НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра ВТ

«Структуры и алгоритмы обработки данных»

Пояснительная записка к контрольной работе
«Сортировка коллекции»

Выполнил:
студент гр. ЗАП-933 Лутохин Г.Н.
шифр 134464479
вариант задания 2

Дата сдачи:
Отметка о зачете:

Новосибирск
2014

Цель работы: Изучение и реализация методов сортировки. Экспериментальное исследование методов сортировки.

Задание:

1. Спроектировать, реализовать и провести тестовые испытания АТД «Вектор» для коллекции, содержащей данные произвольного типа. Размер коллекции и тип данных задаются клиентской программой. АТД «Вектор» представляет собой последовательность элементов, размещенных в одной сплошной области памяти. Доступ к элементам вектора осуществляется по индексу. Интерфейс АТД «Вектор» включает следующие операции:
    - опрос размера вектора;
    - изменение размера вектора;
    - чтение/запись по индексу;
    - элементарная сортировка (по варианту задания);
    - эффективная сортировка (по варианту задания).

    Для тестирования эффективности алгоритмов сортировки интерфейс АТД «Вектор» включает следующие дополнительные операции:
    - опрос числа выполненных сравнений;
    - опрос числа выполненных обменов.

2. Выполнить отладку и тестирование отдельных операций АТД «Вектор» с помощью меню операций.

3. Выполнить сравнительное тестирование трудоемкости алгоритмов сортировки для худшего и среднего случаев.

4. Провести анализ экспериментальных показателей трудоемкости алгоритмов сортировки (обменная сортировка, рекурсивный алгоритм сортировки разделением).

Формат АТД: Шаблон коллекции данных на базе динамического массива указанного типа T. При создании массив имеет минимальный размер 10. Массив может увеличивать и уменьшать свою размерность. Операция индексирования массива выполняет проверку соответствия индекса текущему размеру массива cur_size.

Структура данных: Динамический массив элементов указанного типа T – T arr[cur_size]

Параметры:

**arr* – динамический массив объектов
cur_size – размер динамического массива
comparison_num – число сравнений в алгоритме сортировки
exchange_num – число обменов в алгоритме сортировки

Операции:

Конструктор

Вход: исходный начальный размер массива cur_size (по-умолчанию cur_size=10)
Процесс: создание динамического массива T arr[cur_size]
Постусловия: создан массив с размером cur_size

Опрос размера массива

Вход: нет
Предусловия: нет
Процесс: чтение текущего размера cur_size
Выход: текущий размер массива cur_size
Постусловия: нет

Операция индексирования

Вход: значение индекса i
Предусловия: 0 <= i < cur_size
Процесс: вычисление адреса элемента массива arr[i]

Выход: ссылка на элемент массива arr[i]
Постусловия: генерация сообщения об ошибке при невыполнении предусловия

Изменение размера массива

Вход: новый размер массива s
Предусловия: 0 <= s
Процесс: выделение памяти указанного размера s, копирование s или cur_size элементов в новый массив, удаление старого массива, изменение размера массива cur_size
Выход: True – массив изменен.
Постусловия: массив содержит s элементов старого массива, если s < cur_size; массив содержит все cur_size элементы старого массива, если s > cur_size.

Инициализация массива случайными значениями

Вход: нет
Предусловия: нет
Процесс: в массиве от 0 до cur_size элементам массива присваиваются случайные значения от 0 до RAND_MAX
Выход: нет
Постусловия: массив содержит псевдо-случайные последовательности.

Конец АТД

Определение шаблонного класса вектора и класса итератора для клиентской программы

```
template <typename T>
class Vector {
	T *arr;
	int cur_size; // current amount of objects in vector

	int comparison_num; // number of comparisons
	int exchange_num; // number of exchanges

	void _qsort(int l, int r);


	public:
	Vector(unsigned int size=10); // constructor, initial size of array = 10
	~Vector(); // destructor
	int size(); // current size of array
	T& operator[](int i);
	bool resize(int new_size); // change size of array
	void random_init(); // make random numbers in array values

	void bubble_sort();
	void quick_recursion_sort();
	void lsd_sort(const int);
	int get_comparison_num() { return comparison_num; }
	int get_exchange_num() {return exchange_num; }

	template <typename M> friend ostream& operator<< (ostream &O, const
Vector<M> &v);

	class Iterator {
		Vector<T> *ptr;
		int current_index;

		public:
		Iterator(Vector<T> *); // constructor
		T& begin(); // set iterator to first element of vector
		T& end(); // set iterator to last element
		bool next();
		bool prev();
```

```
            bool on_end_boundary();
            bool on_start_boundary();
            T& operator*(); // access to data of current element
            T& operator++(int);
            T& operator--(int);
      };
      friend class Iterator;
};
```

