СРАВНЕНИЕ РАБОТЫ АЛГОРИТМОВ СОРТИРОВКИ, РЕАЛИЗОВАННЫХ НА ЯЗЫКЕ **PERL.**

Выполнил: **Самунь Виктор**, группа КН-201, e-mail: victor.samun@uralweb.ru

I. ТЕСТИРУЕМЫЕ АЛГОРИТМЫ

Нами будет проведено тестирование следующих алгоритмов:

- Встроенный алгоритм сортировки;
- Быстрая сортировка (сортировка Хоара);
- Сортировка слиянием;
- Сортировка кучей (пирамидальная сортировка);
- Сортировка посредством бинарных вставок;
- Сортировка посредством простых (линейных) вставок;
- Сортировка Бэтчера (Batcher);
- Сортировки Шелла (8 модификаций);
- Сортировка выбором;
- «Пузырьковая» сортировка.

Каждый из приведенных алгоритмов будет описан ниже более подробно.

II. ОПИСАНИЕ ТЕСТИРОВАНИЯ

Для проведения тестирования (измерение времени работы алгоритма, подсчет количества операций над сортируемым массивом) мы специальным образом организовали тестирующую систему.

Сценарий PERL `sorting.pl`, реализующий алгоритмы сортировки имеет специальные ключи[1] запуска для обеспечения:

Отключения выдачи отсортированных данных;

Измерения времени работы алгоритма;

Подсчета количества операций над сортируемым массивом.

Заметим, что система устроена таким образом, что нельзя одновременно проводить измерение времени работы алгоритма и считать количество операций. Это сделано потому, что подсчет количества операций приводит к заметным накладным расходам, а поэтому сравнивать времена работы, измеренные с достаточно большими ненужными накладными расходами (в процессе тестирования было заметно, что время работы снижается примерно в 1,5 раза) будет некорректно.

Основной сценарий `sorting.pl` в самом начале подключает 3 модуля: модуль с сортировками Шелла, простой модуль с остальными алгоритмами сортировки (на котором измерялось время работы) и модуль с алгоритмами сортировки, которые поддерживают формирование отчета о количестве проделанных операций. В модуле с сортировками Шелла эти возможности реализуются двумя разными функциями сортировки.

[1] Ключи запуска можно посмотреть, запустив сценарий `sorting.pl` из командной строки: `sorting.pl -?`

Далее были созданы специальные сценарии на языке **PERL**, которые генерируют файл с входными данными для сортировки; которые запускают генерацию тестов (тестов было создано много и выполнение более ста запусков сценария вручную – нерациональное занятие. Поэтому мы написали сценарии, которые это делают за нас); и те, которые будут выполнять запуск сценария сортировки. А также был создан отладочный сценарий `tester.pl`, который проверяет, результат сортировки правильный или нет (он просто проверяет, что данные результата образуют монотонную последовательность и что все данные присутствуют).

Далее мы сгенерировали входные данные и запустили тестирование.

III. ПРОВОДИМЫЕ ТЕСТЫ

Для каждого алгоритма тестирования мы сначала провели предварительный анализ того, какое количество времени алгоритмы работают в зависимости от размеров входных данных. Опытным путем было установлено, что все алгоритмы сортировки, кроме нулевой модификации сортировки Шелла (shell0), простых вставок, сортировки выбором и пузырьком, работают достаточно быстро и для них было проведено тестирование с входными данными до 1500000 записей; для остальных (которые работают медленно) – этот предел составил всего 100000 записей, т.е. в 15 раз меньше.

Для подсчета количества операций были использованы входные данные до 10000 записей.

Отметим, что каждый тест (на измерение времени) мы будем запускать по 10 раз из-за того, что точное время работы мы измерить не можем, а значит, будем иметь погрешности в измерениях времени работы. Но если есть погрешности в измерении тестирования, то нужно будет запускать один и тот же тест несколько раз, а результатом (временем работы) в этом случае следует считать среднее арифметическое 10 пройденных тестов. А тесты на измерение количества операций будем запускать по одному разу, так как погрешностей тут и быть не может. Сразу скажем, что время мы измеряли встроенной функцией `localtime` и погрешность измерений составляла 1 секунду.

IV. ОПИСАНИЕ ТЕСТИРУЕМЫХ АЛГОРИТМОВ И ИХ ПРЕДВАРИТЕЛЬНЫЙ АНАЛИЗ

Мы реализовывали алгоритмы сортировки, у которых функция сравнения – сравнение чисел. Также некоторые алгоритмы сортировки мы сделали *устойчивыми*, т.е. порядок равных элементов при сортировке не меняется. Для этого необходимо, чтобы элементы для обмена не сравнивались на равенство.

Напомним, что мы проводили тестирование следующих алгоритмов:

- Встроенный алгоритм сортировки;
- Быстрая сортировка (сортировка Хоара);
- Сортировка слиянием;
- Сортировка кучей (пирамидальная сортировка);
- Сортировка посредством бинарных вставок;
- Сортировка посредством простых (линейных) вставок;
- Сортировка Бэтчера (Batcher);
- Сортировки Шелла (8 модификаций);
- Сортировка выбором;
- «Пузырьковая» сортировка.

Теперь приведем сложности каждого алгоритмов, а на некоторых особенностях реализации алгоритмов и самих алгоритмах остановимся более подробно.

1. Встроенный алгоритм сортировки.

