Министерство образования Российской Федерации

Новосибирский государственный технический университет

Кафедра вычислительной техники

Курсовой проект

по дисциплине «Программирование»

«Иерархические структуры данных в памяти»

Факультет: АВТ

Группа: АП-818

Выполнила: Игнатова А.

Преподаватель: Романов Е.Л.

Новосибирск, 2010

Задание

Общая формулировка задачи

На логическим уровне разрабатываемая структура данных представляет собой обычную линейную последовательность элементов (строк) со стандартным набором операций (добавление в конец, вставка и удаление по логическому номеру, сортировка, бинарный поиск, вставка с сохранением порядка, сохранение и загрузка из текстового файла, выравнивание (балансировка - выравнивание размерностей структур данных нижнего уровня). Физическая структура данных имеет два уровня. На нижнем уровне поддерживается ограничение размерности структуры данных: при переполнении она разбивается пополам, соответствующие изменения вносятся в верхний уровень. При выполнении работы произвести измерение зависимости «грязного» времени работы программы и ее трудоемкости (количества базовых операций). Оценить вид полученной зависимости (линейно-логарифмическая, квадратичная).

Выбранная тема

7. Шаблон структуры данных - массив указателей на заголовки списков. Элемент списка содержит указатель на объект. (При включении элемента последним в список предусмотреть ограничение длины текущего списка и переход к следующему).

Комментарии

1. Размерность списков будем хранить в ДМУ.
2. Список сделаем односвязным, где первый элемент является заголовком.

1. Структурное описание разработки

В программе используются 4 класса:

1. `class Meduza`-главный класс программы, массив указателей на заголовки списков. Элемент списка содержит указатель на строку
2. `class stringWrapper`- для удобства работы с указателями на строку
3. `class list`-элемент списка с набором функций работы со списком
4. `class vector`-ДМУ

