017211301

### Output

Normal qsort costs :26510400 ns

Tripivot sort costs :25795300 ns

Normal qsort costs(decreasing sequence) :16150381700 ns

Tripivot sort costs(decreasing sequence) :3808353300 ns

### Online Test

<https://ideone.com/xIgYLg>

## 结果分析

在序列完全随机的情况下，普通快速排序与改进的快排差别不大，但当使用worst case进行测试时（如递减序列），改进快排优势巨大

## 结论

枢轴三元素取中快排在一些具有特定特点的数据中效率远高于未经优化的快排

## 心得体会

测时的clock函数在不同平台下特性不太一样，有时候甚至不能正常工作？原因未知