Тестирование трудоемкости алгоритмов обменной сортировки (сортировка пузырьком) и алгоритм поразрядной LSD-сортировки с числом разрядов в сортируемых значениях равным 32 и 4

Целью данного тестирования является снятие средних показателей трудоемкости для каждой операций сортировки (для среднего и худшего случаев), а также сравнение с теоретической трудоемкостью.

Основными показателями трудоемкости являются количество операций сравнения и обмена, выполненных за операцию сортировки коллекции. Экспериментальная трудоемкость находится как среднее число суммы операций сравнений и обменов для каждого типа сортировки. Сравнительная характеристика трудоемкостей операций производится для среднего и худшего случаев.

Описание методики тестирования

Производился последовательный вызов операций сортировки для коллекции содержащей n (интервал – от 0 + δ(0-100) до 8000 + δ(0-100)) элементов, после каждого прохода, к n прибавлялось случайное число δ(0-100).

Тестирование трудоемкости

Обменная сортировка, средний случай.

Amount	Comp.	Exch.	Total	Theory	Deviation
1000	499500	255998	755498	1000000	0,755
2000	1999000	997534	2996534	4000000	0,749
3000	4498500	2239930	6738430	9000000	0,749
4000	7998000	3992089	11990089	16000000	0,749
5000	12497500	6173235	18670735	25000000	0,747
6000	17997000	8935965	26932965	36000000	0,748
7000	24496500	12420371	36916871	49000000	0,753
8000	31996000	16358544	48354544	64000000	0,756

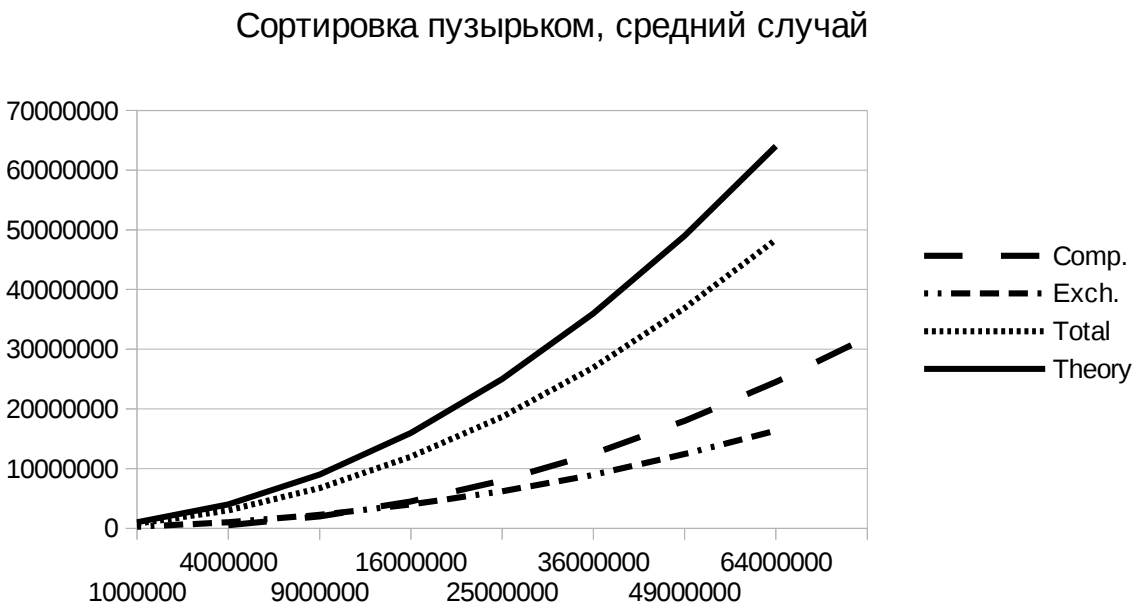

Рис. 1.