За встроенный алгоритм сортировки будем понимать алгоритм, реализуемый встроенной функцией **sort**. В исходном коде perl5.8.8 можно найти, что алгоритм, реализуемый этой функцией – сортировка слиянием (mergesort). Вот отрывок этого кода (код приведен на языке **C**, файл с кодом: pp_sort, строка 1443):

```
/*
=head1 Array Manipulation Functions
=for apidoc sortsv
Sort an array. Here is an example:
    sortsv(AvARRAY(av), av_len(av)+1, Perl_sv_cmp_locale);
See lib/sort.pm for details about controlling the sorting algorithm.
=cut
*/
void
Perl_sortsv(pTHX_ SV **array, size_t nmemb, SVCOMPARE_t cmp)
{
    void (*sortsvp)(pTHX_ SV **array, size_t nmemb, SVCOMPARE_t cmp, U32 flags)
      = S_mergesortsv;
    dSORTHINTS;
    const I32 hints = SORTHINTS;
    if (hints & HINT_SORT_QUICKSORT) {
            sortsvp = S_qsortsv;
    }
    else {
            /* The default as of 5.8.0 is mergesort */
            sortsvp = S_mergesortsv;
    }
    sortsvp(aTHX_ array, nmemb, cmp, 0);
}
```

Красным цветом в коде выделены важные моменты кода. Видно, что по умолчанию вызывается сортировка слиянием. Если же определена **HINT_SORT_QUICKSORT**, то будет вызываться быстрая сортировка (сортиовка Хоара). Но мы тестировали встроенную сортировку слиянием.

Можно с большой долей уверенности сразу заявить, что этот алгоритм будет работать быстрее всех остальный из-за того, что код на языке **C** выполняется быстрее, чем код на языке **PERL**, т.к. код PERL еще нужно интерпретировать виртуальной машиной языка, а скомпилированный код **C** выполняется непосредственно.

1. Быстрая сортировка (сортировка Хоара).

В этом известном алгоритме сортировки в качестве ключа сравнения мы всегда брали центральный элемент сортируемого подмассива. Сложность работы этого алгоритма в среднем составляет $O(N \cdot \log N)$.

2. Сортировка слиянием.

В реализации этого алгоритма сортировки, как известно, используется функция слияния двух отсортированных массивов в один отсортированный. Для вставки элемента в конец массива мы будем использовать встроенную функцию *push*. Мы добавляем элемент в конец массива встроенной функцией из-за того, что она выполняется быстрее, чем если бы мы каждый раз вычисляли длину массива и присваивали нужному элементу конкретное значение. Сложность работы этого алгоритма составляет $O(N \cdot \log N)$.

3. Сортировка кучей (пирамидальная сортировка).

Пирамиду мы организовывали на основе массива – бинарное дерево мы не строили, т.к. это существенно скажется на производительности. Сложность работы этого алгоритма также составляет $O(N \cdot \log N)$.

4. Сортировка бинарными вставками.

5. Сортировка линейными вставками.

Реализации этих алгоритмов имеют следующую особенность. Сначала мы данные считываем в массив, а алгоритм сортировки последовательно считывает данные из полученного массива и записывает их в новый массив. Далее, для добавления нового элемента в нужную позицию мы воспользуемся встроенной функцией *splice*. Это также значительно ускорит выполнение алгоритма. Но какая при этом будет сложность у сортировки бинарными вставками: $O(N^2)$ или $O(N \cdot \log N)$– неясно, т.к. это целиком зависит от внутренней реализации *splice*, точнее от количества перезаписей в памяти. Сложность сортировки линейными (простыми) вставками при этом остается $O(N^2)$.

6. Сортировка Бэтчера.

Сортировка Бэтчера – это обменная сортировка со слиянием. Описание алгоритма можно найти в 3 томе «Искусства программирования» Кнута, глава 5.2, алгоритм M. Сложность этого алгоритма составляет всего $O((\log N)^2)$, но это только при параллельных вычислениях. При вычислениях на одной машине маленькую сложность сводит на нет процедура вычисления индексов сравниваемых элементов, которая может быть достаточно трудозатратой.

7. Сортировка Шелла.

Как было сказано, мы реализовали 8 модификаций этого алгоритма. Это было связано с тем фактом, что входными параметрами этой сортировки является так называемая последовательность смещений, согласно которому сравниваются ключи сортируемый данных. Мы увидим, что от выбора подходящей последовательности смещений можно добиться более лучших (или более худших) результатов.

Последовательность смещений обозначим $\{h_s\}$ и далее мы приведем все последовательности используемых нами смещений, а также для некоторых из них среднее время работы алгоритма[2].

Итак, модификации алгоритма сортировки Шелла:

Модификация 0

Используется самая простая последовательность смещений: $\{1\}$. При этом алгоритм будет достаточно длительное время – его сложность составит $O(N^2)$.

Модификация 1

Последовательность смещений используется такая: $\{2^{s+1} - 1 | s < \lfloor \log_2 N \rfloor\}$. При этом сложность алгоритма составит $O\left(N^{3/2}\right)$.

Модификация 2

Последовательность смещений используется такая: $\{2^s | s \leq \lfloor \log_2 N \rfloor\}$.

Модификация 3

В этой модификации используется последовательность, предложенная Инсерпи и Седгевиком, константа ρ=2,5. Опишем эту последовательность подробнее.

Строим сначала базовую последовательность $\{a_n\}$:

$$a_k = \min\{x \in \mathbb{N} \cap (\rho^k; +\infty) | \forall i \in [1; k): a_i \perp a_k\}$$

После этого строится последовательность смещений таким образом:

$$h_s = h_{s-r} a_r$$
$$\binom{r}{2} < s \leq \binom{r+1}{2}$$

[2] Согласно 3 тому «Искусства программирования» Кнута, где достаточно хорошо исследован этот алгоритм сортировки.

При этом сложность алгоритма составит $O\left(Ne^{\sqrt{8\ln\rho\ln N}}\right)$, т.е. в худшем случае порядок будет значительно лучше, чем $O\left(N^{3/2}\right)$.

