**Лабораторная работа №4**

**Тема**: Сложные алгоритмы сортировки. Быстрая сортировка Хоара

**Цель**: Научиться реализовывать сложные алгоритмы сортировки

Быстрая сортировка Хоара (англ. Quicksort) - алгоритм сортировки, разработанный английским информатики Чарльзом Хоаром в 1960 г.. Один из быстрых универсальных алгоритмов сортировки массивов, хотя и имеет ряд недостатков.

**Краткое описание алгоритма:**

**1 Шаг** выбрать базовый элемент.

**2 Шаг** сравнить все остальные элементы с базовым, на основе сравнения разбить множество элементов на три - «меньше, чем базовый», «уровне базовом» и «больше, чем базовый» и расположить их в порядке меньшие-уровне-больше.

**3 Шаг** повторить рекурсивно для «меньших» и «больших» элементов.

Примечание: на практике обычно разделяют сортуему множество элементов не на три, а на две части: например, «меньше» и «равные и больше». Такой подход в общем случае оказывается эффективнее, так как для осуществления такого разделения достаточно одного прохода по сортуемий множестве и однократного обмена только некоторых избранных элементов.

**Детальное описание алгоритма**

Быстрая сортировка использует стратегию «Разделяй и властвуй». Шаги алгоритма следующие:

**1 Шаг**. Выбираем в массиве некоторый элемент, будем называть базовым элементом. С точки зрения корректности работы алгоритма способ выбора базового элемента может быть любым. С точки зрения повышения эффективности алгоритма выбираться должна медиана, но без дополнительных сведений о сортуеми данные ее обычно невозможно получить.

**Известные стратегии выбора базового элемента:**

- Выбирать постоянно один и тот же элемент, например, средний или последний по расположению;

- Выбирать элемент с случайно выбранным индексом.

**2 Шаг**. Операция разделения массива: реорганизуя массив таким образом, чтобы все элементы, меньшие или равные базовому элементу, оказались слева от него, а все элементы, большие базового - справа от него.

**Обычный алгоритм операции:**

1. Два индексы - l (left) и r (right), приравниваются к минимальному и максимальному индексу разделяемая массива соответственно.

2. Вычисляется индекс базового элемента m (medium).

3. Индекс l последовательно увеличивается до тех пор, пока l-й элемент не окажется большим или равным базовому.

4. Индекс r последовательно уменьшается до тех пор, пока r-й элемент не окажется меньше или равным базовому.

5. Если r = l - найдена середина массива - операция разделения закончена, оба индекса указывают на базовый элемент.

6. Если l < r - найденную пару элементов нужно обменять местами и продолжить операцию разделения с тех значений l и r, которые были достигнуты. Следует учесть, что если какая-либо граница (l или r) дошла до базового элемента, то при обмене значение m изменяется на r-й или l-й элемент соответственно.

**3 Шаг**. Рекурсивно упорядочиваем подмассивы, лежащие слева и справа от базового элемента. Базой рекурсии являются наборы, состоящие из одного или двух элементов. Один элемент возвращается в неизменном виде, а в случае двух сортировка сводится к перестановке двух элементов. Все такие отрезки уже упорядочены в процессе разделения массива.

Поскольку в каждой итерации (на каждом следующем уровне рекурсии) длина обрабатываемого отрезка массива уменьшается минимум на единицу, терминальная ветвь рекурсии будет достигнута всегда и обработка гарантированно завершится.

**Оценка эффективности алгоритма**

Сортировка Хоара является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка» и «шейкерные сортировка»), известного, в том числе, своей низкой эффективностью. Принципиальное отличие заключается в том, что после каждого прохода элементы разделяются на две независимые группы. Интересный факт: улучшение самого неэффективного прямого метода сортировки дало в результате эффективный улучшенный метод.

Быстрая сортировка обладает целым рядом недостатков. Его эффективность проявляется не для всех массивов, например, если выбранное базовое значение оказывается совпадающим с максимальным значением, то быстрая сортировка превращается в медленнее.

**Лучший случай**. Для этого алгоритма лучший случай - если в каждой итерации каждый из подмассив делился бы на два равных по величине массива. В результате количество сравнений, предпринимаемые быстрой сортировкой, была бы равна значению рекурсивного выражения CN = 2CN / 2 + N, в явном выражении дает примерно N lg N сравнений. Это дало бы наименьшее время сортировки.

**Средний случай**. Дает в среднем O (n log n) обменов при упорядочении n элементов. В реальности именно такая ситуация обычно имеет место при случайном порядке элементов и выборе базового элемента из середины массива или случайно.

