**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**ОТЧЕТ**

**ПО КУРСОВОЙ РАБОТЕ**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «ИЗМЕРЕНИЕ ВРЕМЕННОЙ СЛОЖНОСТИ АЛГОРИТМА В ЭКСПЕРИМЕНТЕ НА ЭВМ»**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 3311 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Сапронов К. Д.  Землякова С. А. |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Преподаватель |  | Манирагена Валенс |

Санкт-Петербург

2025

**Задание:**  
На основе программы, составленной по гл. 3, выполнить статистический эксперимент по измерению фактической временной сложности алгоритма обработки данных.

Программа дорабатывается таким образом, чтобы она генерировала множества мощностью, меняющейся, например, от 10 до 200, измеряла время выполнения цепочки операций над множествами и последовательностями и выводила результат в текстовый файл. Каждая строка этого файла должна содержать пару значений «размер входа — время» для каждого опыта. Затем эти данные обрабатываются, и по результатам обработки делается заключение о временной сложности алгоритма.

**Описание эксперимента**

Для проведения эксперимента программа из лабораторной работы №3 была модифицирована следующим образом:

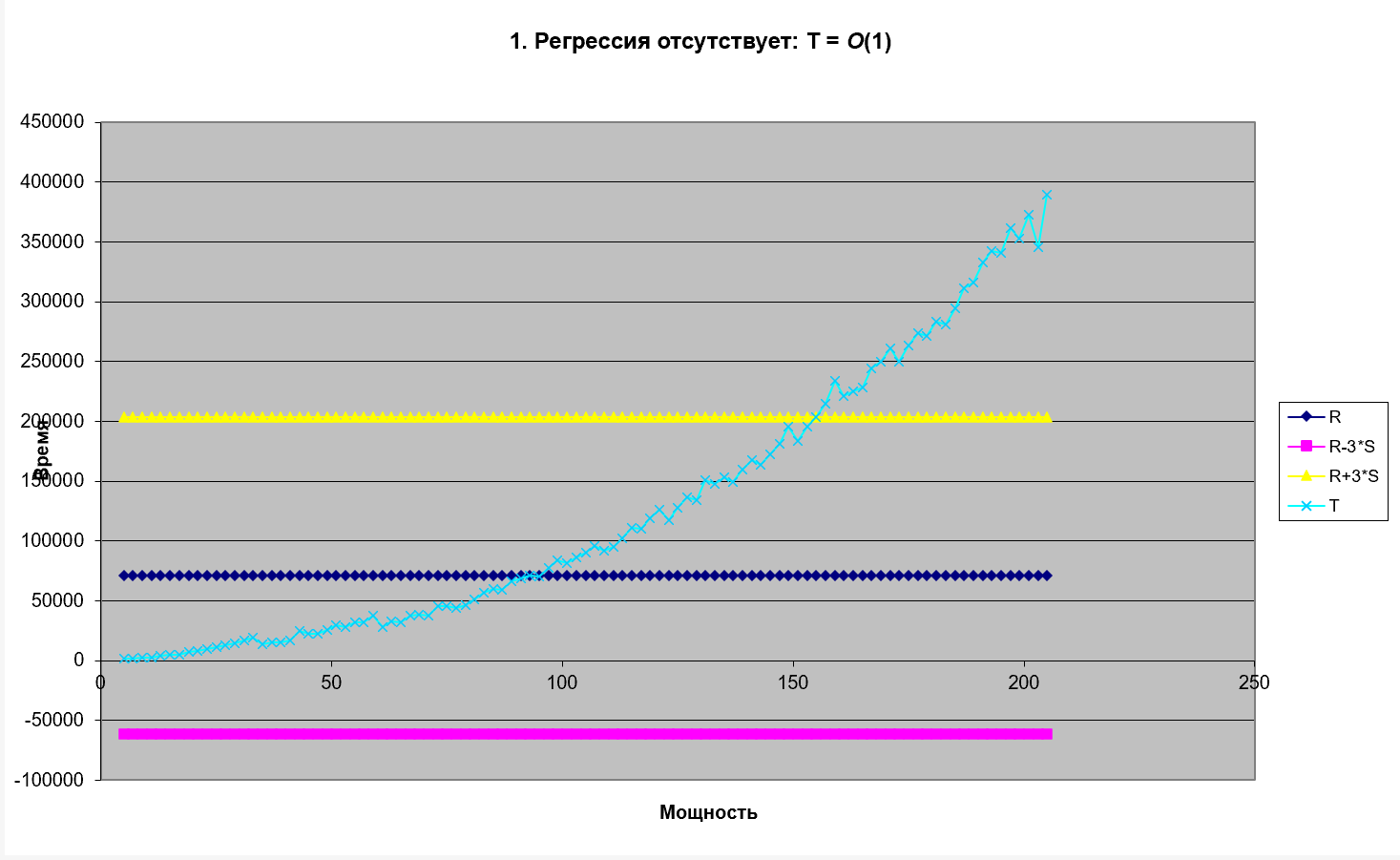
* Все операции с множествами и последовательностями выполнялись в цикле, изменяющим мощности множеств от 6 до 300 элементов с шагом 1;
* Каждую итерацию производился замер времени выполнения операций с помощью библиотеки chrono;
* Для повышения достоверности эксперимента каждую итерацию определялась мощность множества;
* Для повышения достоверности время для каждой мощности множества считается дважды;
* Теоретическая временная сложность определяется по «худшей» операции.

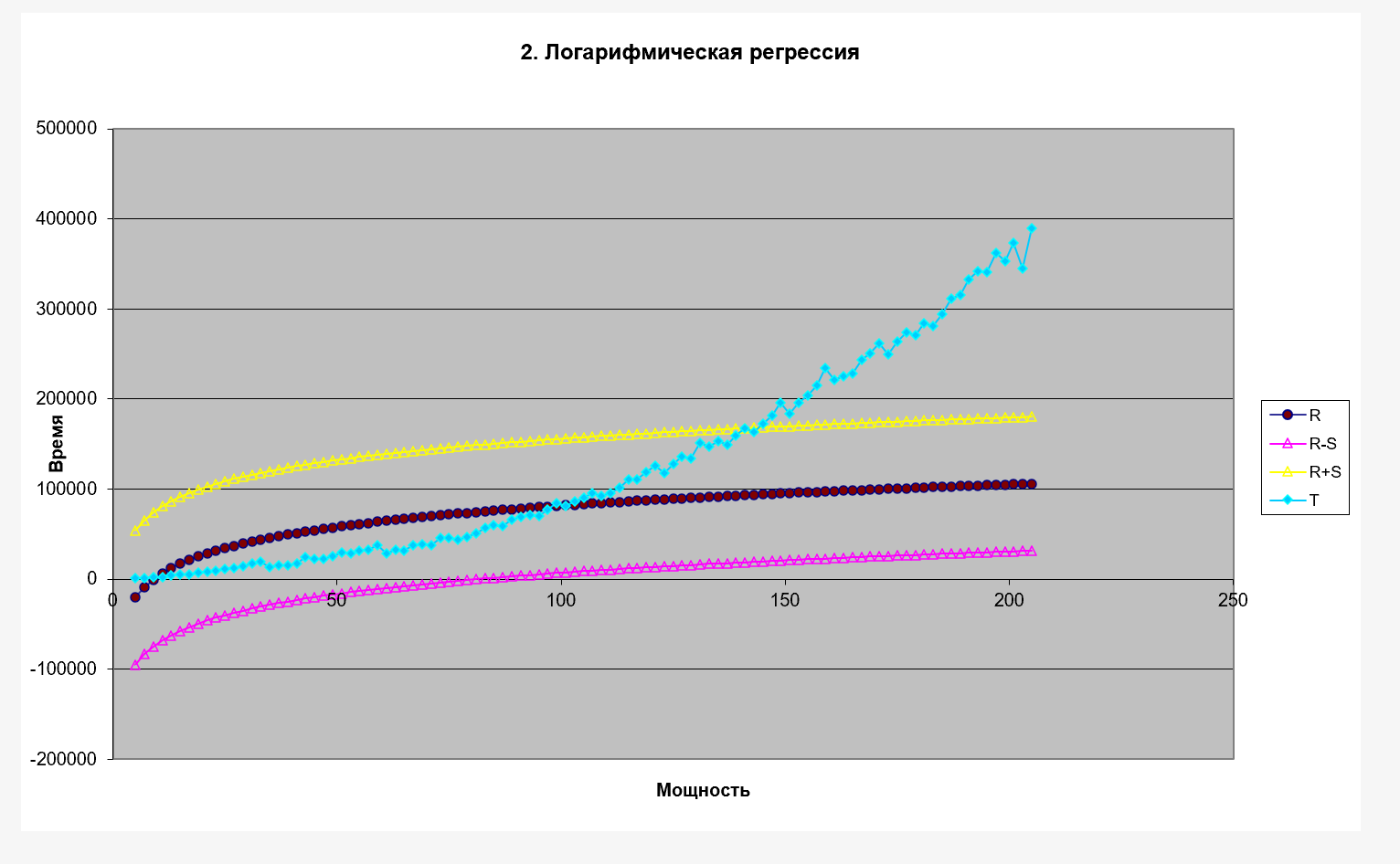
Графики были построены и отображены с использованием библиотеки matplotlib.pyplot и других инструментов Python. Применение RG41 оказалось менее удобным для визуализации данных по следующим причинам:

1. **Сравнение графиков** – В Python можно сразу увидеть все кривые временных сложностей и сопоставить их с графиком, полученным из экспериментальных данных. В RG41 же графики размещены на отдельных страницах, что усложняет их анализ.
2. **Избыточность и читаемость** – В RG41 данные представлены в избыточном формате, что снижает их наглядность.
3. **Гибкость обработки данных** – В RG41 часть измерений может не подходить под заданные параметры, тогда как в Python можно использовать все данные без исключения.
4. **Автоматизация расчетов** – Python автоматически сравнивает полученные СКО, а в RG41 это приходится делать вручную, что увеличивает время обработки и повышает риск ошибок.
5. **Проблемы с масштабированием** – Графики в RG41 имеют неоптимальный масштаб и плохую читаемость, что затрудняет их сопоставление.