Обменная сортировка, худший случай.

Amount	Comp.	Exch.	Total	Theory	Deviation
1000	499500	499500	999000	1000000	0,999
2000	1999000	1999000	3998000	4000000	1,000
3000	4498500	4498500	8997000	9000000	1,000
4000	7998000	7998000	15996000	16000000	1,000
5000	12497500	12497500	24995000	25000000	1,000
6000	17997000	17997000	35994000	36000000	1,000
7000	24496500	24496500	48993000	49000000	1,000
8000	31996000	31996000	63992000	64000000	1,000

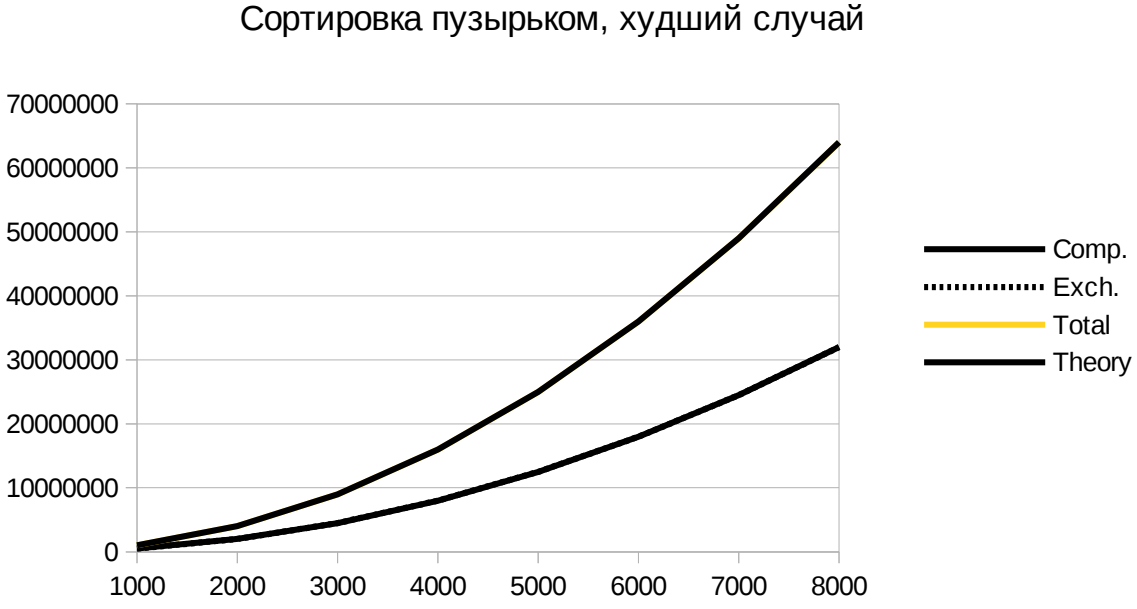

Рис. 2.

LSD-сортировка, 32 символа, средний случай.

Amount	Comp.	Exch.	Total	Theory, 32*n	Deviation
1000	64000	64000	128000	32000	4
2000	128000	128000	256000	64000	4
3000	192000	192000	384000	96000	4
4000	256000	256000	512000	128000	4
5000	320000	320000	640000	160000	4
6000	384000	384000	768000	192000	4
7000	448000	448000	896000	224000	4
8000	512000	512000	1024000	256000	4