Схема структуры данных в программе

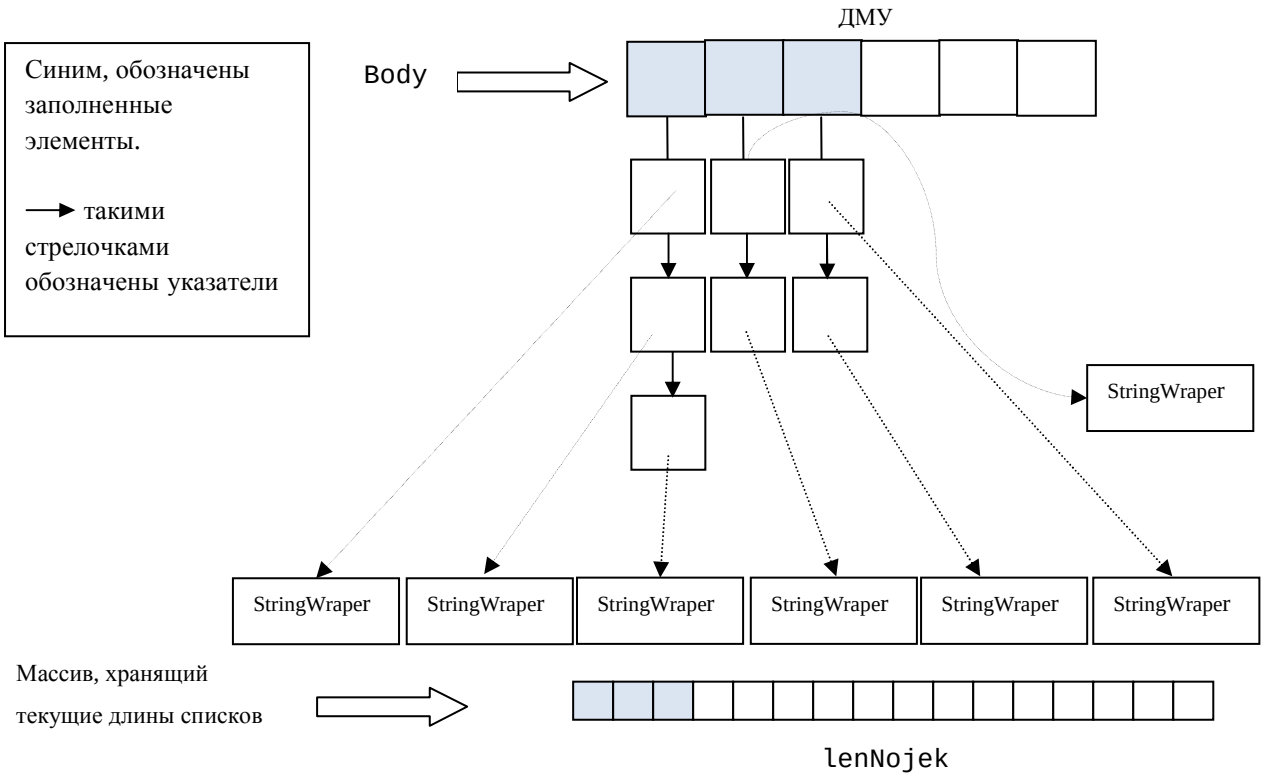

Рисунок №1. Структурная схема.

Немного о структуре данных

Иерархия структуры данных позволяет при правильном соотношении параметров и типа управляющей структуры и структуры нижнего уровня (структуры с данными) обойти недостатки простых структур, например, массива или списка.
По сути своей данная структура данных представляет собой список с индексом: за счет иерархии СД увеличивается быстродействие списка, теперь для получения элемента по номеру понадобится меньше действий.
Итак, массив указателей на заголовки списков. Для реализации этой структуры данных нужно рассмотреть две сущности:

1. динамический массив указателей
2. односвязный список

Массив указателей

`class vector` – шаблонный, т.к. по заданию, во-первых, нам нужно хранить там указатели на заголовки списков. А во-вторых, нам нужно хранить еще в одном массиве длины текущих списков, для исключения их последовательного просмотра.

Список

В операциях по логическому номеру, чтобы не путаться в дальнейшем, я начинаю считать с 0.

С логическим номером связаны следующие методы:

1. Получение элемента по номеру.

Который просто возвращает строку в этом элементе.

2. Вставка по логическому номеру.

Сдвигает элементы в массиве, освобождая нужное место, и затем вставляет на это место полученный из входного параметра указатель.

3. Удаление по логическому номеру.

Который убирает указатель на этот элемент из массива, и затем его же и возвращает.

Поиск нужного элемента в списке

Допустим, нужно найти две координаты: номер элемента по списку и номер самого списка.

Для этого у нас есть ДМУ, где хранятся длины списков, пройдясь по нему нехитрыми вычислениями можно найти эти координаты:

```
void getPosition(int pos,int &nNojki,int &nVNojke) // параметры: логический номер
//элемента, номер элемента по списку, номер списка
	{
		int sum=0;
		int n = lenNojek.getSize(); //массив длин списков
		int i=0;
		for (;i<n;i++)
		{
			int curLen = *lenNojek[i];
			if ((sum+curLen)>pos) break;
			sum+=curLen;
		}
		nNojki = i;
		nVNojke = pos-sum;
	}
```