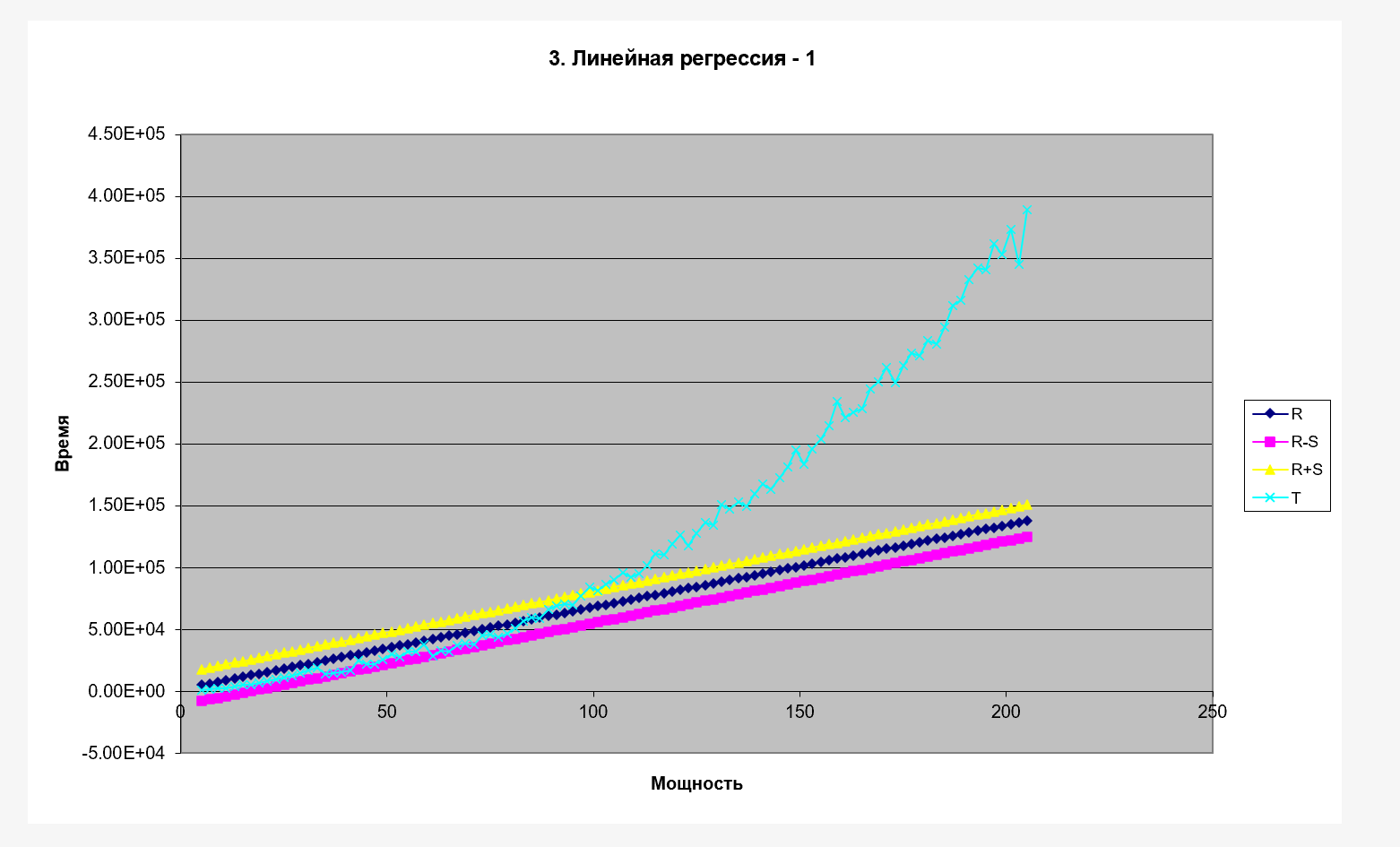
Построим графики по полученным измерениям:

1. График измеренных операций относительно друг друга в нс:

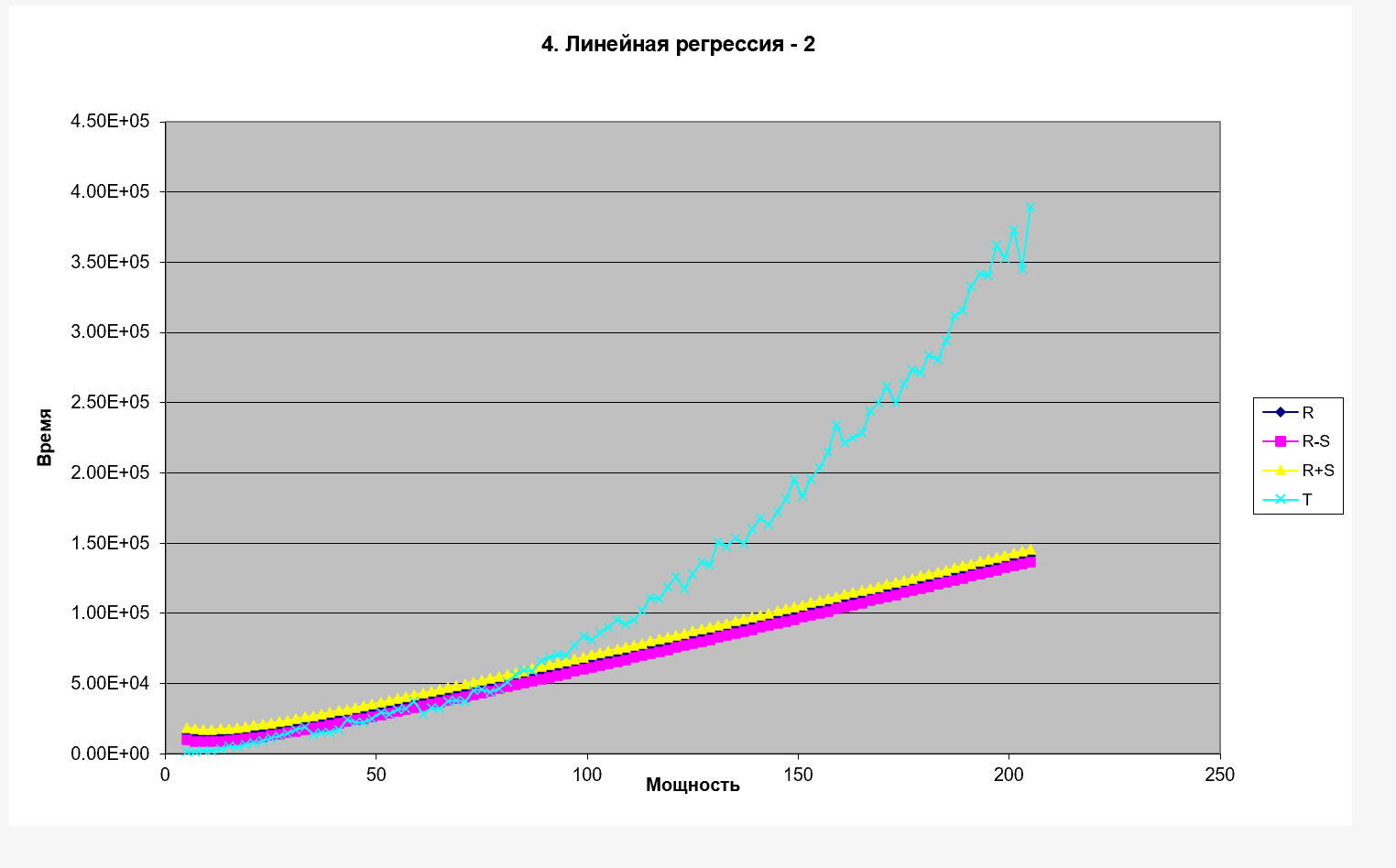


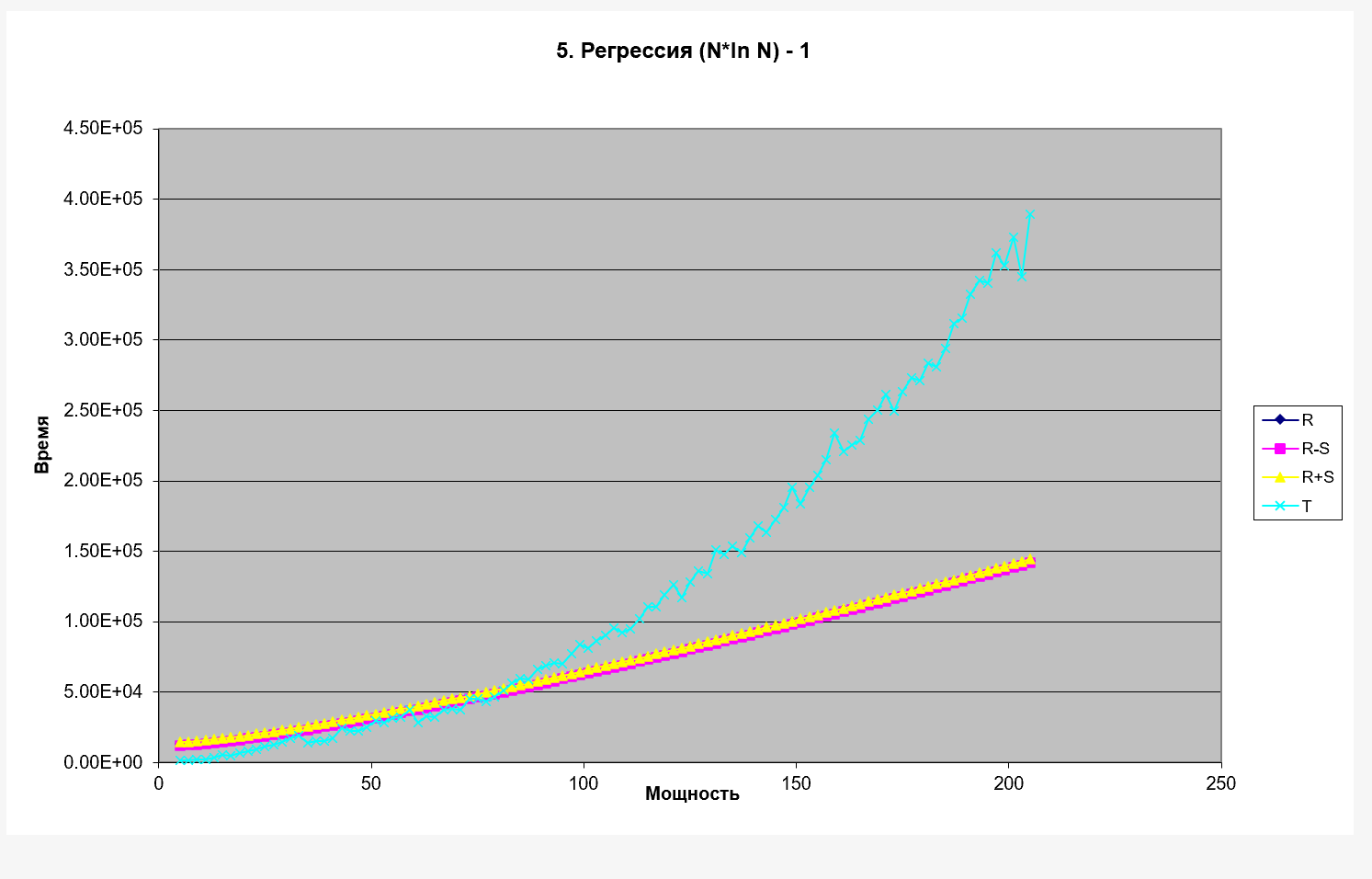


3)

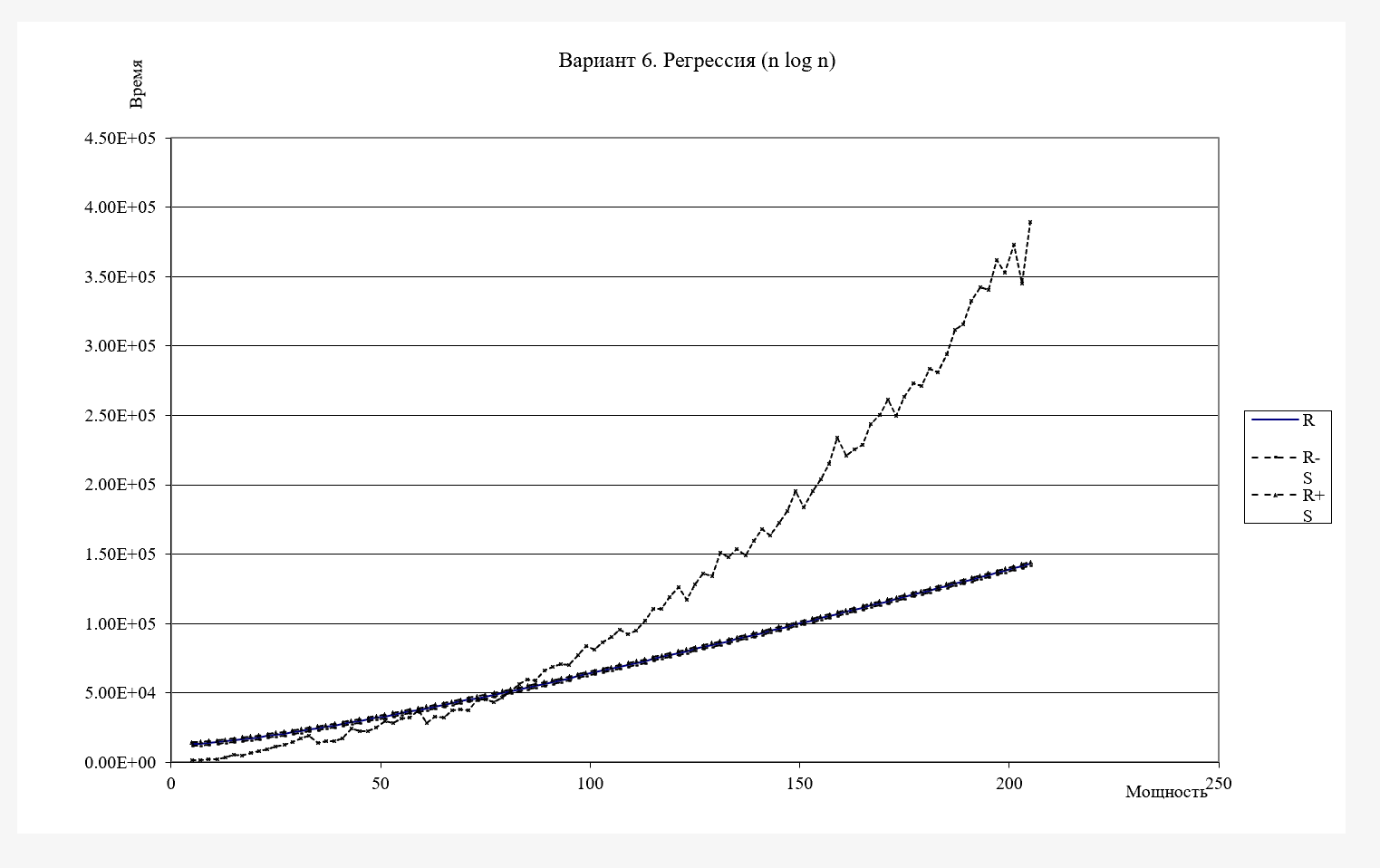


4)

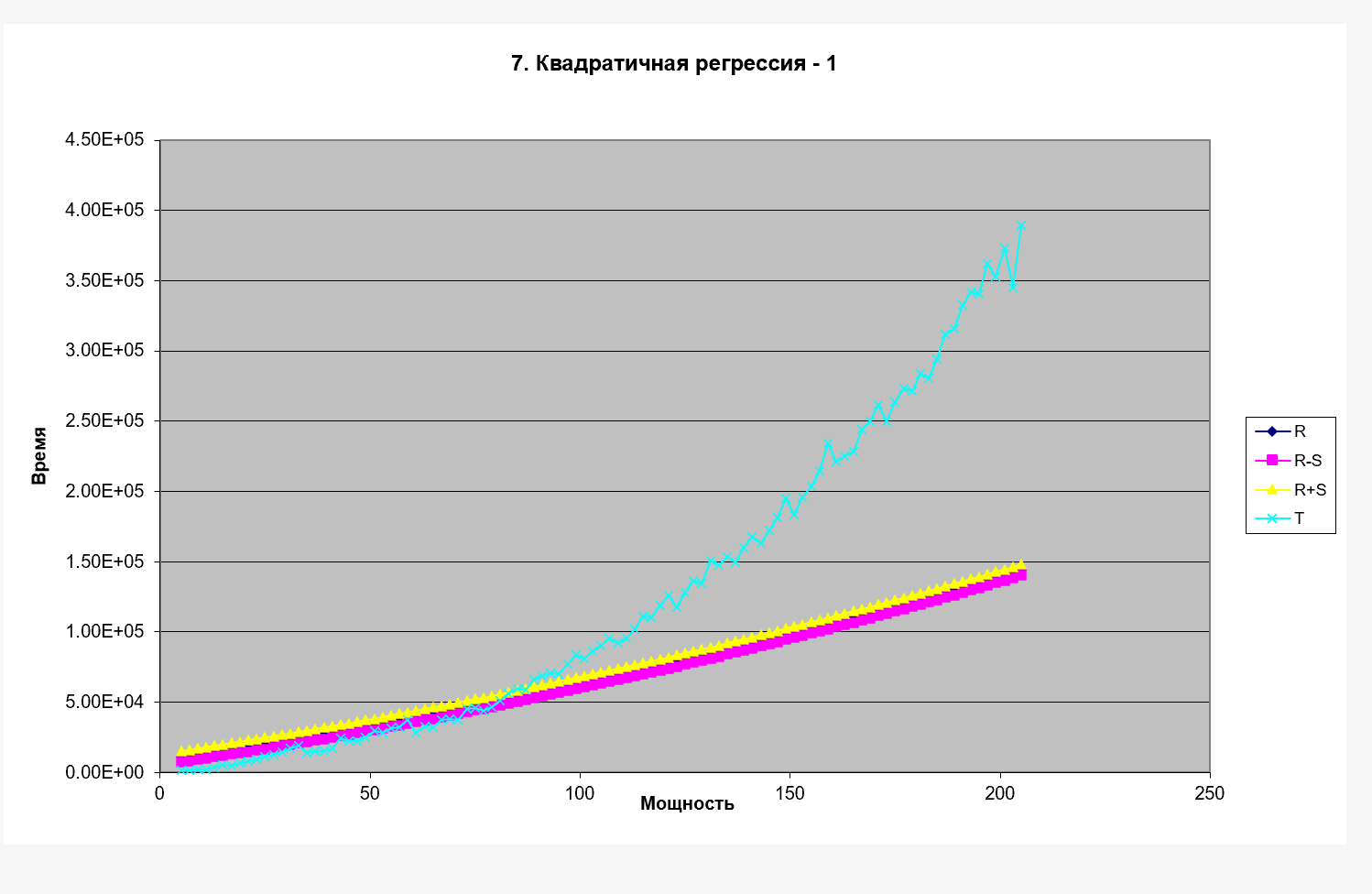


5) 

