Тихобаев Илья

18ПИ-1

Алгоритмы и структуры данных.

Лабораторная работа №1.

Сортировки.

**Цель работы:** реализовать алгоритмы некоторых сортировок и сравнить их время выполнения на контейнерах разной длины, проверить корректность алгоритма.

**Ход работы:**

В ходе работы на языке C++ были написаны алгоритмы сортировок, представленные в таблице. Сортировки были проверены на векторах с элементами (тип double) в диапазоне [0, 100]. Каждая сортировка тестировалась на векторах, содержащих 10, 100, 1000, 10000, 100000 элементов (50 массивов каждого размера). После сортировки циклом for была проверена корректность сортировки. Замеры времени усреднены.

Ниже представлена таблица времени различных алгоритмов сортировок на разном количестве входных данных. Время представлено в секундах.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 10^1 | 10^2 | 10^3 | 10^4 | 10^5 |
| **Selection** | 0 | 0 | 0.0014 | 0.1252 | 12.3572 |
| **Merge** | 0 | 6e-005 | 0.00056 | 0.00514 | 0.04922 |
| **Bucket** | 0 | 4e-005 | 0.00022 | 0.0026 | 0.02328 |
| **Quick(rand)** | 0 | 0 | 0.00012 | 0.00192 | 0.01954 |
| **Quick(worst)** | 0 | 0 | 0.0022 | 0.2132 | Stack overflow |
| **Tim** | 0 | 0 | 0.0002 | 0.0027 | 0.02816 |

На основании таблицы был получен график:

На графике для наглядности для алгоритма selection sort на 10^5 элементов время специально уменьшено с 12,3572 секунд до 0.22 секунды.

Ниже представлены графики для всех сортировок по отдельности.

**Ниже представлен код сортировок и вспомогательных функций:**

**Функция генерации случайного вектора:**

void getTestVectorDouble(vector<double>& arr, int max, int num){  
 srand(time(NULL));  
 for (int i = 0; i < num; i++){  
 arr.push\_back(((double)rand() / RAND\_MAX) \* max);  
 }  
}

**Функции для тестирования:**

Для каждой сортировки для каждого размера создаются по 50 векторов, для каждого замеряется время и складывается в переменную, хранящую время(которая потом будет поделена на 50 будет хранить в себе усреднённое время работы сортировки на данном размере вектора), затем циклом сортировка проверяется на корректность, то есть на отсортированность по возрастанию.

void selectionSortTest(){  
 vector <double> arr;  
  
 cout << " Selection sort:" << endl;  
 for (int n = 10; n <= 100000; n \*= 10){  
 cout << endl << n << " elements " << endl;  
 double midTime = 0;  
 int k = 50; // count of test arrays  
 for (int i = 0; i < k; i++){  
 getTestVectorDouble(arr, 100, n);  
  
 double start = clock();  
 selectionSort(arr, n);  
 double end = clock();  
  
 midTime += (end - start) / CLOCKS\_PER\_SEC;  
 for (int c = 0; c < n - 1; c++){  
 if (arr[c] > arr[c + 1]){  
 cout << endl << "Error: there are unsorted elements!!!" << endl;  
 }  
 }  
 arr.clear();  
 }  
 cout << " time: " << midTime / k << " sec.";  
 }  
}  
  
void bucketSortTest(){  
 vector <double> arr;  
 cout << " Bucket sort:" << endl;  
 for (int n = 10; n <= 100000; n \*= 10){  
 cout << endl << n << " elements " << endl;  
 double midTime = 0;  
 int k = 50; // count of test arrays  
 for (int i = 0; i < k; i++){  
 getTestVectorDouble(arr, 100, n);  
 double start = clock();  
 bucketSort(arr, n);  
 double end = clock();  
 midTime += (end - start) / CLOCKS\_PER\_SEC;  
 for (int c = 0; c < n - 1; c++){  
 if (arr[c] > arr[c + 1]){  
 cout << endl << "Error: there are unsorted elements!!!" << endl;  
 }  
 }  
 arr.clear();  
 }  
 cout << " time: " << midTime / k << " sec.";  
 }  
}  
  