Модификация 4

Эта модификация подобна предыдущей модификации алгоритма, но константа ρ принимается равной 2.

Стоит сказать, что эти последовательности мы предварительно получали, написав предварительно программу на языке **Java** и непосредственно занеся результаты вычислений в массив смещений. В каждой последовательности были получены все элементы, которые не превосходили 15000000000 ($15\cdot10^9$).

Модификация 5

Используется такая последовательность смещений: $h_s = \begin{cases} 1, s = 0, \\ 3h_{s-1} + 1, s \neq 0. \end{cases}$

Модификация 6

Используется последовательность смещений Седгевика, которая выглядит следующим образом: $h_s = \begin{cases} 9 \cdot \left(2^s - 2^{s/2}\right) + 1, s = 2k, \\ 8 \cdot 2^s - 6 \cdot 2^{\frac{s+1}{2}} + 1, s = 2k - 1. \end{cases}$. Сложность алгоритма при этом в худшем случае составляет $O\left(N^{4/3}\right)$. Эту последовательность мы также предварительно рассчитали, написав программу на языке **Java**. Все рассчитанные элементы не превосходят 17000000000 ($17\cdot10^9$).

Модификация 7

Используется последовательность Пратта, которая выглядит следующим образом: $h_s = \{2^p 3^q | p \leq \lceil \log_2 N \rceil, q \leq \lceil \log_3 N \rceil\}$. Сложность алгоритма в этом случае будет порядка $O(N \cdot (\log N)^2)$.

8. **Сортировка выбором.**
9. **«Пузырьковая» сортировка.**

Реализации этих алгоритмов имеют сложность $O(N^2)$.

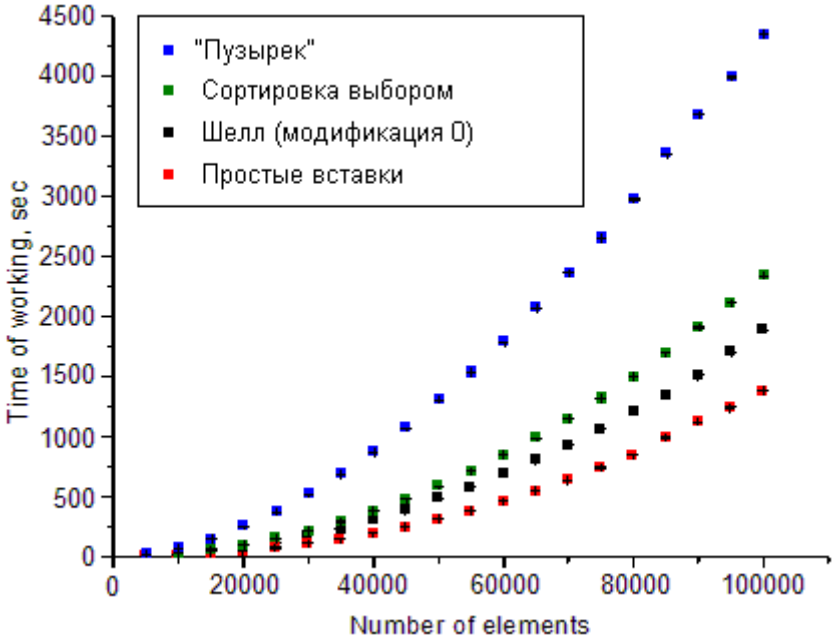
V. ХОД ТЕСТИРОВАНИЯ И ПРЕДВАРИТЕЛЬНЫЕ РЕЗУЛЬТАТЫ

После того, как все тестирующие сценарии **PERL** завершили работу, мы сначала преобразовали отчеты о времени работы алгоритмов, вычислив соответствующее среднее арифметическое и создав тем самым новый отчет.

При тестировании с подсчетом операций над массивами мы собирали следующие данные: количество записей в массив, количество чтений из массива и количество сравнений. Данных о количестве перестановок мы не собирали, т.к. одна перестановка элементов – это, по сути, 4 операции: 2 записи в массив и 2 чтения из массива. Поэтому, на наш взгляд, количество операций чтения-записи нам даст более общую картину того, сколько операций выполнилось над массивом.

На следующей странице приведена таблица зависимости времени работы от количества сортируемых элементов различных алгоритмов.

Из приведенной таблицы можно заметить, что следующие алгоритмы сортировки: Шелл (модификация 0), Простые вставки, «Пузырек», Сортировка выбором работают очень долго по сравнению с остальными сортировками. На графике зависимость времени работы этих алгоритмов от количества сортируемых элементов выглядит следующим образом:



Из графика видно, что зависимость времени работы от числа сортируемых элементов у этих алгоритмов нелинейно; кроме того, алгоритм сортировки «Пузырек» работает более чем в 2 раза дольше всех остальных.