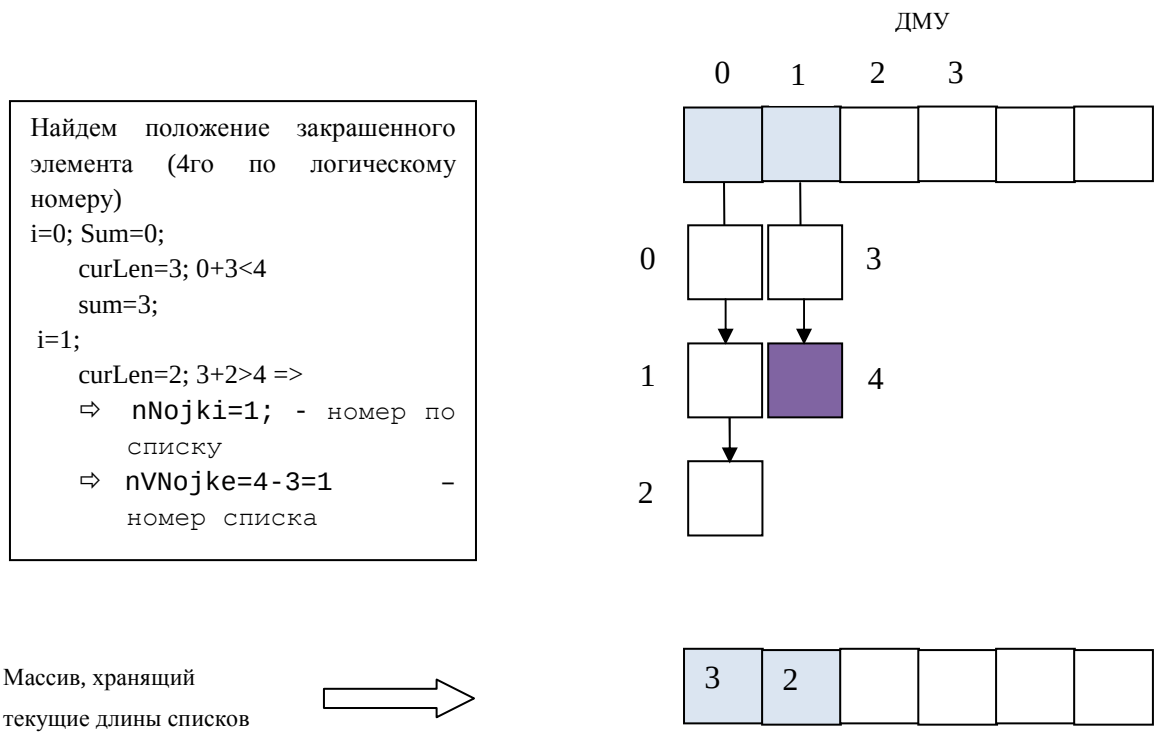

Рисунок № 2. Пример нахождения позиции элемента.

Бинарный поиск

Стандартный алгоритм поиска элемента в отсортированном массиве (векторе), также известен как метод деления пополам или дихотомия. Здесь нам уже ничего не нужно придумывать и усложнять. Данный метод:

- Возвращает индекс, куда может быть вставлен элемент.
- Если искомое значение встречается несколько раз, то вернет индекс самого левого элемента из подходящих.
- Корректно обрабатывает значения, выходящие за пределы массива.
- Корректно работает с массивами очень большого размера, т.е. переполнение при вычислении индекса среднего элемента не происходит.

Балансировка

Одна из самых важных возможностей Верховного Иерарха – это возможность сохранять баланс СД. Иными словами, выравнивать размерность структуры данных второго уровня.
После каждой операции добавления список может оказаться длиннее, чем нужно, поэтому обязательно должна быть проверка на переполнение. Т.е нужна балансировка структуры.
При переполнении списка он разбивается пополам и вносятся изменения на верхний уровень, где создается новый список.

Балансировка:

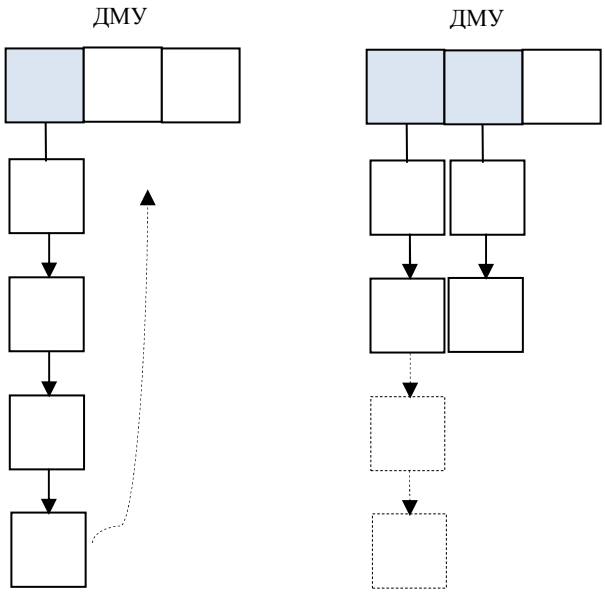

Рисунок №3. Балансировка.

Сортировка

За основу возьмем сортировку вставками.

Происходит следующим образом: мы перебираем «ножки медузы» и совершаем обход списка. Пока список не закончится, вставляем его элементы в новый список с **сохранением порядка,** т.е. если элемент больше максимального, то он вставляется за ним. Далее, обходя уже новый отсортированный список, возвращаем «ножки» на прежнее место.