  
void quickSortTest(){  
 vector <double> arr;  
  
 cout << " Quick sort(random pivot):" << endl;  
 for (int n = 10; n <= 100000; n \*= 10){  
 cout << endl << n << " elements " << endl;  
 double midTime = 0;  
 int k = 50; // count of test arrays  
 for (int i = 0; i < k; i++){  
 getTestVectorDouble(arr, 100, n);  
 double start = clock();  
 quickSort(arr, 0, n - 1);  
 double end = clock();  
  
 midTime += (end - start) / CLOCKS\_PER\_SEC;  
 for (int c = 0; c < n - 1; c++){  
 if (arr[c] > arr[c + 1]){  
 cout << endl << "Error: there are unsorted elements!!!" << endl;  
 }  
 }  
 arr.clear();  
 }  
 cout << " time: " << midTime / k << " sec.";  
 }  
  
 cout << endl <<" Quick sort(worst pivot):" << endl;  
 for (int n = 10; n <= 100000; n \*= 10){  
 cout << endl << n << " elements " << endl;  
 double midTime = 0;  
 int k = 50; // count of test arrays  
 for (int i = 0; i < k; i++){  
 getTestVectorDouble(arr, 100, n);  
 double start = clock();  
 quickSortWorstPivot(arr, 0, n - 1);  
 double end = clock();  
  
 midTime += (end - start) / CLOCKS\_PER\_SEC;  
 for (int c = 0; c < n - 1; c++){  
 if (arr[c] > arr[c + 1]){  
 cout << endl << "Error: there are unsorted elements!!!" << endl;  
 }  
 }  
 arr.clear();  
 }  
 cout << " time: " << midTime / k << " sec.";  
 }  
}  
  
void mergeSortTest(){  
 vector <double> arr;  
  
 cout << " Merge sort:" << endl;  
 for (int n = 10; n <= 100000; n \*= 10){  
 cout << endl << n << " elements " << endl;  
 double midTime = 0;  
 int k = 50; // count of test arrays  
 for (int i = 0; i < k; i++){  
 getTestVectorDouble(arr, 100, n);  
  
 double start = clock();  
 mergeSort(arr, 0, n - 1);  
 double end = clock();  
  
 midTime += (end - start) / CLOCKS\_PER\_SEC;  
 for (int c = 0; c < n - 1; c++){  
 if (arr[c] > arr[c + 1]){  
 cout << endl << "Error: there are unsorted elements!!!" << endl;  
 }  
 }  
 arr.clear();  
 }  
 cout << " time: " << midTime / k << " sec.";  
 }  
  
}  
  
  
void timSortTest(){  
 vector <double> arr;  
  
 cout << " Tim sort:" << endl;  
 for (int n = 10; n <= 100000; n \*= 10){  
 cout << endl << n << " elements " << endl;  
 double midTime = 0;  
 int k = 50; // count of test arrays  
 for (int i = 0; i < k; i++){  
 getTestVectorDouble(arr, 100, n+1);  
  
 double start = clock();  
 timSort(arr, n);  
 double end = clock();  
  
 midTime += (end - start) / CLOCKS\_PER\_SEC;  
 for (int c = 0; c < n - 1; c++){  
 if (arr[c] > arr[c + 1]){  
 cout << endl << "Error: there are unsorted elements!!!" << endl;  
 }  
 }  
 arr.clear();  
 }  
 cout << " time: " << midTime / k << " sec.";  
 }  
}

**Сортировки.**

**1. Selection sort**

void selectionSort(vector<double>& arr, int n){  
 for (int i = 0; i < n; i++){  
 double min = arr[i];  
 int minIndex = i;  
 for (int j = i; j < n; j++){  
 if (arr[j] < min){  
 min = arr[j];  
 minIndex = j;  
 }  
 }  
 double tmp = arr[i];  
 arr[i] = min;  
 arr[minIndex] = tmp;  
 }  
}