Теперь рассмотрим работу остальных алгоритмов. Начнем с алгоритма бинарных вставок. График зависимости времени работы у него выглядит следующим образом:

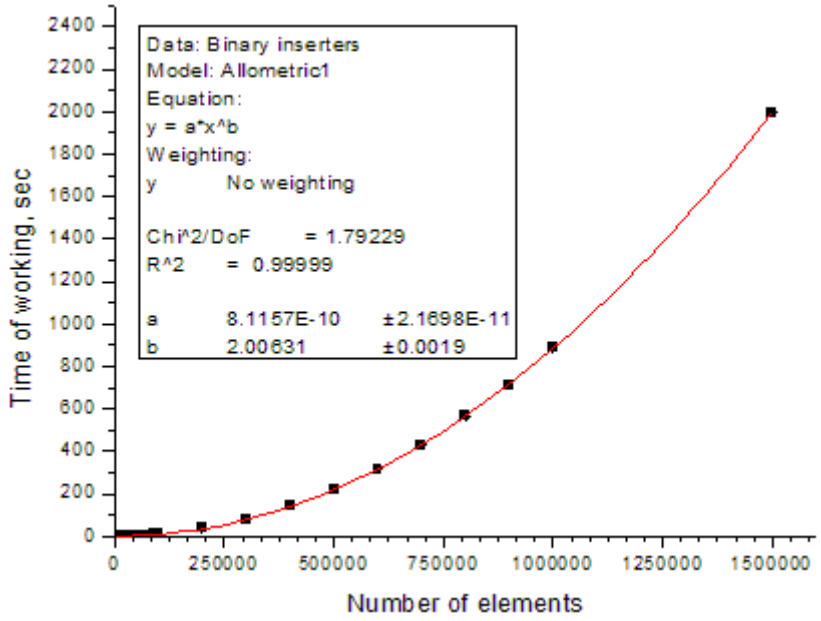


Таблица 1

Количество сортируемых элементов	Название алгоритма сортировки																
	Внутренняя	Хоара	Бинарные вставки	Сортировка кучей	Бэтчер	Шелл, модиф. 0	Шелл, модиф. 1	Шелл, модиф. 2	Шелл, модиф. 3	Шелл, модиф. 4	Шелл, модиф. 5	Шелл, модиф. 6	Шелл, модиф. 7	Простые вставки	«Пузырек»	Сортировка выбором	Сортировка слиянием
5000	0	0	0	0	0	5	1	0	0	0	0	0	0	2	16	5	0
10000	0	0	0	1	1	19	0	0	0	1	0	0	1	11	63	23	1
15000	0	0	1	1	1	45	0	0	0	0	0	1	1	27	139	52	1
20000	0	1	1	1	2	75	0	1	1	1	0	0	2	47	249	93	1
25000	0	0	1	1	2	117	1	2	0	0	0	1	2	73	364	145	2
30000	0	1	1	1	3	169	1	3	0	0	1	1	2	107	514	210	2
35000	0	0	1	2	3	227	1	3	1	1	1	1	3	147	679	286	2
40000	0	1	2	3	4	300	1	3	1	1	1	1	4	194	862	371	2
45000	0	1	2	2	5	382	1	5	2	1	1	1	4	247	1065	471	2
50000	0	1	3	2	5	483	2	5	1	1	1	1	4	311	1298	582	3
55000	1	1	4	3	5	576	2	5	1	2	1	2	5	378	1528	703	3
60000	0	1	4	3	5	680	2	4	1	1	2	1	5	459	1778	840	3
65000	0	1	5	4	6	796	2	6	2	1	2	2	6	545	2063	984	4
70000	1	1	6	4	7	921	2	10	2	1	2	2	6	635	2359	1141	3
75000	1	1	7	5	8	1060	2	10	2	2	2	2	7	739	2645	1311	4
80000	1	2	7	5	9	1205	3	11	2	2	3	2	7	847	2969	1490	3
85000	1	2	7	5	9	1337	3	12	2	2	3	2	8	990	3349	1692	4
90000	1	2	9	5	10	1501	3	12	2	2	3	2	8	1121	3672	1901	4
95000	1	2	9	6	10	1700	3	12	2	3	3	2	9	1238	3985	2104	4
100000	1	2	10	7	11	1883	3	13	3	3	3	3	10	1379	4341	2330	5
200000	2	4	37	13	24		7	32	5	6	7	5	22				9
300000	3	6	82	19	39		13	59	9	9	12	8	35				15
400000	3	9	143	27	54		17	117	12	12	16	11	49				20

500000	4	11	221	35	68		22	132	15	16	22	14	63				26
600000	6	13	317	42	84		27	179	18	19	25	16	78				32
700000	6	15	432	51	99		33	205	21	23	34	20	94				37
800000	8	18	566	58	117		39	235	25	25	40	22	109				43
900000	9	20	714	66	134		47	334	28	29	46	26	125				47
1000000	10	23	886	74	148		53	353	32	33	52	28	141				54
1500000	15	34	1998	115	235		92	784	48	51	83	44	224				83

На приведенном графике, построенном в программе Origin 7.5, красная линия – наилучшее приближение данных к кривой с уравнением $y = ax^b$, причем программа вычислила уравнение кривой с погрешностями (коэффициенты *a* и *b* приведены на графике в рамочке). Коэффициент, описывающий отклонение графика от точек приблизительно равен единице (он равен 0,99999), то есть график почти не отклоняется от рассчитанной кривой. Можно заметить, что коэффициент *b* равен 2 (в пределах погрешности), а значит, алгоритм работает квадратичное время! Но он все же работает быстрее, чем предыдущие четыре, т.к. константа у этого алгоритма меньше.

Приведем рассчитанные аналогичным образом зависимости у четырех предыдущих алгоритмов (графики сейчас мы приводить не будем – они будут приведены в приложении 1):

Таблица 2

Алгоритм сортировки	**Fitting equation**[3]	**Коэффициенты (без погрешностей)**		R^2**(отклонение)**
		a	*b*	
«Пузырек»	$y = ax^b$	6,8909E-6	1,76078	0,99978
Сортировка выбором	$y = ax^b$	2,2354E-7	2,00379	0,99998
Сортировка Шелла (0)	$y = ax^b$	2,1239E-7	1,98906	0,99984
Простые вставки	$y = ax^b$	2,2276E-8	2,15881	0,99978
Бинарные вставки	$y = ax^b$	8,1157E-10	2,00631	0,99999

Теперь сравним между собой сортировки Шелла (модификации 1-7).