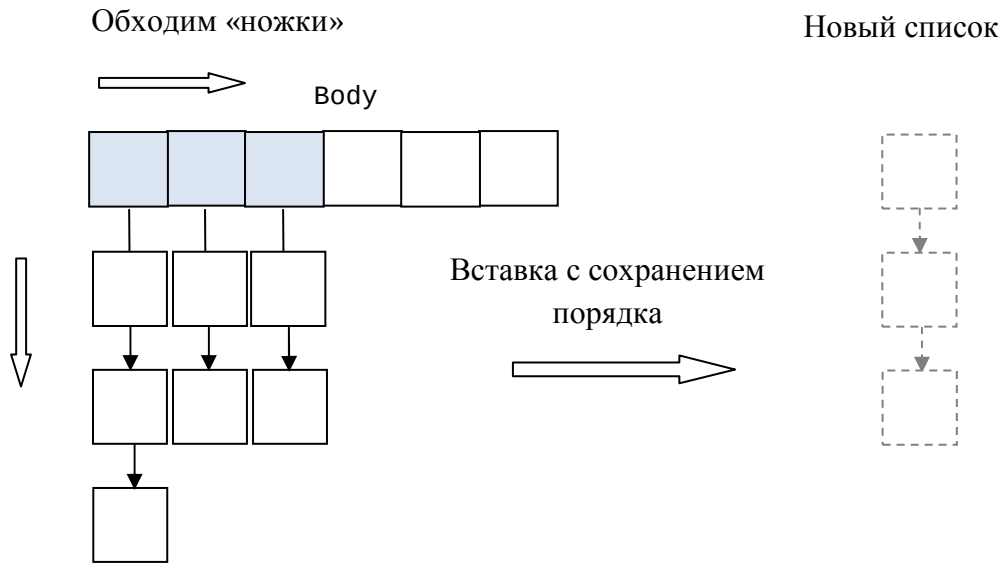

Рисунок № 4. Сортировка.

2. Функциональное описание

Рассмотрим методы всех классов, используемых программой.

Класс vector.

template<class T> // динамический массив указателей
class vector
{
private:
    T **arr; //массив указателей
    int size, n; //число элементов
};

vector(vector& R); // конструктор копирования

~vector(void); // деструктор

vector(void); //конструктор по умолчанию
создает массив и заполняет его нулями

void swap(int i, int j); // обмен

int getSize(); //получение размерности массива

void extend(int new_size); // перераспределение памяти

T* max(); // возвращает указатель на max элемент

T* min(); // возвращает указатель на min элемент

void sort(); // сортировка

void insert(int pos, T *ptr); // вставка по логическому номеру

void insSort(T *ptr); // вставка с сохранением порядка

void add(T *ptr); // вставка в конец

T* remove(int pos); // удаление по логическому номеру
Возвращает указатель на удаленный элемент

T* operator[](int i); // переопределение операции []
Возвращает указатель на объект из массива указателей по индексу i

friend ostream &operator <<(ostream& IO,vector &t)
friend istream &operator >>(istream& IO,vector &t)

переопределение операций ввода/вывода в поток

Класс stringWrapper

class stringWrapper // класс для удобства работы с указателями на строку
{

```
        char *str;
…
```

public:

char * get(); // возвращает указатель на строку

stringWrapper(); // конструктор по умолчанию

stringWrapper(stringWrapper &R); // конструктор копирования

~stringWrapper(); // деструктор

friend ostream &operator<<(ostream& IO, stringWrapper &t);
friend istream &operator>>(istream &IO, stringWrapper &t);
переопределение операций ввода/вывода в поток

operator int(); // переопределение int()
возвращает длину строки

stringWrapper &operator=(stringWrapper &r);
Загружает в текущий объект, объект r

int operator==(stringWrapper &t) { return strcmp(str,t.str)==0; }
int operator!=(stringWrapper &t) { return strcmp(str,t.str)!=0; }
int operator< (stringWrapper &t) { return strcmp(str,t.str)< 0; }
int operator<=(stringWrapper &t) { return strcmp(str,t.str)<=0; }
int operator> (stringWrapper &t) { return strcmp(str,t.str)> 0; }
int operator>=(stringWrapper &t) { return strcmp(str,t.str)>=0; }
переопределение операций ставнения