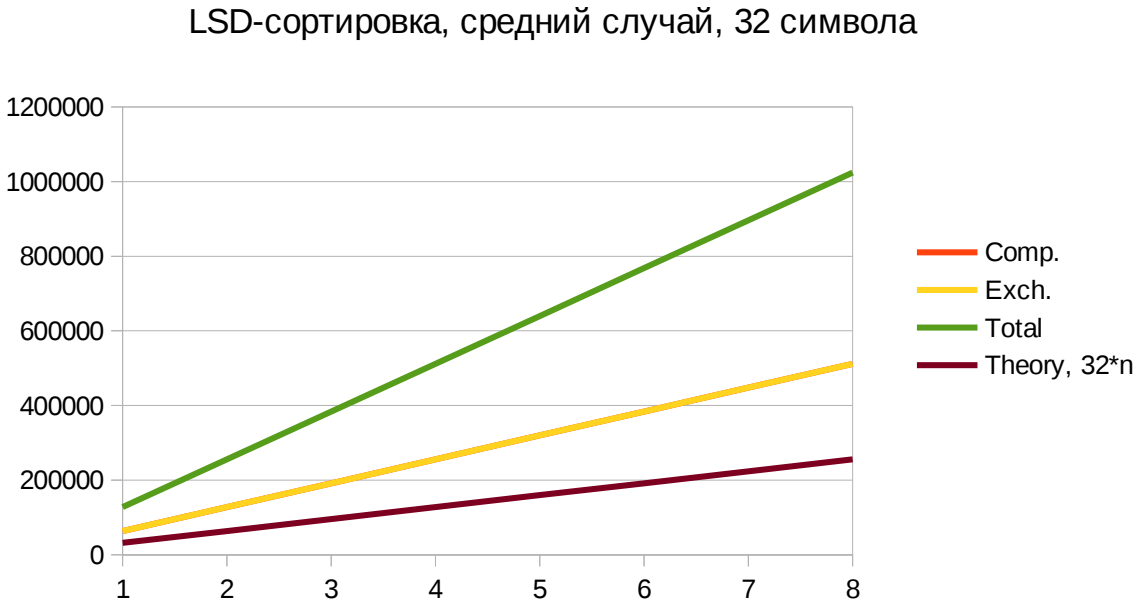

Рис. 3.

LSD-сортировка, худший случай, 32 символа.

Amount	Comp.	Exch.	Total	Theory, 32*n	Deviation
1000	64000	64000	128000	32000	4
2000	128000	128000	256000	64000	4
3000	192000	192000	384000	96000	4
4000	256000	256000	512000	128000	4
5000	320000	320000	640000	160000	4
6000	384000	384000	768000	192000	4
7000	448000	448000	896000	224000	4
8000	512000	512000	1024000	256000	4

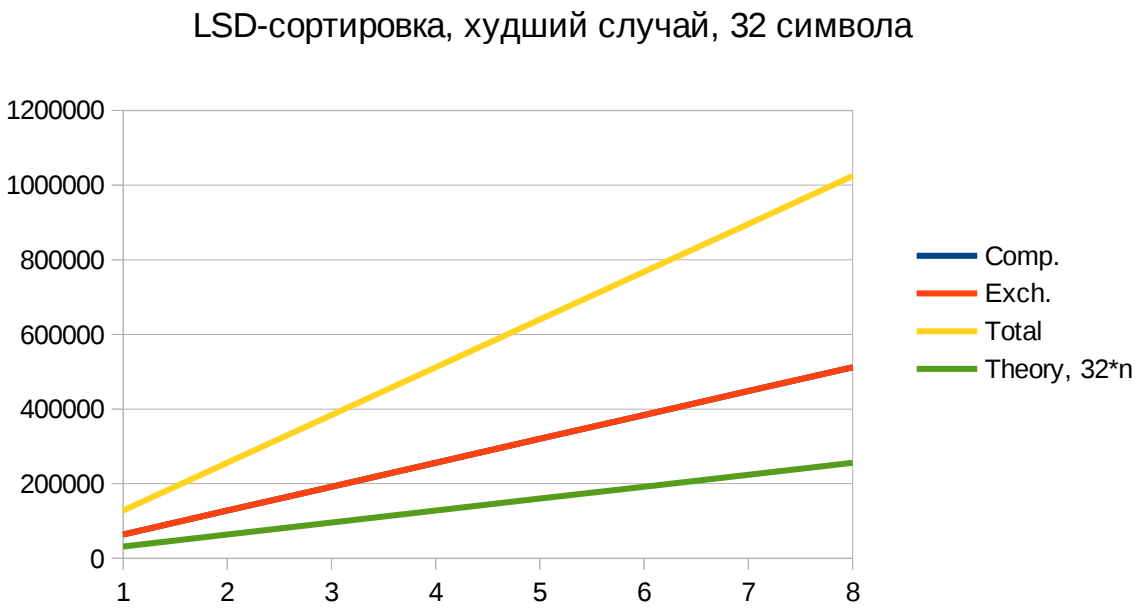

Рис. 4.