Из таблицы можно увидеть, что модификации сортировки Шелла расположены в порядке возрастания быстродействия следующим образом:

[3] Аналитическое уравнение кривой, к которой производилось приближение

Таблица 3

Место	Модификация алгоритма	Время работы (при 1500000 элементов)
1	6	44
2	3	48
3	4	51
4	5	83
5	1	92
6	7	224
7	2	784
8	0	не измерялось

Если вспомнить, что последовательность смещений, используемая в модификации 7 предполагает сложность алгоритма $O(N \cdot (\log N)^2)$, то может показаться несколько странным, что этот алгоритм занимает только шестое место в нашей «рейтинговой» таблице. Все дело в том, что все быстродействие сводится на нет большим количеством проходов по массиву – например, если количество сортируемых элементов 1500000, то алгоритм совершает 231 проход по массиву. Теперь приведем зависимости времени работы алгоритма от количества элементов, полученные так же, как и в предыдущих рассмотренных алгоритмах:

Таблица 4

Номер модификации алгоритма	Fitting equation	Коэффициенты (без погрешностей)		R^2(отклонение)
		a	*b*	
6	$y = ax^b$	0,00001	1,05681	0,99881
3	$y = ax^b$	0,00001	1,07087	0,99884
4	$y = ax^b$	0,00001	1,08604	0,99869
5	$y = ax^b$	2,6581E-6	1,21411	0,99889
1	$y = ax^b$	8,9293E-7	1,29663	0,99893
7	$y = ax^b$	0,00002	1,15239	0,99995
2	$y = ax^b$	2,6206E-8	1,69448	0,99446
0	$y = ax^b$	2,1239E-7	1,98906	0,99984

Модификации алгоритма мы расположили согласно «рейтинговой» таблице (таблица 3), приведенной выше.

Из приведенной таблицы видно, что коэффициент *b* убывает по мере спуска по таблице вниз, исключение составляет только седьмая модификация алгоритма – она, по сути, должна располагаться до пятой модификации. Причина этого, как уже говорилось, заключается в большом количестве проходов по массиву. Также можно заметить, что шестая модификация (последовательность Седгевика) работает почти линейное время.

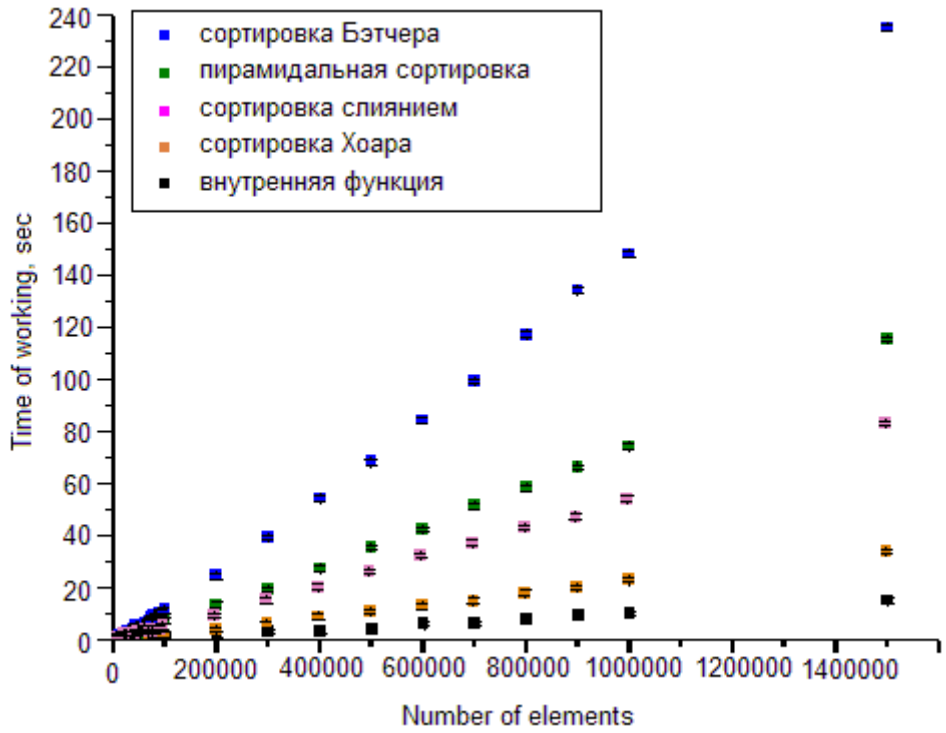
Теперь сравним между собой оставшиеся алгоритмы сортировки: внутренняя функция, быстрая сортировка, пирамидальная сортировка, сортировка Бэтчера, сортировка слиянием.

Расположим эти сортировки в аналогичной «рейтинговой» таблице:

Таблица 5

Место	Название алгоритма	Время работы (при 1500000 элементов)
1	Внутренняя функция	15
2	Сортировка Хоара	34
3	Сортировка слиянием	83
4	Пирамидальная сортировка	115
5	Сортировка Бэтчера	235

Из этой и предыдущих таблиц видно, что лидирует по быстродействию внутренняя функция сортировки, что не удивительно – ведь сортируемый код написан на языке **C** и скомпилирован, а не на языке **PERL**, код которого нужно интерпретировать и возникают затраты на интерпретацию кода. Построим графики зависимости времени сортировки от числа сортируемых элементов и наложим их на один график:



Из приведенных графиков видно, что все 5 алгоритмов не имеют на первый взгляд какой-либо ярко выраженной зависимости времени работы от числа элементов, кроме линейной. Рассчитаем, как это уже делалось, коэффициенты приближения (*a* и *b*):