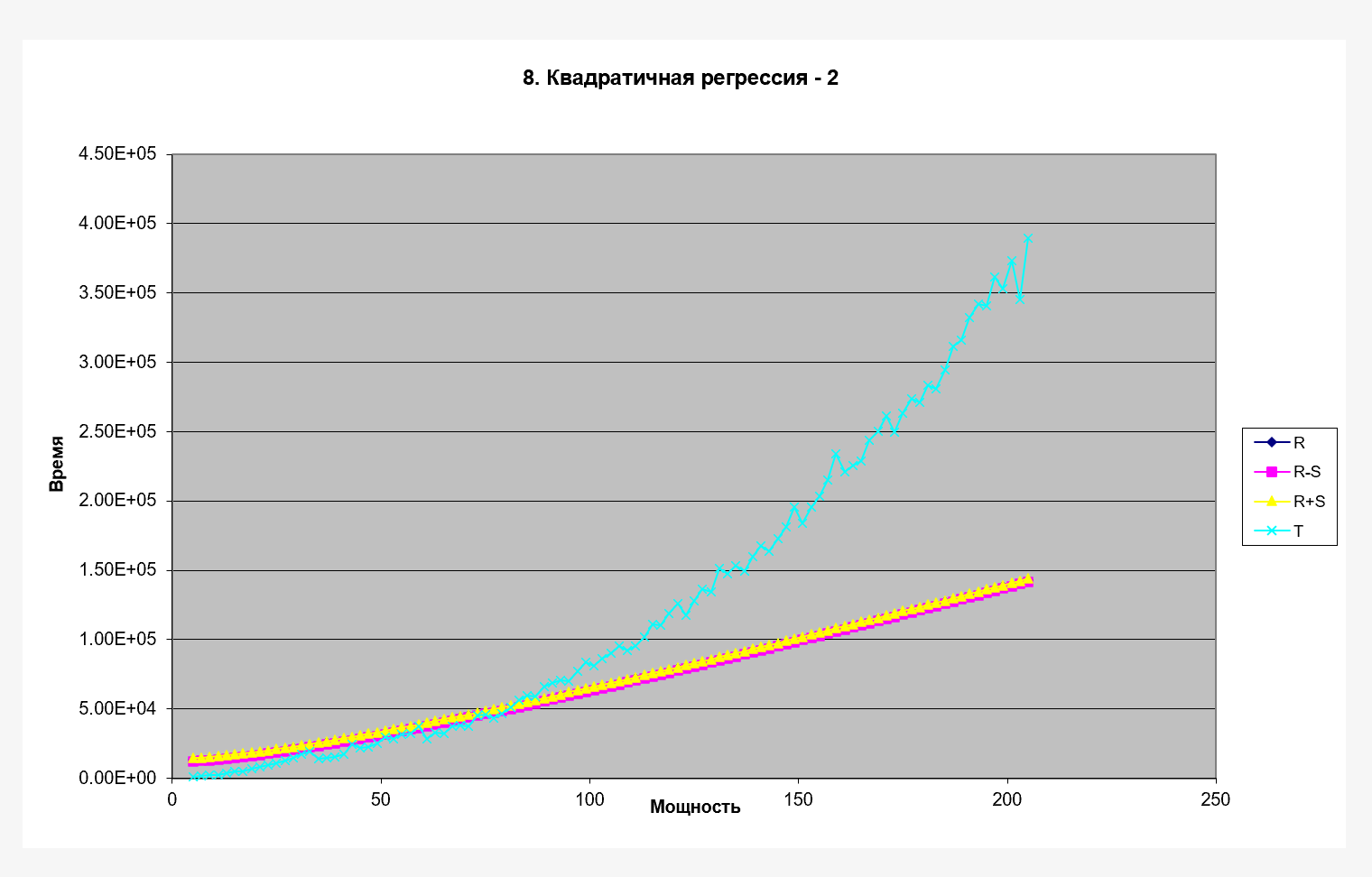
6)



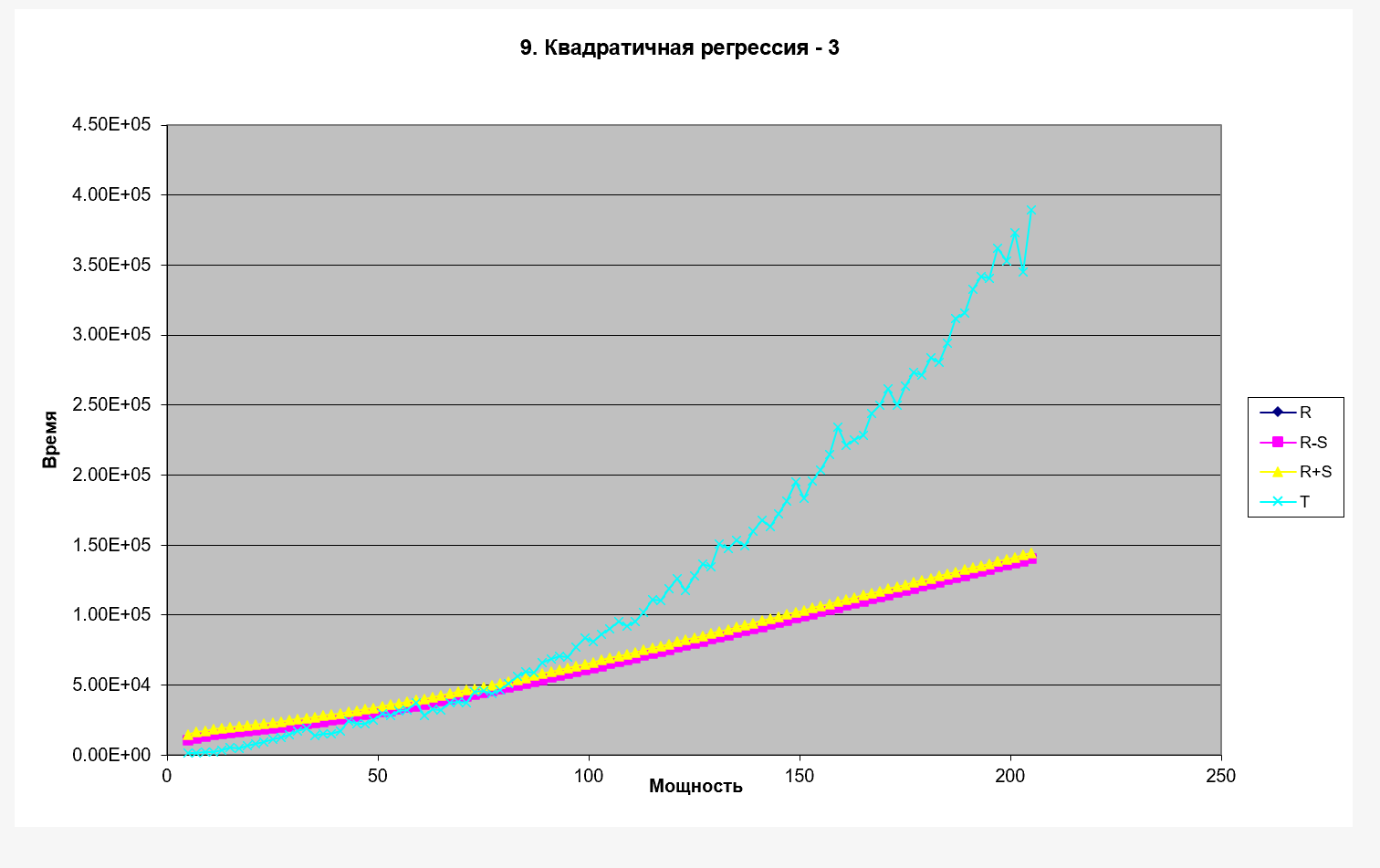
7)



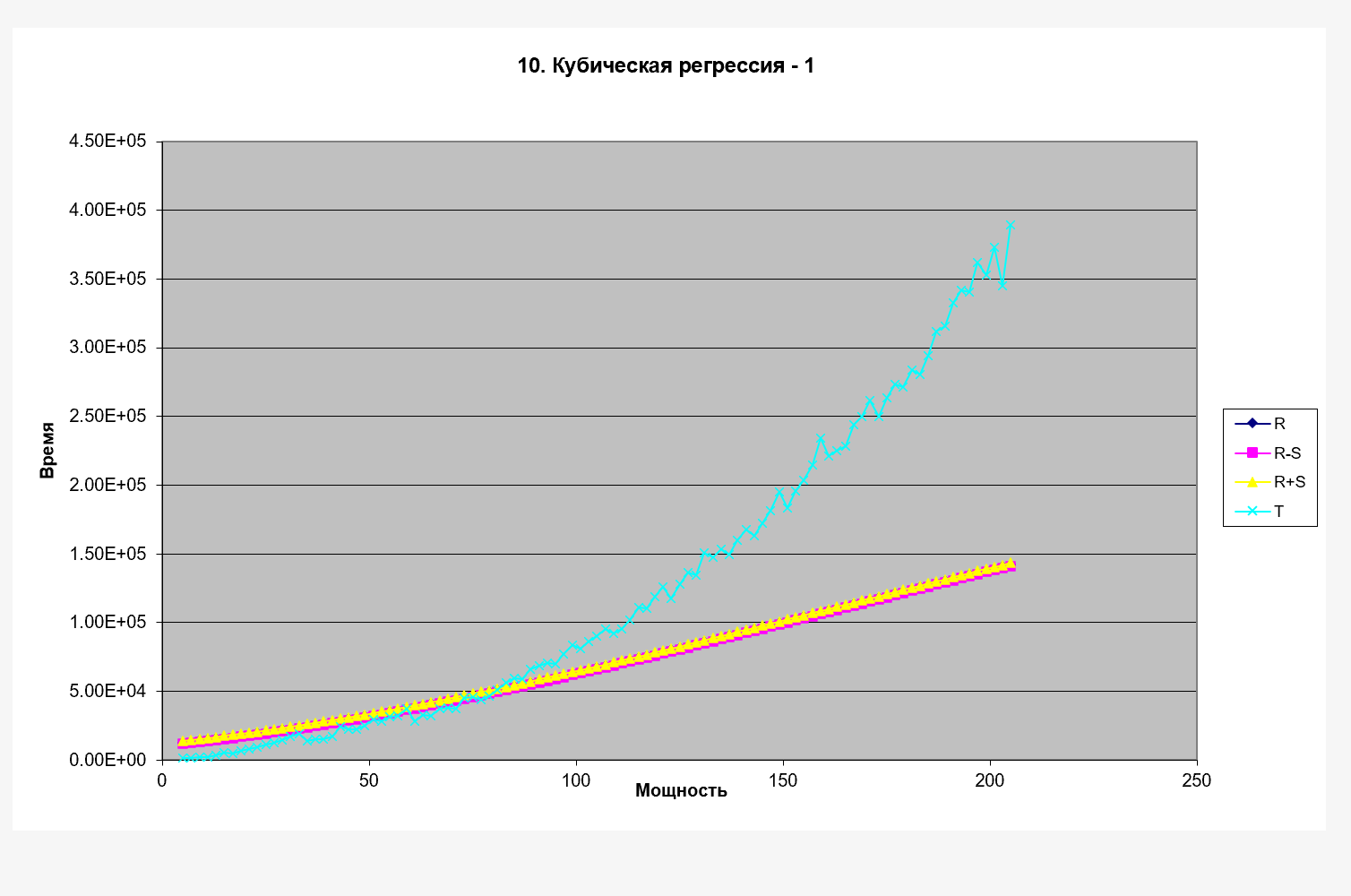
8)