LSD-сортировка, 4 символа, средний случай.

Amount	Comp.	Exch.	Total	Theory, 4*n
1000	8000	8000	16000	4000
2000	16000	16000	32000	8000
3000	24000	24000	48000	12000
4000	32000	32000	64000	16000
5000	40000	40000	80000	20000
6000	48000	48000	96000	24000
7000	56000	56000	112000	28000
8000	64000	64000	128000	32000

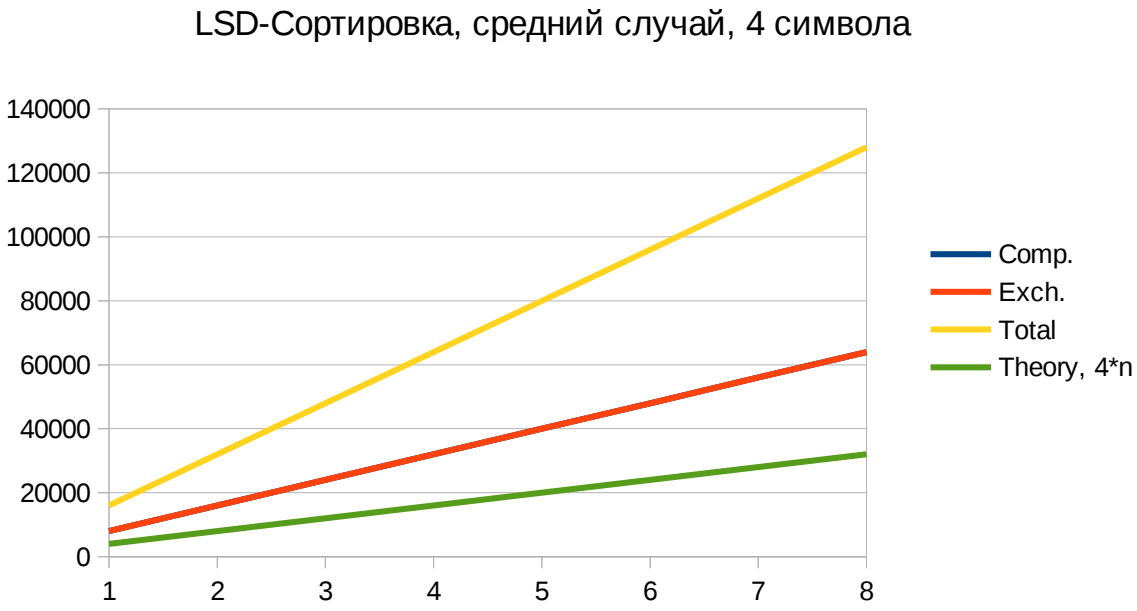

Рис. 5

LSD-сортировка, худший случай, 4 символа.

Amount	Comp.	Exch.	Total	Theory, 4*n
1000	8000	8000	16000	4000
2000	16000	16000	32000	8000
3000	24000	24000	48000	12000
4000	32000	32000	64000	16000
5000	40000	40000	80000	20000
6000	48000	48000	96000	24000
7000	56000	56000	112000	28000
8000	64000	64000	128000	32000