На практике (в случае, когда обмены более затратной операцией, чем сравнение) быстрая сортировка значительно быстрее, чем другие алгоритмы оценки O (n lg n), потому что внутренний цикл алгоритма может быть эффективно реализован почти на любой архитектуре . 2CN / 2 покрывает расходы по сортировке двух полученных подмассив; N - это стоимость обработки каждого элемента, используя один или другой указатель. Известно также, что приблизительное значение этого выражения равен CN = N lg N.

**Худший случай**. Хуже случаем, видимо, будет такой, при котором на каждом этапе массив будет разделяться на вырожденный под массив из одного базового элемента и на подмассив из всех других элементов. Такое может произойти, если в качестве базового на каждом этапе будет выбран элемент или маленький, или самый большой из всех обрабатываемых.

Худой конец дает O (n²) обменов. Но количество обменов и, соответственно, время работы - это не самый его недостаток. Хуже то, что в таком случае глубина рекурсии при выполнении алгоритма достигнет n, что будет означать n-кратное сохранения адреса возврата и локальных переменных процедуры разделения массивов. Для больших значений n худший случай может привести к исчерпанию памяти во время работы алгоритма. Впрочем, на большинстве реальных данных можно найти решения, которые минимизируют вероятность того, что понадобится квадратичный время.

**улучшение:**

1. Выбирать базовый элемент случайным образом, тогда худший случай становится очень маловероятным, а ожидаемое время выполнения алгоритма сортировки - O (n lg n).

2. Выбирать базовым элементом средний из трех (первого, среднего и последнего элементов). Такой выбор также направлен против наихудшего случая.

3. Чтобы избежать достижения опасной глубины рекурсии в худшем случае (или при приближении к нему) возможна модификация алгоритма, устраняет одну ветку рекурсии: вместо того, чтобы после распределения массива вызвать рекурсивно процедуру распределения для обоих найденных подмассив, рекурсивный вызов делается только для меньшего подмассив, а больше обрабатывается в цикле в пределах этого же вызова процедуры. С точки зрения эффективности в среднем случае разницы практически нет: накладные расходы на дополнительный рекурсивный вызов и на организацию сравнения длин подмассив и цикла - примерно одного порядка. Зато глубина рекурсии ни при каких условиях не превысит log2n, а в худшем случае вырожденного распределения она вообще будет не более 2 - вся обработка пройдет в цикле первого уровня рекурсии.

4. Разбивать массив не на две, а на три части.

Пример. Рассмотрим пример, в котором как базой элемент избираются крайние элементы массива.

Дан массив:

{9,6,3,4,10,8,2,7}

Берем 9 как базовый элемент. Сравниваем 9 с противоположно расположенным элементом, в данном случае это 7.

7 меньше, чем 9, следовательно элементы меняются местами.

{7,6,3,4,10,8,2,9}

Далее начинаем последовательно сравнивать элементы из 9, и менять их местами в зависимости от сравнения.

{7,6,3,4,10,8,2,9}

{7,6,3,4,10,8,2,9}

{7,6,3,4,10,8,2,9}

{7,6,3,4,9,8,2,10} - 9 и 10 меняем местами.

{7,6,3,4,8,9,2,10} - 9 и 8 меняем местами.

{7,6,3,4,8,2,9,10} - 2 и 9 меняем местами.

После такого опрокидывания элементов весь массив разбивается на два подмножества разделены элементом 9.

1 подмножество {7,6,3,4,8,2}

2 подмножество {10}

Далее по уже отработанному алгоритму сортируются эти подмножества. Подмножество, состоящее из одного элемента, конечно же сортировать не нужно. Выбираем в первой подмножестве базовый элемент 7.

{7,6,3,4,8,2}

{2,6,3,4,8,7} - меняем местами 2 и 7.

{2,6,3,4,8,7}

{2,6,3,4,8,7}

{2,6,3,4,8,7}

{2,6,3,4,7,8} - меняем местами 7 и 8

Получили снова два подмножества.

{2,6,3,4}

{8}

А дальше все происходит аналогично.

**Лабораторная работа №4**

**Тема**: Сложные алгоритмы сортировки. Быстрая сортировка Хоара

**Цель**: Научиться реализовывать сложные алгоритмы сортировки

Задача: Реализовать алгоритм быстрой сортировки Хоара. Протестировать для лучшего, среднего и худшего случае входных данных.

Дополнительное задание: Реализовать один из перечисленных ниже алгоритмов и сравнить его скорость с быстрой сортировкой Хоара.

**1. Сортировка Шелла**.

2. Пирамидальное сортировки.

3. Поразрядное сортировки.

4. Карманное сортировки.

5. Сортировка подсчетом.

6. Сортировка ячейками.

7. Сортировка расческой.

8. Сортировка перемешиванием.