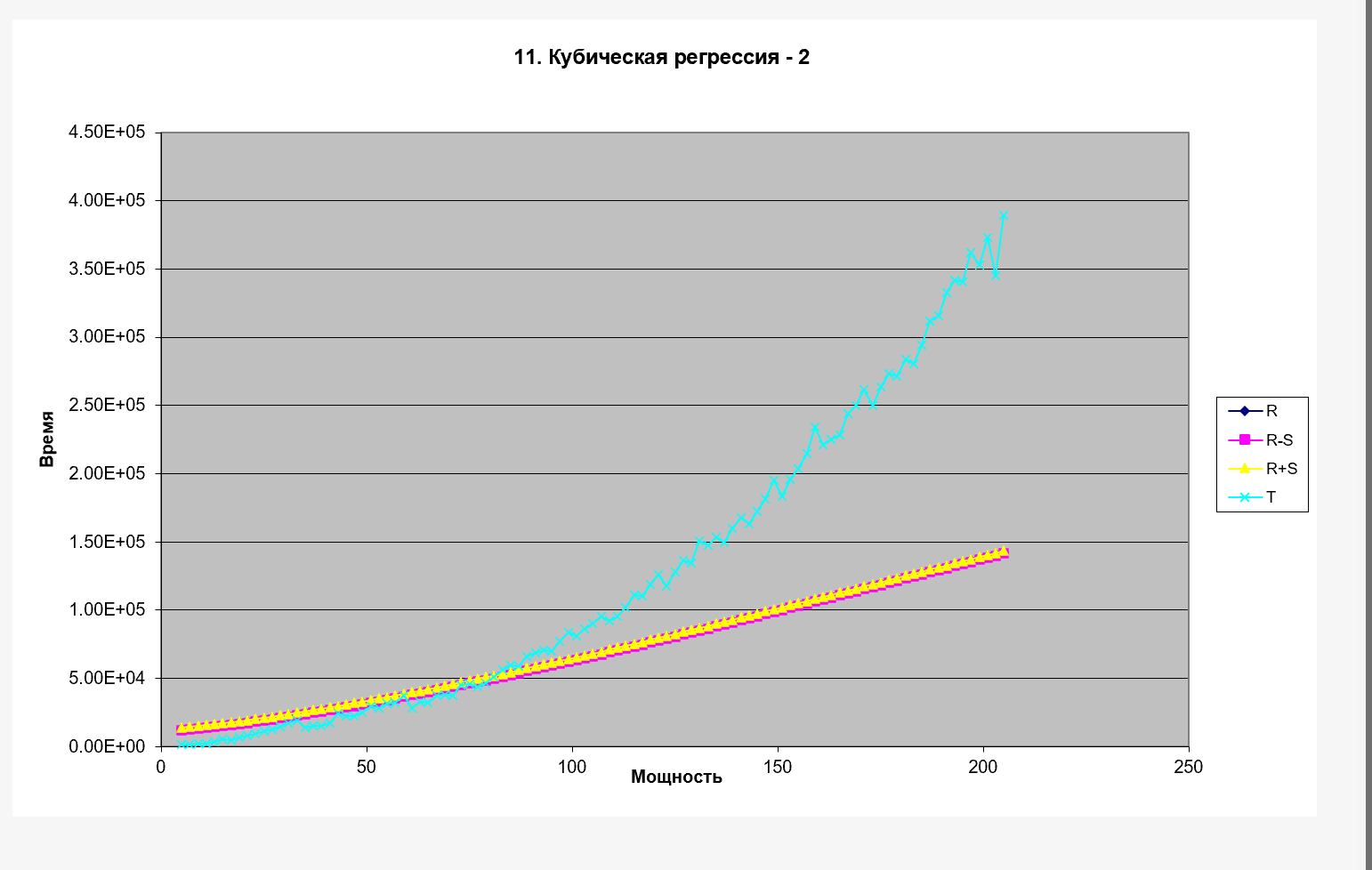
9)



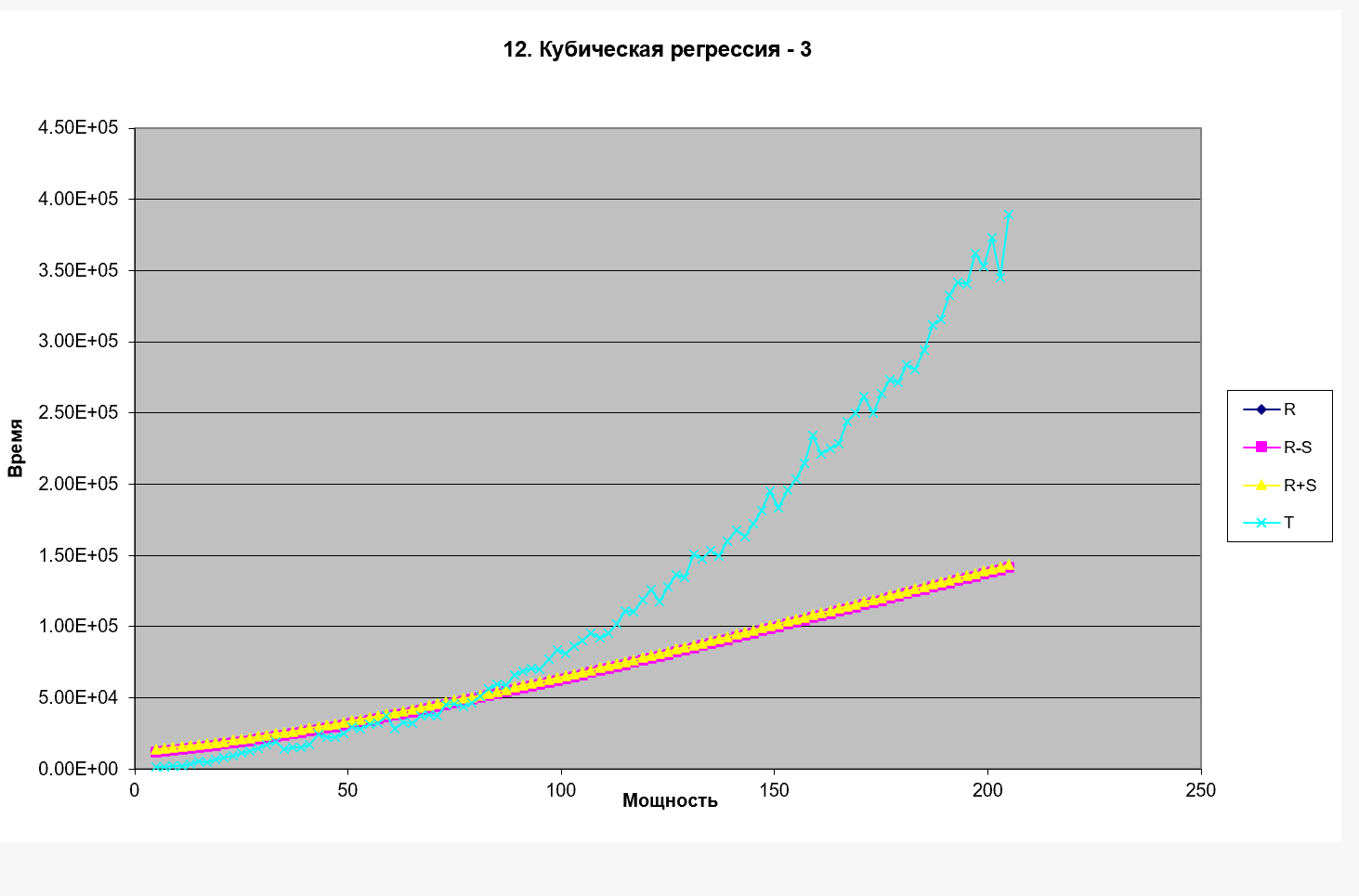
10)