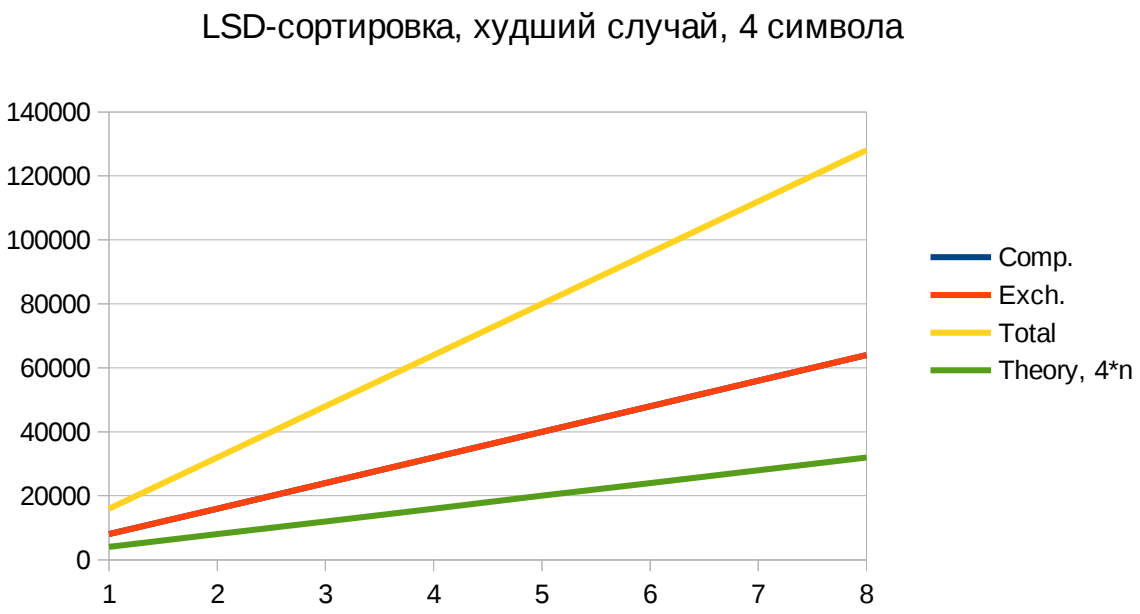

Рис. 6.

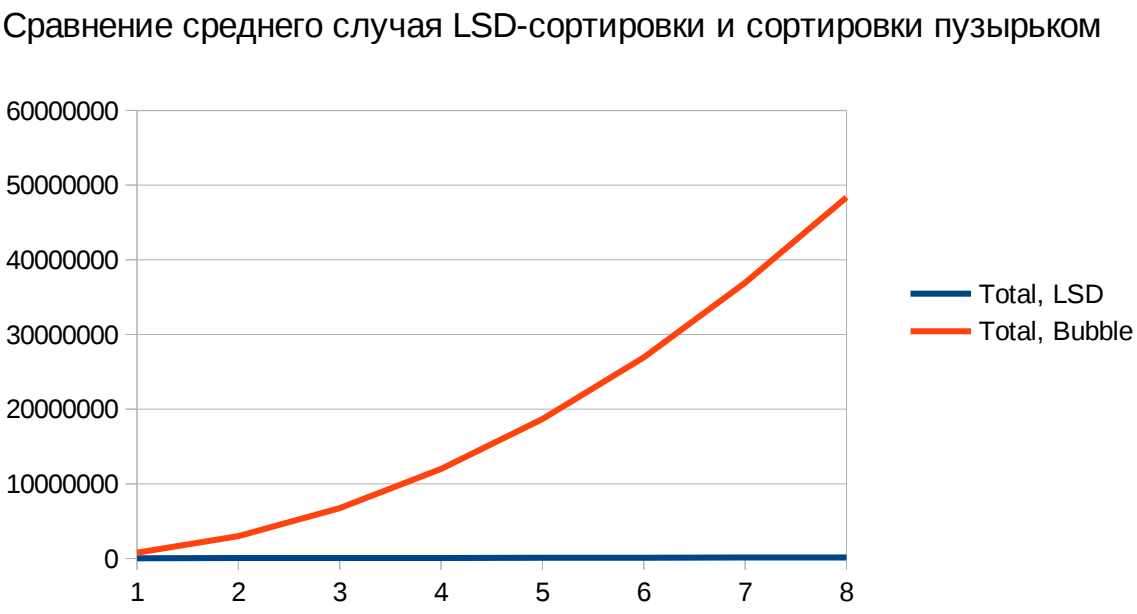

Рис. 7

Рис. 8.

Полученные экспериментальные оценки полностью соответствуют теоретическим с учётом небольших отклонений.

Вывод: LSD-сортировка во много раз эффективней обычной сортировки.