**2. Merge sort**

void mergeSort(vector<double>& arr, int l, int r)  
{  
 if (l < r)  
 {  
 // Same as (l+r)/2, but avoids overflow for  
 // large l and h  
 int m = l + (r - l) / 2;  
  
 // Sort first and second halves  
 mergeSort(arr, l, m);  
 mergeSort(arr, m + 1, r);  
  
 merge(arr, l, m, r);  
 }  
}  
  
void merge(vector<double>& arr, int l, int m, int r)  
{  
 int n1 = m - l + 1;  
 int n2 = r - m;  
  
 /\* create temp arrays \*/  
 double\* L = new double[n1];  
 double\* R = new double[n2];  
  
 /\* Copy data to temp arrays L[] and R[] \*/  
 for (int i = 0; i < n1; i++)  
 L[i] = arr[l + i];  
 for (int j = 0; j < n2; j++)  
 R[j] = arr[m + 1 + j];  
  
 /\* Merge the temp arrays back into arr[l..r]\*/  
 i = 0; // Initial index of first subarray  
 j = 0; // Initial index of second subarray  
 int k = l; // Initial index of merged subarray  
 while (i < n1 && j < n2)  
 {  
 if (L[i] <= R[j])  
 {  
 arr[k] = L[i];  
 i++;  
 }  
 else  
 {  
 arr[k] = R[j];  
 j++;  
 }  
 k++;  
 }  
  
 /\* Copy the remaining elements of L[], if there  
 are any \*/  
 while (i < n1)  
 {  
 arr[k] = L[i];  
 i++;  
 k++;  
 }  
  
 /\* Copy the remaining elements of R[], if there  
 are any \*/  
 while (j < n2)  
 {  
 arr[k] = R[j];  
 j++;  
 k++;  
 }  
 delete[]R;  
 delete[]L;  
}

**3. Bucket sort**

В данной сортировке количество вёдер = N/3, где N – количество элементов в векторе. Сначала элементы распределяются по вёдрам, в зависимости от их значения, а затем сортируются встроенной сортировкой std::sort().

void bucketSort(vector<double>& arr, int n)  
{  
 struct bucket buckets[n / 3];  
 int i, j, k;  
 for (i = 0; i < n / 3; i++)  
 {  
 buckets[i].count = 0;  
 }  
 double max = arr[0], min = arr[0];  
 for(int i = 0; i < n; i++){  
 if (arr[i] < min){  
 min = arr[i];  
 }  
 if (arr[i] > max){  
 max = arr[i];  
 }  
 }  
 for (i = 0; i < n; i++)  
 {  
 int bucketNum = int((arr[i] - min) / ((max - min) / (n/3)));  
 if (bucketNum >= n/3) {  
 bucketNum = n/3 - 1;  
 }  
 buckets[bucketNum].value.push\_back(arr[i]);  
 buckets[bucketNum].count++;  
 }  
 for (k = 0, i = 0; i < n/3; i++)  
 {  
 // now using quicksort to sort the elements of buckets  
 sort(buckets[i].value.begin(), buckets[i].value.end());  
 for (j = 0; j < buckets[i].count; j++)  
 {  
 arr[k + j] = buckets[i].value[j];  
 }  
 k += buckets[i].count;  
 }  
}

**4. Quick sort (random pivot)**

В качестве рандомного элемента берётся первый.