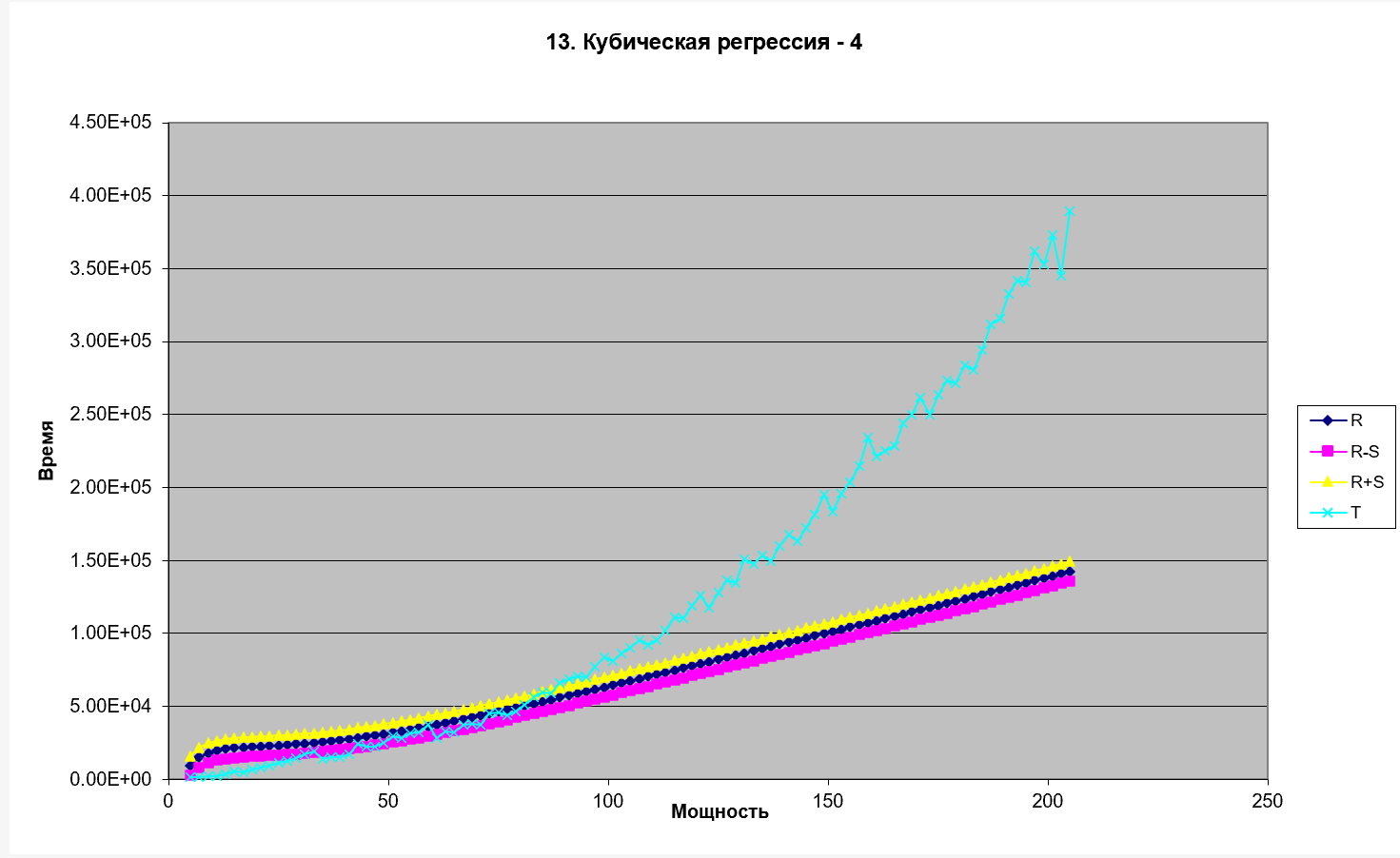
11)



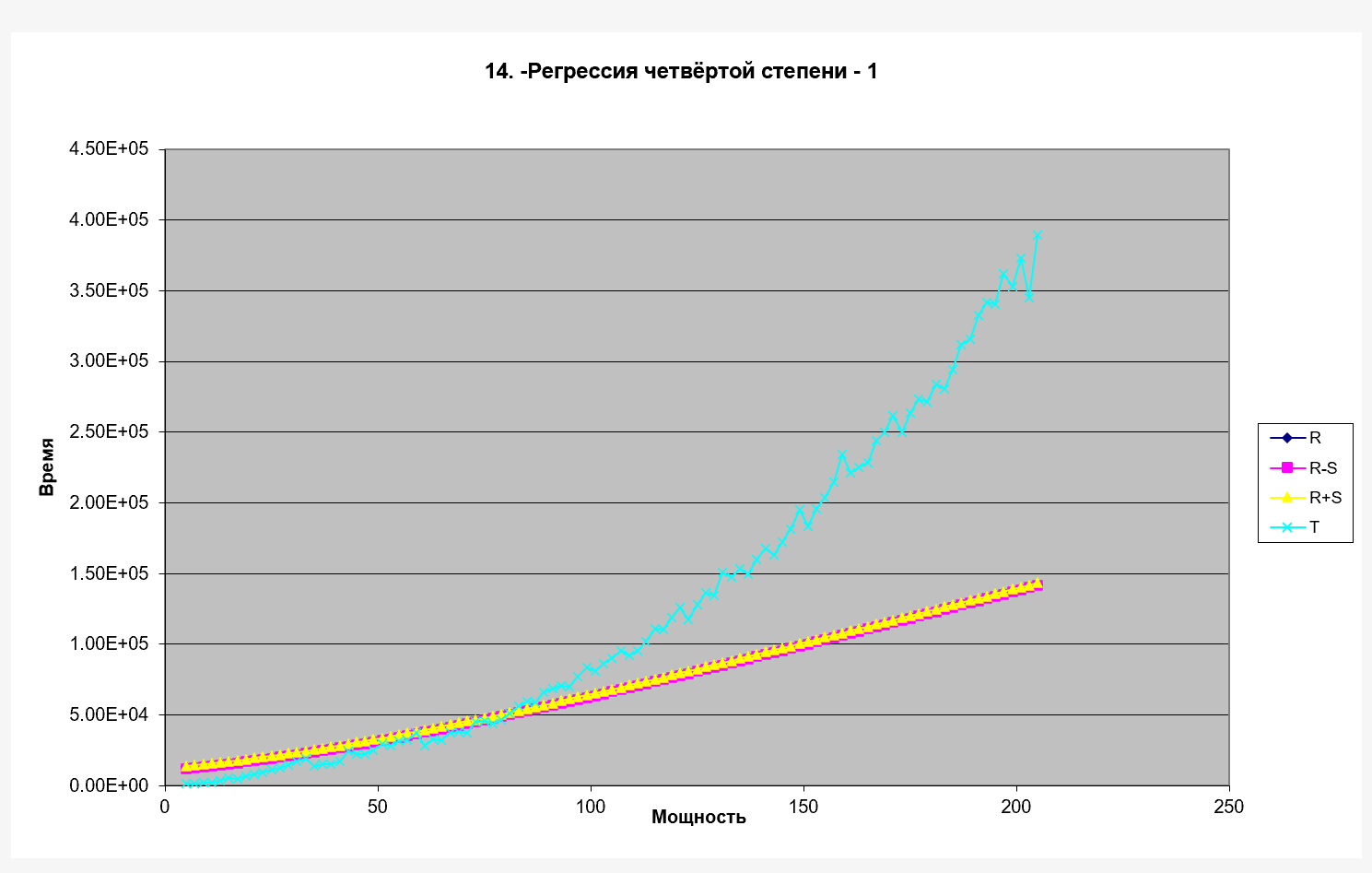
12)



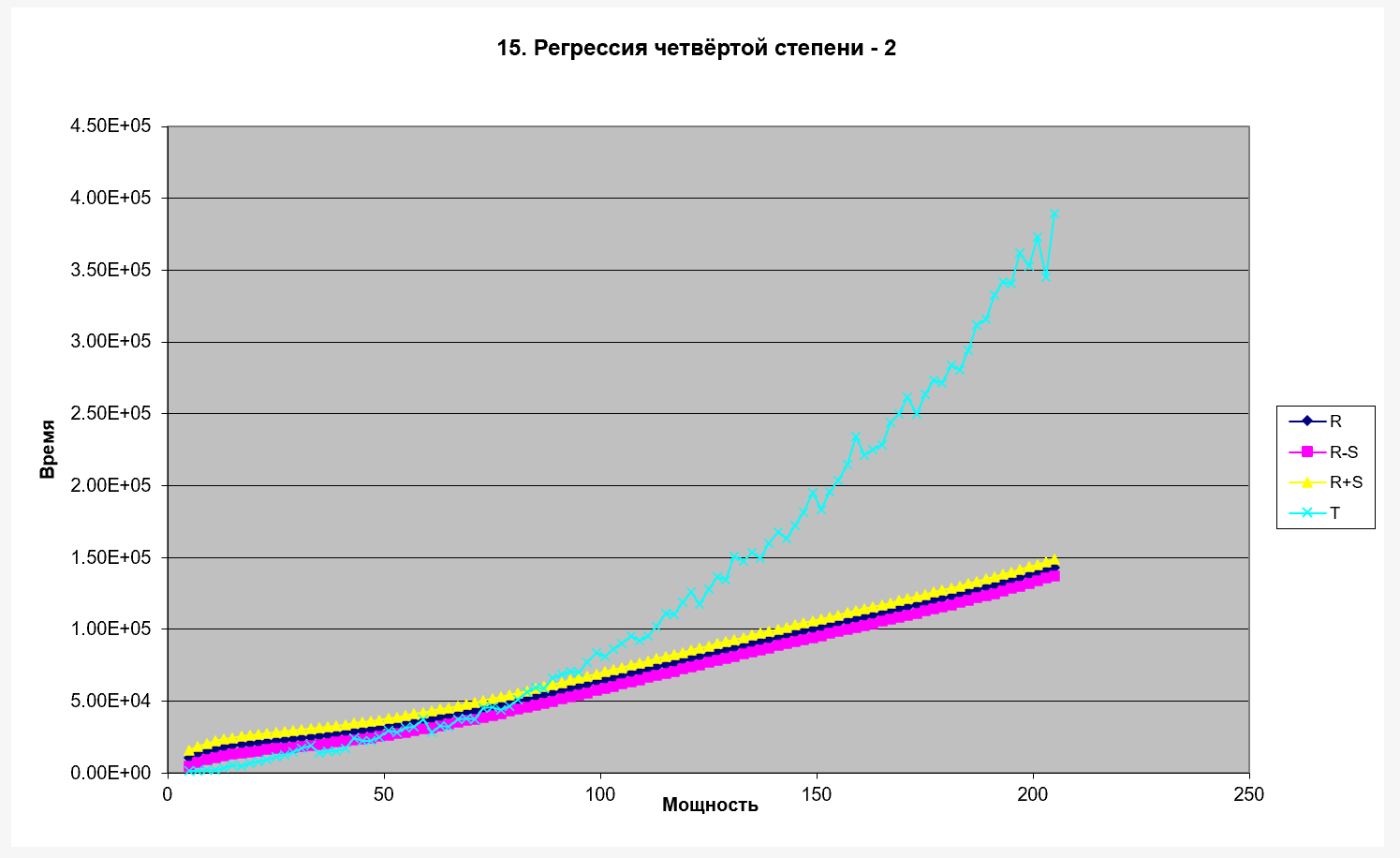
13)