Листинг тестирования

```
#include <iostream>
#include <cstring>
#include "myvector.h"

using namespace std;

#define TAB "\t"

void middle_case_init(Vector<int> &v) {
	v.random_init();
}

void worst_case_init(Vector<int> &v) {

	Vector<int>::Iterator iter(&v);
	iter.begin();
	for(int i=(v.size() - 1); i >=0; i--){
		*iter = i;
		iter.next();
	}
}

void (*vector_init)(Vector<int> &v);

int main(int argc, char **argv){

	if (argc >= 2 && strcmp("middle", argv[1]) == 0) {
		vector_init = &middle_case_init;

	} else if (argc >= 2 && strcmp("worst", argv[1]) == 0) {
		vector_init = &worst_case_init;

	} else {
		cerr << "error: bad argument" << endl;
		cerr << "help: " << argv[0] << " middle|worst" << endl;
		return 0;
	}

	Vector<int> v;
	Vector<int>::Iterator iter(&v);
	int c, e, //comparisons and exchanges
	    da; // amount increment

	cout << "Amount" TAB "Comp." TAB "Exch." TAB "Total" TAB "Theory" <<
endl;
	for(int amount=1000; amount <= 8000; amount+=1000){

		v.resize(amount);

		(*vector_init)(v);

		v.bubble_sort();

		c = v.get_comparison_num();
		e = v.get_exchange_num();

		cout << v.size() << TAB << c << TAB << e << TAB << c+e << TAB <<
v.size() * v.size() << endl;
	}

	return 0;
}
```