**//Листинг Prog\_Lab\_4\_1**

#include <iostream>

#include <ctime>

using namespace std;

unsigned int \_time; // переменная для хранения значения времени

void quick\_sort(int \*a, int c); // прототип функции

int main()

{

setlocale(LC\_ALL, "rus"); // локализация

srand(time(0)); // автоматизация рандома

int count\_of\_array; // количество элементов в массиве

// считывание размера массива

cout << "Enter count of array ";

cin >> count\_of\_array;

int qs\_array[count\_of\_array]; // целочисленный массив

// заполнение массива случайными числами

for (int i=0; i < count\_of\_array; i++)

qs\_array[i] = 1+rand()%100;

// отображение массива в не отсортированном виде

for (int i=0; i<count\_of\_array; i++)

cout << qs\_array[i] << ",";

// вызов функции быстрой сортировки

quick\_sort(qs\_array, count\_of\_array);

// время затраченное для сортировки массива размера N

cout <<"\n\n" <<(float)\_time/CLOCKS\_PER\_SEC << endl;

cout<<endl;

// отображение отсортированного массива

for (int i=0; i<count\_of\_array; i++)

cout << qs\_array[i] << ",";

return 0;

}

// быстрая сортировка Хоара

void quick\_sort(int \*array, int j)

{

\_time =clock();

int \_size = j; // сохранение размера массива

int pivot = array[(int)j/2]; // базовый элемент

int k =0; // итератор второй половины массивы

int temp =0; // временный буфер для обмена значений

do{

while(array[k] < pivot) k++; // пока не найдет элемент большей pivot

while(array[j] > pivot) j--; // пока не найдет элемент меньше pivot

if ( k <= j) {

temp = array[k];

array[k] = array[j];

array[j] = temp;

k++;

j--; }

} while(k <=j ); // пока не сойдут два итератора

if(j>0)

quick\_sort(array,j);

if(\_size>k)

quick\_sort(array+k, \_size-k);

\_time = clock()-\_time;

}

**//Листинг Prog\_Lab\_4\_2**

#include <iostream>

#include <ctime>

using namespace std;

unsigned int \_time;

void Shell\_sort(int \*array, int size); // прототип функции сортировки Шелла

int main()

{

setlocale(LC\_ALL, "rus"); // локализация

srand(time(0)); // автоматизация рандома

int count\_of\_array; // количество элементов в массиве

// считывание размера массива

cout << "Enter count of array ";

cin >> count\_of\_array;

int qs\_array[count\_of\_array]; // целочисленный массив

// заполнение массива случайными числами

for (int i=0; i < count\_of\_array; i++)

qs\_array[i] = 1+rand()%100;

// отображение массива в не отсортированном виде

for (int i=0; i<count\_of\_array; i++)

cout << qs\_array[i] << ",";

// вызов функции быстрой сортировки

Shell\_sort(qs\_array, count\_of\_array);

// время затраченное для сортировки массива размера N

cout <<"\n\n" <<(float)\_time/CLOCKS\_PER\_SEC << endl;

cout<<endl;

// отображение отсортированного массива

for (int i=0; i<count\_of\_array; i++)

cout << qs\_array[i] << ",";

return 0;

}

void Shell\_sort(int \*array, int size)

{

\_time = clock();

int step = size / 2;//инициализируем шаг.

while (step > 0){ //пока шаг не 0

for (int i = 0; i < (size - step); i++) {

int j = i; //будем идти начиная с i-го элемента

while (j >= 0 && array[j] > array[j + step]) //пока не пришли к началу массива

{ int temp = array[j]; //и пока рассматриваемый элемент больше

array[j] = array[j + step];//чем элемент находящийся на расстоянии шага

array[j + step] = temp;

j--;

} }

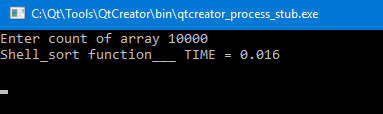
step = step / 2;} //уменьшаем шаг

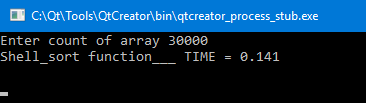
\_time = clock()-\_time;

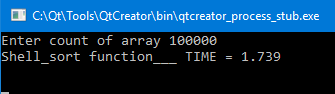
}

Сравнение двух видов сортировок

1. Сортировка Шелла

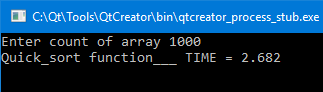
n=10000;

n=30000;

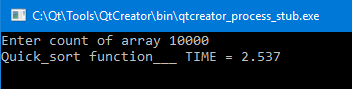
n=100000;

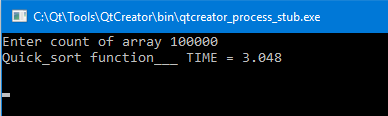
1. Сортировка Хоара

N =1000;



N=10000;



N=100000;