14)



15)



16)



17)



**Отзыв о курсе "Алгоритмы и структуры данных"**

Курс оказался очень полезным: мы освоили ключевые концепции, такие как наследование, полиморфизм, обработка исключений, комбинированные структуры данных и работа с пользовательскими структурами. Однако, чтобы материал был ещё более практико-ориентированным, хотелось бы видеть больше примеров применения структур данных и пользовательских контейнеров в реальных задачах.

Также было бы здорово уделить больше внимания современным возможностям C++, включая:

* Использование встроенных функций стандартной библиотеки.
* Изучение новых стандартов языка.
* Написание эффективного и современного кода.

Особенно полезными были бы комплексные задания, объединяющие пройденные темы, чтобы лучше понимать, как разные аспекты C++ взаимодействуют в рамках одной программы. Кроме того, важно больше практических примеров, демонстрирующих, как эти знания можно применять в будущих профессиональных задачах, особенно при работе с данными.

В целом, курс дал нам прочную теоретическую базу и ценные практические навыки, которые, безусловно, пригодятся в дальнейшей работе.

1. 1-2-Tree.cpp

// 1-2-Tree-time.cpp

#include "1-2-Tree.h"

#include <algorithm>

#include <iostream>

#include <vector>

#include <chrono>

// Constants for tree display

const int FIRSTCOL = 80;

const int MAXCOL = 160;

const int OFFSET[] = {80, 40, 20, 10, 4, 2};

const int MAXROW = 30;

const int SHIFT = 3;

std::string SCREEN[MAXROW];

int SET\_SIZE;

std::chrono::high\_resolution\_clock::time\_point start, finish;

std::chrono::high\_resolution\_clock::time\_point getTime() {

return std::chrono::high\_resolution\_clock::now();

}

// Display methods for Node

void Node::Out(int row, int col) const {

if (row >= MAXROW || col < 0 || col >= MAXCOL) return;

std::string keyStr = std::to\_string(key);

int len = std::min(3, (int)keyStr.length());

SCREEN[row].replace(col, len, keyStr.substr(0, len));

if (next && right) {

std::string rightStr = std::to\_string(right->key);

len = std::min(3, (int)rightStr.length());

SCREEN[row+1].replace(col+1, len, rightStr.substr(0, len));

}

}

void Node::Display(int level, int col, int& max\_level) const {

if (level > max\_level) max\_level = level;

int current\_offset = (level+1 < (int)(sizeof(OFFSET)/sizeof(OFFSET[0]))) ?

OFFSET[level+1] : OFFSET[sizeof(OFFSET)/sizeof(OFFSET[0])-1];

Out(level\*2, col);

if (left) left->Display(level+1, col - current\_offset, max\_level);

if (right) {

if (next) {

if (right->left) right->left->Display(level+1, col, max\_level);

if (right->right) right->right->Display(level+1, col + current\_offset, max\_level);

} else {

right->Display(level+1, col + current\_offset, max\_level);

}

}

}

// Tree display method

void Tree::Display() const {

std::fill\_n(SCREEN, MAXROW, std::string(MAXCOL, ' '));

int max\_level = 0;

if (root) {

root->Display(0, FIRSTCOL - SHIFT, max\_level);

} else {

SCREEN[0] = "Empty Tree";

}

for (int i = 0; i <= max\_level\*2+2 && i < MAXROW; ++i) {

std::string line = SCREEN[i];

line.erase(line.find\_last\_not\_of(' ') + 1);

std::cout << line << '\n';

}

}

// Iterator increment operator

myiter& myiter::operator++() {

if (!Ptr) return \*this;

if (Ptr->right) {

St.push({Ptr, 1});

Ptr = Ptr->right;

while (Ptr->left) {

St.push({Ptr, 0});

Ptr = Ptr->left;

}

} else {

auto pp = std::make\_pair(Ptr, 1);

while (!St.empty() && pp.second) {

pp = St.top();

St.pop();

}

Ptr = pp.second ? nullptr : pp.first;

}

return \*this;

}

// Tree iterator methods

myiter Tree::begin() const {

MyStack St;

Node\* p = root;

while (p && p->left) {

St.push({p, 0});

p = p->left;

}

return myiter(p, St);

}

// Tree insertion method

std::pair<myiter, bool> Tree::insert(int k, myiter where) {

Node \*t = root;

bool cont = true, up = false;

MyStack St;

int current\_depth = 0;

if (!where.Ptr) {

if (!t) {

root = new Node(k);

count = 1;

tree\_height = 1;

return {myiter(root), true};

}

St.push({root, 1});

current\_depth = 1;

} else {

t = where.Ptr;

St = std::move(where.St);

current\_depth = St.size();

}

while (cont) {

if (is\_set && k == t->key) {

return {myiter(t, std::move(St)), false};

}

if (k <= t->key) { // Changed from < to <= for sequence mode

if (t->left) {

St.push({t, 2});

t = t->left;

current\_depth++;

} else {

t->left = new Node(k);

cont = false;

if (current\_depth + 1 > tree\_height) {

tree\_height = current\_depth + 1;

}

}

} else {

if (!t->right) {

t->right = new Node(k);

t->next = true;

cont = false;

if (current\_depth + 1 > tree\_height) {

tree\_height = current\_depth + 1;

}

} else if (t->next) {

if (k <= t->right->key) { // Changed from < to <=

if (t->right->left) {

St.push({t, 3});

t = t->right->left;

current\_depth++;

} else {

t->right->left = new Node(k);

cont = false;

if (current\_depth + 2 > tree\_height) {

tree\_height = current\_depth + 2;

}

}

} else {

if (t->right->right) {

St.push({t, 4});

t = t->right->right;

current\_depth++;

} else {

t->right->right = new Node(k);

up = t->right->next = true;

cont = false;

if (current\_depth + 2 > tree\_height) {

tree\_height = current\_depth + 2;

}

}

}

} else {

if (t->right) {

St.push({t, 3});

t = t->right;

current\_depth++;

} else {

t->right = new Node(k);

t->next = true;

cont = false;

if (current\_depth + 1 > tree\_height) {

tree\_height = current\_depth + 1;

}

}

}

}

while (up) {

std::swap(t->key, t->right->key);

Node \*t1 = t->right;

t->next = t1->next = false;

t->right = t1->right;

t1->right = t1->left;

t1->left = t->left;

t->left = t1;

t1 = t;

if (St.empty()) {

up = false;

tree\_height++;

} else {

t = St.top().first;

switch (St.top().second) {

case 2:

std::swap(t->key, t1->key);

t->left = t1->left;

t1->left = t1->right;

t1->right = t->right;

t->right = t1;

up = t1->next = t->next;

break;

case 3:

if (t->next) {

t->right->left = t1->right;

t1->right = t->right;

t->right = t1;

t1->next = true;

} else {

t->next = true;

up = t1->next = false;

St.pop();

}

break;

case 4:

t->right->next = true;

t1->next = false;

break;

}

if (up) St.pop();

}

}

}

++count;

return {myiter(t, std::move(St)), true};

}

// Tree erase method

bool Tree::erase(int k) {

Node \*p = nullptr, \*q = root;

int a = 0;

bool cont = (q != nullptr);

while (cont && (k != q->key)) {

p = q;

a = (k > q->key);

q = a ? q->right : q->left;

cont = (q != nullptr);

}

if (!cont) return false;

Node \*r = q->right;

if (r && r->left) {

p = q;

a = 1;

do {

p = r;

r = r->left;

a = 0;

} while (r->left);

q->key = r->key;

p->left = r->right;

q = r;

}

else if (r) {

(a ? r->right : r->left) = q->left;

if (p) (a ? p->right : p->left) = r;

else root = r;

}

else {

if (p) {

if (a) {

p->right = q->left;

p->next = false;

}

else {

p->left = q->left;

}

}

else {

root = q->left;

}

}

q->left = q->right = nullptr;

--count;

delete q;

return true;

}

int random\_int(int min, int max) {

static std::random\_device rd;

static std::mt19937 gen(rd());

std::uniform\_int\_distribution<> distrib(min, max);

return distrib(gen);

}

template<typename Iter>

void printSet(const std::string& name, Iter begin, Iter end) {

std::cout << name << " = { ";

for (; begin != end; ++begin) {

std::cout << \*begin << " ";

}

std::cout << "}\n";