Листинг класса АТД «Вектор»

```
#ifndef MYVECTOR_H
#define MYVECTOR_H

#ifndef IAMREMEMBER
	#warning "Don't remember to include `srand((unsigned) time(0))` in
`main` function"
#endif

#include <iostream>
using namespace std;

#include <ctime>
#include <cstdlib>
#include <cmath>

template <typename T>
class Vector {
	T *arr;
	int cur_size; // current amount of objects in vector

	int comparison_num; // number of comparisons
	int exchange_num; // number of exchanges

	void _qsort(int l, int r);


	public:
	Vector(unsigned int size=10); // constructor, initial size of array = 10
	~Vector(); // destructor
	int size(); // current size of array
	T& operator[](int i);
	bool resize(int new_size); // change size of array
	void random_init(); // make random numbers in array values

	void bubble_sort();
	void lsd_sort(const int);
	void quick_recursion_sort();
	int get_comparison_num() { return comparison_num; }
	int get_exchange_num() {return exchange_num; }

	template <typename M> friend ostream& operator<< (ostream &O, const
Vector<M> &v);

	class Iterator {
		Vector<T> *ptr;
		int current_index;

		public:
		Iterator(Vector<T> *); // constructor
		T& begin(); // set iterator to first element of vector
		T& end(); // set iterator to last element
		bool next();
		bool prev();
		bool on_end_boundary();
		bool on_start_boundary();
		T& operator*(); // access to data of current element
		T& operator++(int);
		T& operator--(int);
	};
	friend class Iterator;
};
```

```
// Iterator functions
template <typename T> Vector<T>::Iterator::Iterator(Vector<T> *v) {
      ptr = v;
      current_index = 0;
}

template <typename T> bool Vector<T>::Iterator::on_start_boundary() {
      return (current_index == 0);
}

template <typename T> bool Vector<T>::Iterator::on_end_boundary() {
      return (current_index == (ptr->cur_size - 1));
}

template <typename T> T& Vector<T>::Iterator::begin(){
      current_index = 0;

      return ptr->arr[current_index];
}

template <typename T> T& Vector<T>::Iterator::end(){
      current_index = ptr->cur_size - 1;

      return ptr->arr[current_index];
}

template <typename T> bool Vector<T>::Iterator::next() {
      if (current_index != (ptr->cur_size - 1)) {
            current_index++;
            return true;
      } else {
            return false;
      }
}

template <typename T> bool Vector<T>::Iterator::prev(){
      if (current_index != 0) {
            current_index--;
            return true;
      } else {
            return false;
      }
}

template <typename T> T& Vector<T>::Iterator::operator*(){
      return ptr->arr[current_index];
}

template <typename T> T& Vector<T>::Iterator::operator++(int i) {
      this->next();
      return **this;
}

template <typename T> T& Vector<T>::Iterator::operator--(int i) {
      this->prev();
      return **this;
}

// Vector functions
template <typename T> Vector<T>::Vector(unsigned int size){

      arr = new T[size];
      cur_size = size;
```

```
	comparison_num = exchange_num = 0;
}

template <typename T> Vector<T>::~Vector(){
	delete [] arr;
}

template <typename T> int Vector<T>::size(){
	return this->cur_size;
}

template <typename T> T& Vector<T>::operator[](int i){
	if (i < 0 || i >= cur_size) {
		throw "Bad array index";
	}

	return arr[i];
}

template <typename T> bool Vector<T>::resize(int new_size) {
	if (new_size <= 0) {
		throw "Wrong parameter of vector function `resize`";
	}

	T *tmp;
	tmp = new T[new_size];

	int min_size = cur_size < new_size ? cur_size : new_size;

	for(int i = 0; i < min_size; i++) {
		tmp[i] = arr[i];
	}

	delete [] arr;
	arr = tmp;
	cur_size = new_size;

	return true;
}

template <typename T> void Vector<T>::random_init(){

	for(int i = 0; i < cur_size; i++){
		arr[i] = rand();
	}
}

template <typename T> void Vector<T>::bubble_sort(){

	T temp;

	comparison_num = exchange_num = 0;

	for(int i=0; i < this->cur_size - 1; i++) {
		for(int j = this->cur_size - 1; i < j; j--) {
			comparison_num++;
			if (arr[j-1] > arr[j]){
				temp = arr[j];
				arr[j] = arr[j-1];
				arr[j-1] = temp; exchange_num++;
			}
		}
	}
}
```

```
template <typename T> void Vector<T>::quick_recursion_sort(){
	comparison_num = exchange_num = 0;
	_qsort(0, this->size() - 1);
}

template <typename T> void Vector<T>::lsd_sort(const int R =16) {
	comparison_num = exchange_num = 0;

	// input param test
	double _ = log2(R);

	if ( (_ - (int)_ ) > 0) {
		throw "Radix is not a power of 2!!!";
	}

	// vars
	int d; // number of current digit
	T *C = new T[this->cur_size]; // helper array for buckets

	int k = sizeof(T) * 8 / log2(R); // maximum amount of digits in element

	// helper, extraction of digit no. d
	class H {
		public:
		static int digit(const T v, int d, int R) {
			d = -(d - (sizeof(T) * 8 / log2(R) - 1));
			return (v >> ((int)log2(R) * d)) % R;
		}
	};

	for(d = k - 1; d >= 0; d--) {

		T count[R]; // counter of each bucket size
		for(int i = 0; i < R; i++) {
			count[i] = 0;
		}

		int j = 0;
		for(int i = 0; i < this->cur_size; i++) {
			j = H::digit(arr[i], d, R);
			comparison_num++;
			count[j+1]++;
		}

		for(int r = 1; r < R; r++)
			count[r]+= count[r-1];

		j = 0;
		for(int i = 0; i < this->cur_size; i++) {
			j = H::digit(this->arr[i], d, R);
			comparison_num++;
			C[count[j]++] = this->arr[i];
			exchange_num++;
		}

		for (int i = 0; i < this->cur_size; i++) {
			this->arr[i] = C[i];
			exchange_num++;
		}
	}

	delete [] C;
}
```

```
template <typename T> void Vector<T>::_qsort(int left, int right){

	int i, j;
	T x;

	i = left; j = right;
	x = arr[(left+right)/2];

	do {
		while((arr[i] < x) && (i < right)) {i++; comparison_num++;}
comparison_num++;
		while((x < arr[j]) && (j > left)) {j--; comparison_num++;}
comparison_num++;

		if(i <= j) {
			T tmp;
			tmp = arr[i];
			arr[i] = arr[j];
			arr[j] = tmp; exchange_num++;
			i++; j--;
		}
	} while(i <= j);

	if(left < j) _qsort(left, j);
	if(i < right) _qsort(i, right);

}

template <typename M> ostream& operator<< (ostream &o, const Vector<M> &v) {
	for (int i = 0 ; i < v.cur_size ; i++) {
				o << v.arr[i] << " ";
	}

	o << endl;

	return o;
}
#endif
```