Таблица 6

Название алгоритма	Fitting equation	Коэффициенты (без погрешностей)		R^2(отклонение)
		a	*b*	
Внутренняя функция	$y = ax^b$	3,3148E-6	1,07821	0,99037
Сортировка Хоара	$y = ax^b$	0,00001	1,04571	0,99843
Сортировка слиянием	$y = ax^b$	0,00002	1,08949	0,99973
Пирамидальная сортировка	$y = ax^b$	0,00003	1,12513	0,99988
Сортировка Бэтчера	$y = ax^b$	0,00002	1,05711	0,99932

Из таблицы видно, что коэффициент *b* у внутренней функции сортировки практически совпадает с соответствующем коэффициентом у сортировки слиянием, разница лишь в константах – в 6 раз. Так как (мы это уже выяснили из исходных кодов) внутренняя функция использует сортировку слиянием, то: во-первых, наше предположение подтвердилось (то есть среди десятков тысяч строк кода **C** мы нашли десяток нужных), во-вторых, коэффициент отражает то, что внутренняя функция выполняется всегда приблизительно шесть раз быстрее, чем аналогичная, но написанная на **PERL**. Далее, заявленная алгоритмическая сложность сортировки Бэтчера – всего $O((\log N)^2)$ сводится на нет трудозатратной процедурой вычисления индексов сравниваемых элементов. При параллельных вычислениях на машинах с общей памятью можно получить такую сложность, так как сравниваемые пары не пересекаются. А на одной ЭВМ с одним процессором такого, к сожалению, достичь не получилось.

Итак, подведем промежуточный итог – приведем таблицу, отражающую быстродействие рассмотренных алгоритмов сортировки:

Таблица 7

№ п/п	Название алгоритма сортировки
1	Внутренняя функция
2	Сортировка Хоара
3	Шелл, модификация 6
4	Шелл, модификация 3
5	Шелл, модификация 4
6	Шелл, модификация 5
7	Сортировка слиянием
8	Шелл, модификация 1
9	Пирамидальная сортировка
10	Шелл, модиикация 7
11	Сортировка Бэтчера
12	Шелл, модификация 2
13	Бинарные вставки
14	Простые вставки
15	Шелл, модификация 0
16	Сортировка выбором
17	«Пузырек»

Теперь проанализируем отчеты о количестве операций над массивами, выполненных при каждом алгоритме сортировки. Для облегчения анализа результатов (нам нужно проанализировать 51 график с данными, что достаточно трудно сделать) мы проанализируем лишь некоторые полученные данные.

Сначала мы приведем таблицу, в которой приведено количество операций над массивом при сортировке 10000 элементов:

Таблица 8

Название алгоритма	Количество операций			
	Чтение из массива	Запись в массив	Сравнение элементов	Итого
Внутренняя функция	Не фиксировалось		120957	120957
Сортировка Хоара	215023	67694	138503	421220
Шелл - 6	401384	200692	200692	802768
Шелл - 3	423164	211582	211582	846328
Шелл - 4	445424	222712	222712	890848
Шелл - 5	457164	228582	228582	914328
Сортировка слиянием	520104	272640	123732	916476
Шелл - 1	498172	249086	249086	996344
Пирамидальная сорт.	832164	248308	291928	1372400
Шелл - 7	1227136	613568	613568	2454272
Сортировка Бэтчера	1111696	260306	425695	1797697
Шелл - 2	1559130	779565	779565	3118260
Бинарные вставки	247998	20000	118999	386997
Простые вставки	49999358	20000	24984688	75004046
Шелл - 0	50037590	25018795	25018795	100075180
Сортировка выбором	100010000	20000	49995000	150025000
«Пузырек»	145336190	45326190	50005000	240667380

Алгоритмы расположены в таблице в порядке возрастания времени работы. Из приведенной таблицы видно, что время растет с возрастанием количества операций над массивом, исключение составляет только сортировка Бэтчера и бинарные вставки. Если вспомнить, сколько работает этот

алгоритм и сколько работает 7-ая модификация сортировки Шелла, то можно вычислить, что алгоритм Шелла работает примерно на 10 секунд быстрее, выполняя при этом большее число операций. Такое поведение можно снова объяснить затратами на вычисление последовательности сравнений. Если же рассмотреть алгоритм бинарных вставок, то можно выдвинуть единственное предположение о том, почему алгоритм долго работает – при вставке элемента в список функции *splice* нужно сдвинуть некоторое количество элементов массива, то есть внутренний код (код на **C**) реорганизовывает память для вставки – эту реорганизацию мы не «отловили». По сути, происходит много перезаписей элементов. Если их учесть, то количество операций существенно возрастет. А в остальном получился достаточно ожидаемый результат.

Еще в таблице можно заметить следующую особенность: внутренняя функция сортировки производит меньшее число сравнений элементов, чем сортировка слиянием. Это можно объяснить какими-то оптимизациями реализации алгоритма. Меньше всего сравнений произвела сортировка с помощью бинарных вставок, но количество сравнений (даже если операция сравнения очень трудоемкая) сводится на нет процедурой реорганизации памяти. Также можно заметить, что нулевая модификация алгоритма Шелла – по сути, алгоритм простых вставок (если сравнивать количество чтений и сравнений). А последнее место в нашей таблице занимает алгоритм «Пузырьковой» сортировки – он самый неэффективный.