}

Tree generate\_random\_set() {

Tree result(true); // Set mode

while (result.size() < SET\_SIZE) {

int num = random\_int(1, SET\_SIZE\*10);

result.insert(num);

}

return result;

}

Tree generate\_random\_sequence() {

Tree result(false); // Sequence mode

for (int i = 0; i < SET\_SIZE; ++i) {

int num = random\_int(1, SET\_SIZE\*10);

result.insert(num);

}

return result;

}

void execute\_set\_operations() {

try {

Tree A = generate\_random\_set();

Tree B = generate\_random\_set();

Tree C = generate\_random\_set();

Tree D = generate\_random\_set();

Tree E = generate\_random\_set();

std::cout << "Generated sets:\n";

printSet("A", A.begin(), A.end());

printSet("B", B.begin(), B.end());

printSet("C", C.begin(), C.end());

printSet("D", D.begin(), D.end());

printSet("E", E.begin(), E.end());

std::cout << "----------------------------------\n";

Tree A\_minus\_B(true);

std::set\_difference(A.begin(), A.end(), B.begin(), B.end(),

outinserter(A\_minus\_B, A\_minus\_B.begin()));

printSet("A minus B", A\_minus\_B.begin(), A\_minus\_B.end());

std::cout << "----------------------------------\n";

Tree A\_minus\_B\_union\_C(true);

std::set\_union(A\_minus\_B.begin(), A\_minus\_B.end(),

C.begin(), C.end(),

outinserter(A\_minus\_B\_union\_C, A\_minus\_B\_union\_C.begin()));

printSet("(A minus B) union C", A\_minus\_B\_union\_C.begin(), A\_minus\_B\_union\_C.end());

std::cout << "----------------------------------\n";

Tree D\_intersect\_E(true);

std::set\_intersection(D.begin(), D.end(),

E.begin(), E.end(),

outinserter(D\_intersect\_E, D\_intersect\_E.begin()));

printSet("D intersect E", D\_intersect\_E.begin(), D\_intersect\_E.end());

std::cout << "----------------------------------\n";

Tree result(true);

std::set\_symmetric\_difference(A\_minus\_B\_union\_C.begin(), A\_minus\_B\_union\_C.end(),

D\_intersect\_E.begin(), D\_intersect\_E.end(),

outinserter(result, result.begin()));

std::cout << "Final result: (A minus B union C) symmetric\_diff (D intersect E)\n";

printSet("Result", result.begin(), result.end());

std::cout << "Tree visualization:\n";

result.Display();

}

catch (const std::exception& e) {

std::cerr << "Error in set operations: " << e.what() << std::endl;

}

}

void execute\_sequence\_operations() {

try {

int choice = 0;

std::cout << "\nSequence Operations:\n";

std::cout << "1. Merge two sequences\n";

std::cout << "2. Concatenate two sequences\n";

std::cout << "3. Erase range from sequence\n";

std::cout << "Enter your choice: ";

std::cin >> choice;

Tree seq1 = generate\_random\_sequence();

if (choice == 3) {

std::cout << "\nGenerated sequence:\n";

printSet("Sequence", seq1.begin(), seq1.end());

int startindex, end;

std::cout << "Enter start and end positions to erase (1-based): ";

std::cin >> startindex >> end;

start = getTime();

if (startindex < 1 || end > seq1.size() || startindex > end) {

std::cout << "Invalid range!\n";

return;

}

// Convert to vector for range deletion

std::vector<int> elements(seq1.begin(), seq1.end());

elements.erase(elements.begin() + startindex - 1, elements.begin() + end);

// Rebuild the tree

seq1.clear();

for (int val : elements) {

seq1.insert(val);

}

finish = getTime();

std::cout << std::chrono::duration\_cast<std::chrono::microseconds>(finish - start).count() << std::endl;

printSet("Sequence after erase", seq1.begin(), seq1.end());

std::cout << "Tree visualization:\n";

seq1.Display();

}

else {

Tree seq2 = generate\_random\_sequence();

std::cout << "\nGenerated sequences:\n";

printSet("Sequence 1", seq1.begin(), seq1.end());

printSet("Sequence 2", seq2.begin(), seq2.end());

if (choice == 1) {

// Merge needs sorted sequences

start = getTime();

std::vector<int> temp1(seq1.begin(), seq1.end());

std::vector<int> temp2(seq2.begin(), seq2.end());

std::sort(temp1.begin(), temp1.end());

std::sort(temp2.begin(), temp2.end());

Tree result(false); // Sequence mode

std::merge(temp1.begin(), temp1.end(),

temp2.begin(), temp2.end(),

outinserter(result, result.begin()));

printSet("Merged sequence", result.begin(), result.end());

finish = getTime();

std::cout << std::chrono::duration\_cast<std::chrono::microseconds>(finish - start).count() << std::endl;

std::cout << "Tree visualization:\n";

result.Display();

}

else if (choice == 2) {

// Simple concatenation - first output as vector

std::vector<int> concat\_result;

concat\_result.insert(concat\_result.end(), seq1.begin(), seq1.end());

concat\_result.insert(concat\_result.end(), seq2.begin(), seq2.end());

start = getTime();

std::cout << "Concatenated sequence (as vector):\n";

printSet("Result", concat\_result.begin(), concat\_result.end());

// Now build tree from concatenated sequence

Tree result\_tree(false);

for (int val : concat\_result) {

result\_tree.insert(val);

}

finish = getTime();

std::cout << std::chrono::duration\_cast<std::chrono::microseconds>(finish - start).count() << std::endl;

std::cout << "Tree visualization of concatenated sequence:\n";

result\_tree.Display();

}

else {

std::cout << "Invalid choice!\n";

return;

}

}

}

catch (const std::exception& e) {

std::cerr << "Error in sequence operations: " << e.what() << std::endl;

}

}

int main() {

std::cout << "Enter set size: " << std::endl;

std::cin >> SET\_SIZE;

try {

int choice = 0;

while (true) {

std::cout << "\nMenu:\n";

std::cout << "1. Set operations\n";

std::cout << "2. Sequence operations\n";

std::cout << "3. Exit\n";

std::cout << "4. Change set size\n";

std::cout << "Enter your choice: ";

std::cin >> choice;

switch (choice) {

case 1:

start = getTime();

execute\_set\_operations();

finish = getTime();

std::cout << std::chrono::duration\_cast<std::chrono::microseconds>(finish - start).count() << std::endl;

break;

case 2:

execute\_sequence\_operations();

break;

case 3:

return 0;

case 4:

std::cout << "Enter new set size: ";

std::cin >> SET\_SIZE;

default:

std::cout << "Invalid choice. Please try again.\n";

}

}

}

catch (const std::exception& e) {

std::cerr << "Fatal error: " << e.what() << std::endl;

return 1;

}

}

1. 1-2-Tree.h

// 1-2-Tree.h

#ifndef TREE\_1\_2\_H

#define TREE\_1\_2\_H

#include <iostream>

#include <string>

#include <stack>

#include <utility>

#include <algorithm>

#include <vector>

#include <random>

// Node structure for 1-2 Tree

struct Node {

int key;

Node \*left, \*right;

bool next = false; // true if node contains 2 keys

Node(int k, Node \*l = nullptr, Node \*r = nullptr)

: key(k), left(l), right(r) {}

~Node() { delete left; delete right; }

void Display(int level, int col, int& max\_level) const;

void Out(int row, int col) const;

};

using MyStack = std::stack<std::pair<Node\*, int>>;

// Iterator for Tree class

struct myiter {

using iterator\_category = std::forward\_iterator\_tag;

using value\_type = int;

using difference\_type = std::ptrdiff\_t;

using pointer = int\*;

using reference = int&;

myiter(Node \*p = nullptr) : Ptr(p) {}

myiter(Node \*p, MyStack St) : Ptr(p), St(St) {}

bool operator==(const myiter &Other) const { return Ptr == Other.Ptr; }

bool operator!=(const myiter &Other) const { return !(\*this == Other); }

myiter& operator++();

myiter operator++(int) { myiter tmp(\*this); ++\*this; return tmp; }

reference operator\*() const { return Ptr->key; }

pointer operator->() const { return &Ptr->key; }

Node \*Ptr;

MyStack St;

};

// 1-2 Tree class implementation

class Tree {

Node \*root = nullptr;

int count = 0;

int tree\_height = 0;

bool is\_set = true; // Flag to distinguish between set and sequence

public:

using value\_type = int;

Tree(bool is\_set = true) : is\_set(is\_set) {}

~Tree() { delete root; }

std::pair<myiter, bool> insert(int k, myiter where = myiter());

bool erase(int k);

void clear() { delete root; root = nullptr; count = 0; tree\_height = 0; }

int size() const { return count; }

int height() const { return tree\_height; }

myiter begin() const;

myiter end() const { return myiter(); }

void Display() const;

template<typename Iter>

Tree(Iter first, Iter last, bool is\_set = true) : is\_set(is\_set) {

for (; first != last; ++first) insert(\*first);

}

void set\_mode(bool set\_mode) { is\_set = set\_mode; }

};

// Output iterator adapter for set operations

template <typename Container>

class outiter {

Container& container;

typename Container::value\_type value;

myiter iter;

public:

using iterator\_category = std::output\_iterator\_tag;

explicit outiter(Container& c, myiter it) : container(c), iter(it) {}

outiter& operator\*() { return \*this; }

outiter& operator++() { return \*this; }

outiter& operator++(int) { return \*this; }

outiter& operator=(const typename Container::value\_type& val) {

value = val;

iter = container.insert(value, iter).first;

return \*this;

}

};

template <typename Container>

inline outiter<Container> outinserter(Container& c, myiter it) {

return outiter<Container>(c, it);

}

int random\_int(int min, int max);

// Function declarations

void execute\_set\_operations();

void execute\_sequence\_operations();

template<typename Iter>

void printSet(const std::string& name, Iter begin, Iter end);

#endif