Класс list.

```
template <class T>
class list{
        list<T> *next;                          // указатель на следующий элемент
        T data;                                 // хранит сам объект
        list(T& v){data=v; next=NULL;} //конструктор для элементов с данными
```

public:

list() ; // конструктор для заголовка
~list() ; // деструктор
list(list &R); // конструктор копирования

void end(T &v); // вставка в конец списка

list* tail(int n); //отрезает «кусок хвоста» и возвращает отрезанную часть

list* getNext(list* l); // возвращает следующий элемент

void insert(T& d,int n); // включение по логическому номеру

void set(T &d); // устанавливает данные

T get(); // получение элемента

T get(int n); // получение элемента по номеру

void inssort(T &d); // включение с сохранением порядка

void after(T &d); // вставка элемента после текущего

T remove(int n); // удаление по логическому номеру

operator int(); // приведение к int()
возвращает длину списка

friend ostream &operator <<(ostream& IO,list <T> &t);
friend istream &operator >>(istream& IO,list <T> &t);
переопределение операций ввода/вывода в поток

void sort(); // сортировка «вставками»

Класс Meduza

class Meduza // основной класс программы
{

vector<list<stringWrapper>> body; // Массив указателей на заголовки списков. Элемент списка содержит указатель на строку.

vector<int> lenNojek; // массив, в котором хранятся длины списков

int maxNojkaLen; // максимальная длина списка

…

void getPosition(int pos,int &nNojki,int &nVNojke); // поиск нужного элемента
возращает координаты элемента.

Meduza(void) // конструктор по умолчанию
Задает максимальную длину списка = 10

Meduza(int nojkaLen); // конструктор

~Meduza(void); // деструктор

void growing(int n=-1); // метод балансировки

void end(char* str); // включение в конец

int length(); // количество элементов в медузе

char* get(int pos); // получение элемента по номеру

int insert(int pos,char *str); // вставка по логическому номеру

char* remove(int pos); // удаление по логическому номеру

void sort(); // сортировка

int binSearch(char *str); // бинарный поиск

void inssort(char *str); // вставка с сохранением порядка

friend ostream &operator <<(ostream& IO,Meduza &t); // сохранение в текстовый файл

friend istream &operator >>(istream& IO,Meduza &t); // загрузка из текстового файла

void friendlyShow(); // «красивый» вывод структуры на экран

3. Описание работы программы

Итак, приступим к тестированию программы. Для этого необходимо установить зависимость времени ее работы от объема входных данных.

Измерять время работы программы будем при помощи имеющихся в библиотеках функций. Используя функцию текущего системного времени (**clock**), мы получим «грязное» время с учетом всех других работ, выполняющихся в операционной системе. Кроме того, это время зависит от аппаратной и программной конфигурации, на которой производится измерение.

Для начала проверим зависимость времени сортировки от размерности максимальной длины списков (параметра `maxNojkaLen`).

Алгоритм сортировки:

1. Создаем новый список
2. Обходим все «ножки», и вставляем элементы в новый список с сохранением порядка.
3. Возвращаем отсортированный список на прежнее место и удаляем новый.

Отдадим Meduze , к примеру, 10 000 строк, сгенерированных рандомно. Вот что у нас получается:

Таблица 1. Зависимость максимальной длины списка от времени сортировки

MaxLen	35	45	55	65	75	85	95	200	300	400	500	600	700	800	900	1000
t(мс)	3576	2366	2086	2037	1992	2209	2168	2054	1972	2071	2012	1994	1921	1961	1952	1905