void quickSort(vector<double>& arr, int left, int right)  
{  
 double pivot;  
 int l\_hold = left;  
 int r\_hold = right;  
 pivot = arr[left];  
 while (left < right)  
 {  
 while ((arr[right] >= pivot) && (left < right))  
 right--;  
 if (left != right)  
 {  
 arr[left] = arr[right];  
 left++;  
 }  
 while ((arr[left] <= pivot) && (left < right))  
 left++;  
 if (left != right)  
 {  
 arr[right] = arr[left];  
 right--;  
 }  
 }  
 arr[left] = pivot;  
 pivot = left;  
 left = l\_hold;  
 right = r\_hold;  
 if (left < pivot)  
 quickSort(arr, left, pivot - 1);  
 if (right > pivot)  
 quickSort(arr, pivot + 1, right);  
}

**5. Quick sort(worst pivot)**

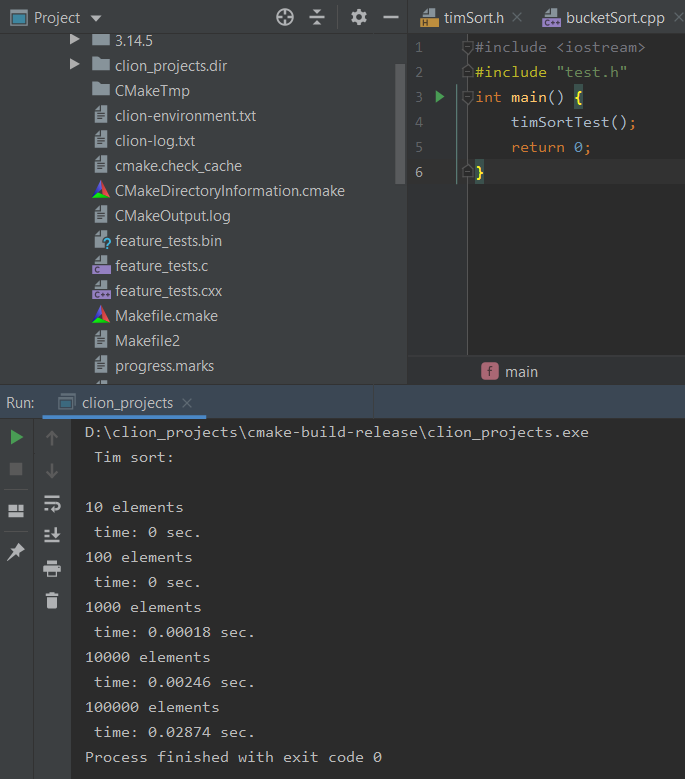
В качестве худшего элемента выступает максимальный.

void quickSortWorstPivot(vector<double>& arr, int left, int right)  
{  
 double pivot;  
 int l\_hold = left;  
 int r\_hold = right;  
 double maxElem = arr[left];  
 int maxInd = left;  
 for (int i = left; i <= right; i++){  
 if (arr[i] > maxElem){  
 maxElem = arr[i];  
 maxInd = i;  
 }  
 }  
 double tmp = arr[left];  
 arr[left] = maxElem;  
 arr[maxInd] = tmp;  
 pivot = arr[left];  
 while (left < right)  
 {  
 while ((arr[right] >= pivot) && (left < right))  
 right--;  
 if (left != right)  
 {  
 arr[left] = arr[right];  
 left++;  
 }  
 while ((arr[left] <= pivot) && (left < right))  
 left++;  
 if (left != right)  
 {  
 arr[right] = arr[left];  
 right--;  
 }  
 }  
 arr[left] = pivot;  
 pivot = left;  
 left = l\_hold;  
 right = r\_hold;  
 if (left < pivot)  
 quickSortWorstPivot(arr, left, pivot - 1);  
 if (right > pivot)  
 quickSortWorstPivot(arr, pivot + 1, right);  
}