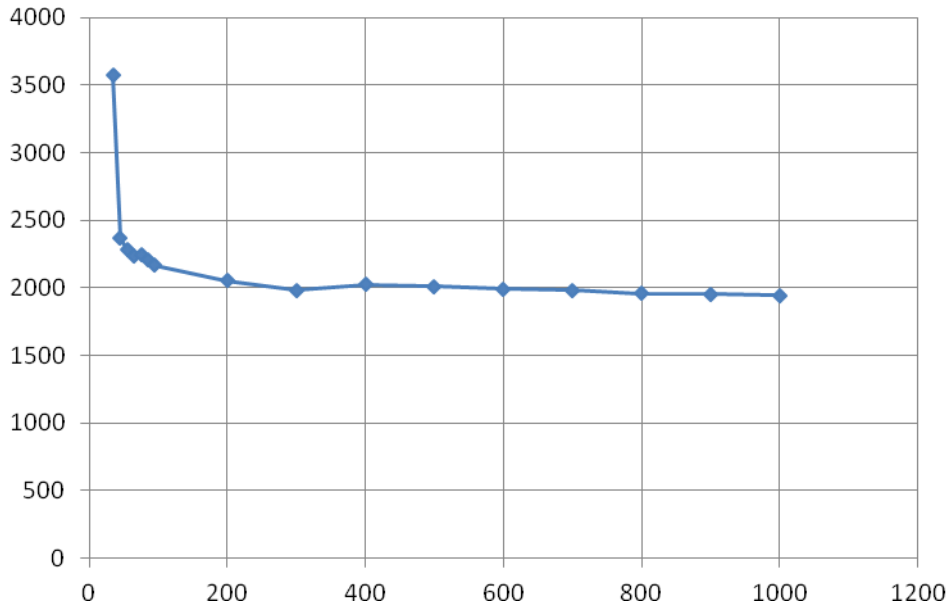

Рисунок № 5. Зависимость максимальной длины списка от времени сортировки.

Значения размерности с 10000 позволяют поместить все строки в один список. Но при этом весьма не эффективны, т.к. убивают преимущество иерархической СД. На рисунке №5 представлен график зависимости на участке до 1000 элементов массива. Очевидно, что значения меньше ста не оптимальны, а больше тысячи будут затратны.

Теперь можно проследить, как влияет размерность входных данных (количество строк) на количество операций сравнения в программе.
Для получения трудоемкости в программу включаем переменные-счетчики, изменяющие свое значение в той точке программы, где присутствует соответствующая операция, в данном случае – операция сравнения.

Таблица 2. Зависимость количества операций сравнения от размерности входных данных.

время работы (мс)	0	0	1	4	2	4	6	8	18	75
количество строк	10	50	100	200	300	400	500	600	1000	2000
количество операций сравнения	33	681	2531	10086	22634	42245	60734	89372	244352	1010327

Проверим, насколько экспериментальные данные **F1(N)** соответствуют виду степенной функции $\mathbf{F(N)=K*N^P}$

Таблица №3.

N	cmp(эскп)	cmp*(матем)	отклонение	коэф по N1	Степень
100	2531	2531	0%	0,253100	2
200	10086	10124	0%		
300	22634	22779	-1%		
400	42245	40496	4%		
500	60734	63275	-4%		
600	89372	91116	-2%		
1000	244352	253100	-3%		
2000	1010327	1012400	0%		

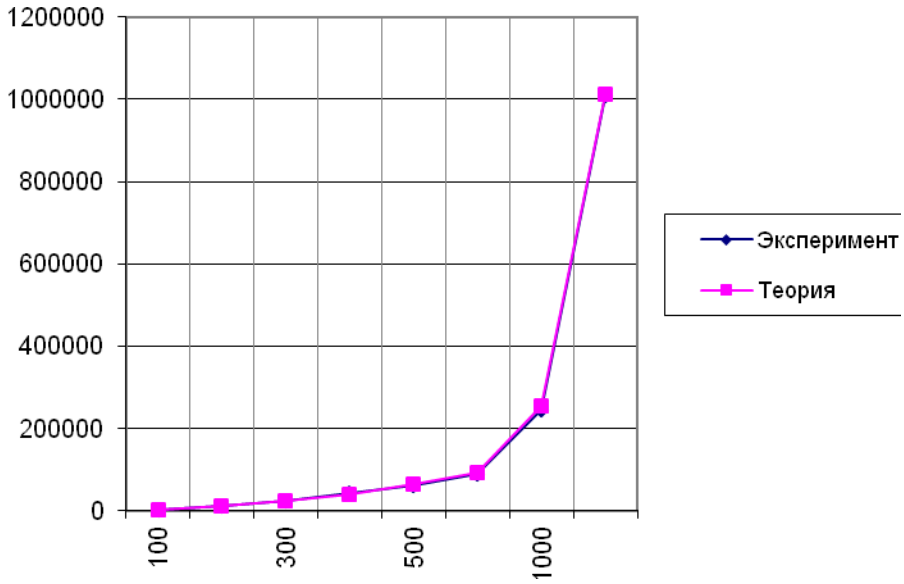

Рисунок №6. Зависимость количества операций сравнения от размерности входных данных.

Таблица №4.

N	T1(эскп)	T1*(матем)	отклонение	коэф по N1	Степень
100	1	1,0	0%	0,0015848932	1,40
200	4	2,6	52%		
300	2	4,7	-57%		
400	4	7,0	-43%		
500	6	9,5	-37%		
600	8	12,3	-35%		
1000	18	25,1	-28%		
2000	75	66,3	13%		

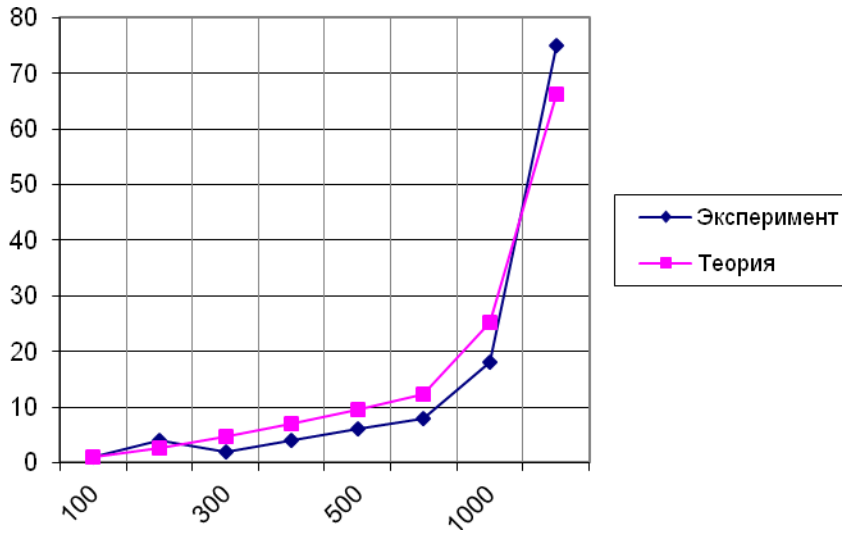

Рисунок № 7. Зависимость времени работы от размерности входных данных.

Таким образом, можно сделать некоторые выводы:

1. Количество операций сравнения практически точно имеет зависимость $\mathbf{T(N)=N^2/3}$ (K=0.33 N=2), что соответствует нашим предварительным теоретическим оценкам;
2. «Грязное» время работы программы подчиняется степенной зависимости с показателем P=2.4, т.е. примерно $\mathbf{N^2\sqrt{N}}$, т.е. хуже, чем квадратичная.

4. Приложение. Исходный текст программы.

```
#pragma once
#include <iostream>
using namespace std;

#include "stringWrapper.h"
#include "vector.h"
#include "list.h"

class Meduza
{
	vector<list<stringWrapper> > body;
	vector<int> lenNojek;
	int maxNojkaLen; //ограничение длины текущего списка

	void getPosition(int pos,int &nNojki,int &nVNojke)
	{
		int sum=0;

		int n = lenNojek.getSize();
		int i=0;
		for (;i<n;i++)
		{
			int curLen = *lenNojek[i];
			if ((sum+curLen)>pos) break;
			sum+=curLen;
		}

		nNojki = i;
		nVNojke = pos-sum;
	}

public:
	Meduza(void)
	{
		maxNojkaLen = 10;
	}

	Meduza(Meduza& R)
	{
		maxNojkaLen = R.maxNojkaLen;
		int n = R.lenNojek.getSize();
		for (int i=0;i<n;i++)
		{
			lenNojek.add(new int(*R.lenNojek[i]));
			body.add(new list<stringWrapper>(*R.body[i]));
		}

	}

	Meduza(int nojkaLen)
	{
		maxNojkaLen = nojkaLen;
	}

	~Meduza(void)
	{
		//так как вектор хранит указатели
		int n = lenNojek.getSize();
		for (int i=0;i<n;i++)
		{
			delete lenNojek[i]……..
```