**6. Timsort**

int getMinrun(int n) {  
 int r = 0;  
 while (n >= 64) {  
 r |= n & 1;  
 n >>= 1; // == n / 2  
 }  
 return n + r;  
}  
  
void insertionSort(vector<double> &arr, int start, int end) {  
 for (int i = start + 1; i < end; ++i) {  
 double current = arr[i];  
 int j = i - 1;  
 while ((j >= start) && (current < arr[j])) {  
 swap(arr[j], arr[j + 1]);  
 j--;  
 }  
 }  
}  
  
struct run {  
 int start;  
 int length;  
};  
  
void merge(vector<double> &arr, int fStart, int fLength, int sStart, int sLength) {  
 vector<double> tmp;  
 for(int i = 0; i < fLength; i++) {  
 tmp.push\_back(arr[fStart + i]);  
 }  
 int fIndex = 0;  
 int sIndex = sStart;  
 int i = fStart;  
 int firstStart = fStart;  
 int secondStart = sStart;  
 int tmpCount = 0, secondCount = 0;  
 while (i < sStart + sLength - 1) {  
 if (arr[sIndex] > tmp[fIndex]) {  
 arr[i] = tmp[fIndex];  
 ++fIndex;  
 tmpCount++;  
 secondCount = 0;  
 fIndex = std::min(fIndex, fLength);  
 } else if (arr[sIndex] <= tmp[fIndex]) {  
 arr[i] = arr[sIndex];  
 secondCount++;  
 tmpCount = 0;  
 ++sIndex;  
 sIndex = std::min(sIndex, sStart + sLength);  
 }  
 if (sIndex == sStart + sLength) {  
 if (fIndex != fLength) {  
 for (int d = 0; d < fLength - fIndex; d++) {  
 arr[i + 1 + d] = tmp[fIndex + d];  
 }  
 i = sStart + sLength;  
 }  
 }  
 if (fIndex == fLength) {  
 i = sStart + sLength;  
 }  
 ++i;  
 }  
}  
  
  
void timSort(vector<double> &arr, int len) {  
//Count minrun  
 int minrun = getMinrun(len);  
  
 vector<run> minrunStack;  
  
//divide into runs  
 int currentIndex = 0;  
  
 while (currentIndex < len - 1) {  
 int currentRunIndex = currentIndex;  
 int nextIndex = currentRunIndex + 1;  
  
 if (arr[currentRunIndex] > arr[nextIndex]) {  
 swap(arr[currentRunIndex], arr[nextIndex]);  
 }  
  
 bool goNext = true;  
 while (goNext) {  
 if (arr[currentRunIndex] > arr[nextIndex])  
 goNext = false;  
 if (!(goNext) && (currentRunIndex - currentIndex < minrun))  
 goNext = true;  
 if (currentRunIndex == len - 1)  
 goNext = false;  
 if (goNext) {  
 currentRunIndex++;  
 nextIndex++;  
 }  
 }  
 insertionSort(arr, currentIndex, currentRunIndex + 1);  
  
 run r;  
 r.start = currentIndex;  
 r.length = currentRunIndex - currentIndex;  
  
 minrunStack.push\_back(r);  
  
 currentIndex = currentRunIndex;  
 }  
  
//merge runs  
 while (minrunStack.size() > 1) {  
 run x = minrunStack.at(0);  
 minrunStack.erase(minrunStack.begin());  
 run y = minrunStack.at(0);  
 minrunStack.erase(minrunStack.begin());  
 if (x.start > y.start) {  
 swap(x.start, y.start);  
 swap(x.length, y.length);  
 }  
 if (y.start != x.start + x.length) {  
 minrunStack.push\_back(y);  
 minrunStack.push\_back(x);  
 continue;  
 }  
 merge(arr, x.start, x.length, y.start, y.length);  
 x.length = x.length + y.length;  
 minrunStack.push\_back(x);  
 }  
}

Проект собирается и работает без ошибок.



**Вывод:** Наиболее эффективной из всех сортировок оказалась сортировка quick sort с рандомным опорным элементом, немного уступает ей по времени сортировки bucket sort, ещё чуть менее быстрой оказалась timsort, далее по скорости идёт merge sort, затем selection, а самой неэффективной сортировкой оказалась quick sort с самым плохим(максимальным) опорным элементом, на больших данных она и вовсе ломает программу, вызывая переполнение стека.