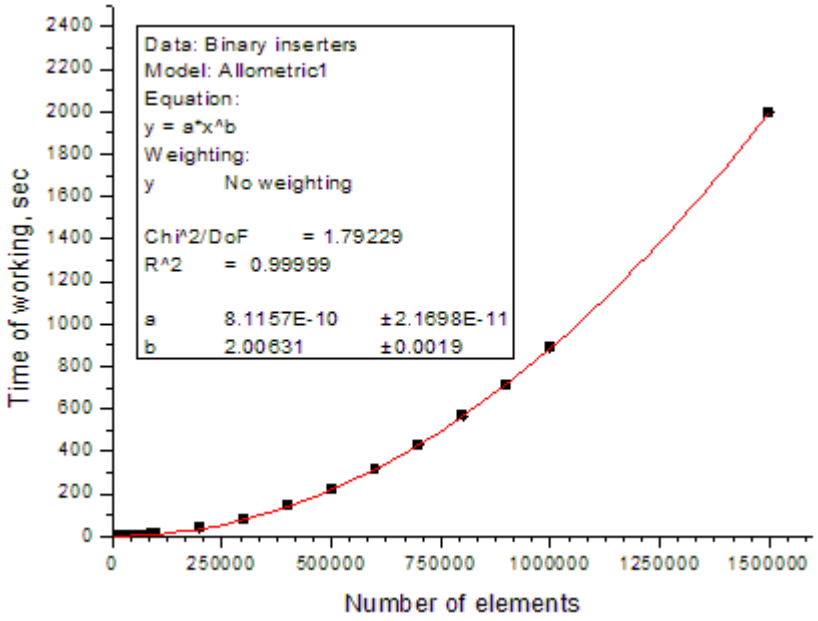
Данные о зависимости количества операций над массивами можно посмотреть в файле `op.xls`, который поставляется вместе с отчетом. Первый столбец в алгоритме содержит информацию о количестве чтений из массива, второй – о количестве записей в массив, третий – количество сравнений элементов.

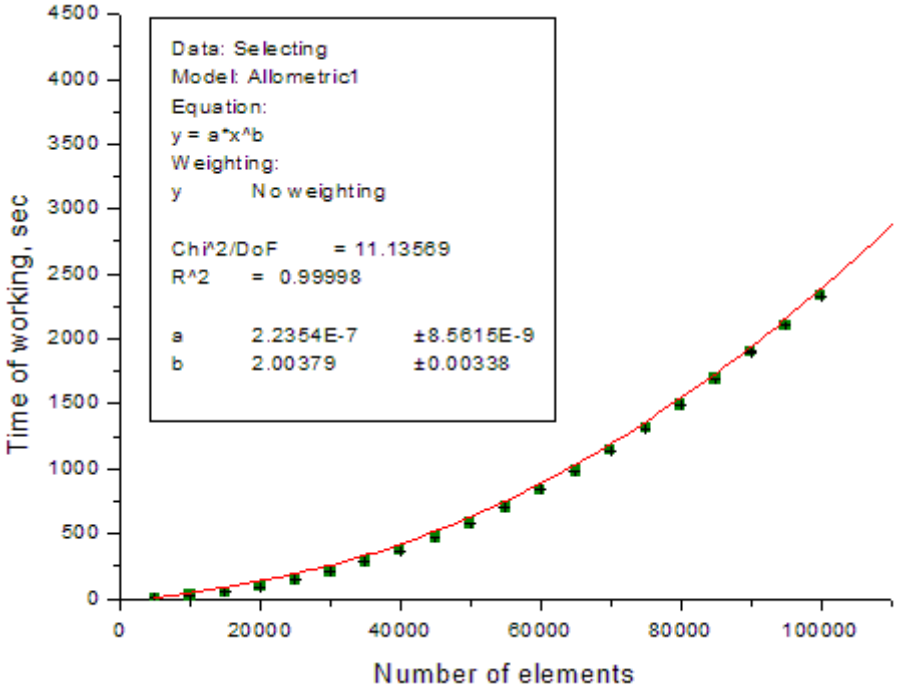
Теперь проанализируем *устойчивость* сортировок. Как уже говорилось, сортировка называется устойчивой, если она не меняет порядок равных элементов. (Но если мы сортируем элементы по убыванию, то порядок равных элементов инвертируется). Так как мы реализовали алгоритмы, сортирующие массив с числами (это зависит от функции сравнения элементов – мы использовали сравнение чисел), то можно предложить следующий способ проверки сортировки на устойчивость. Чтобы различать равные элементы, мы в их конец допишем символы латинского алфавита. При этом функция сравнения будет сравнивать только число, которое идет в начале, а буквы проигнорирует. Тем самым мы можем отслеживать порядок равных элементов. Для анализа устойчивости мы написали файл, который содержит только равные элементы и ничего более, запускали сортировку и на выходе мы должны были получить идентичный файл. Результаты такого тестирования следующие:

Таблица 9

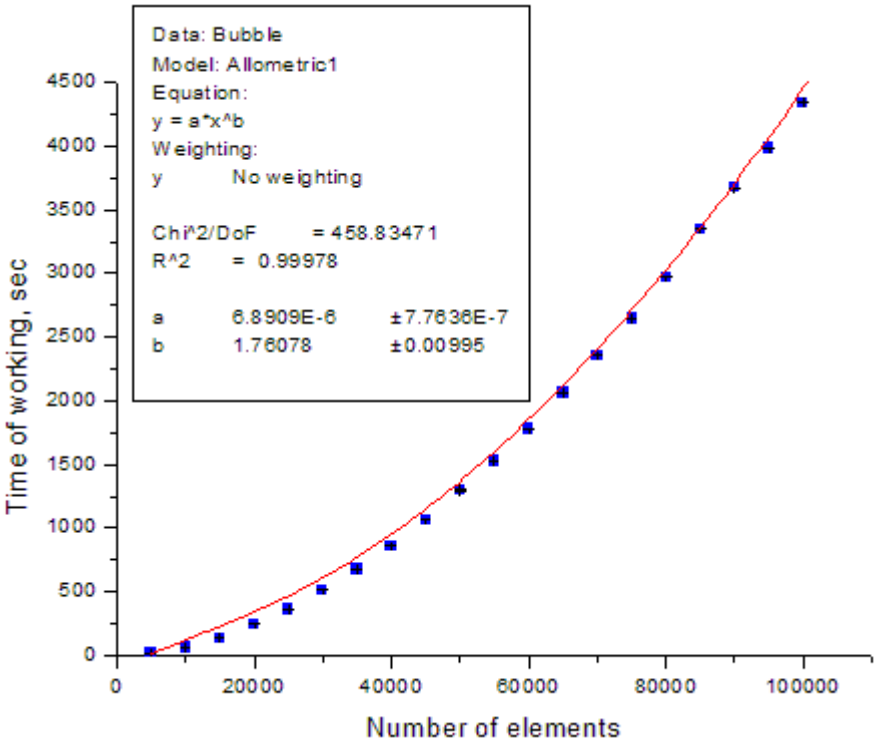
Алгоритм сортировки	Устойчив?
Внутренняя функция	Да
Сортировка Хоара	Да
Шелл - 6	Нет
Шелл - 3	Нет
Шелл - 4	Нет
Шелл - 5	Нет
Сортировка слиянием	Да
Шелл - 1	Нет
Пирамидальная сортировка	Нет
Шелл - 7	Нет
Сортировка Бэтчера	Нет
Шелл - 2	Нет
Бинарные вставки	Да
Простые вставки	Да
Шелл - 0	Да
Сортировка выбором	Да
«Пузырек»	Да

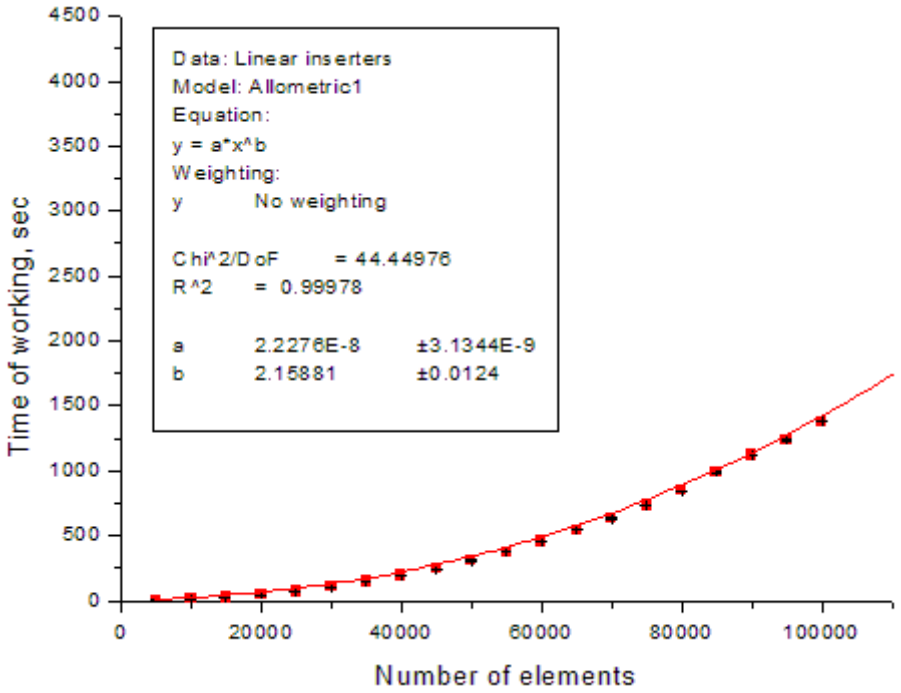
Приложение 1. Графики результатов тестирования



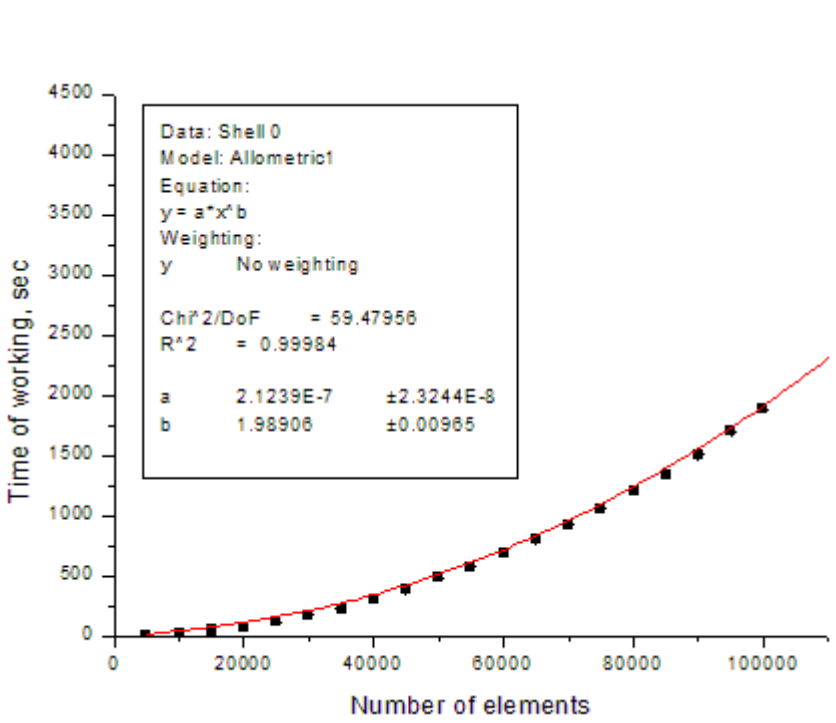


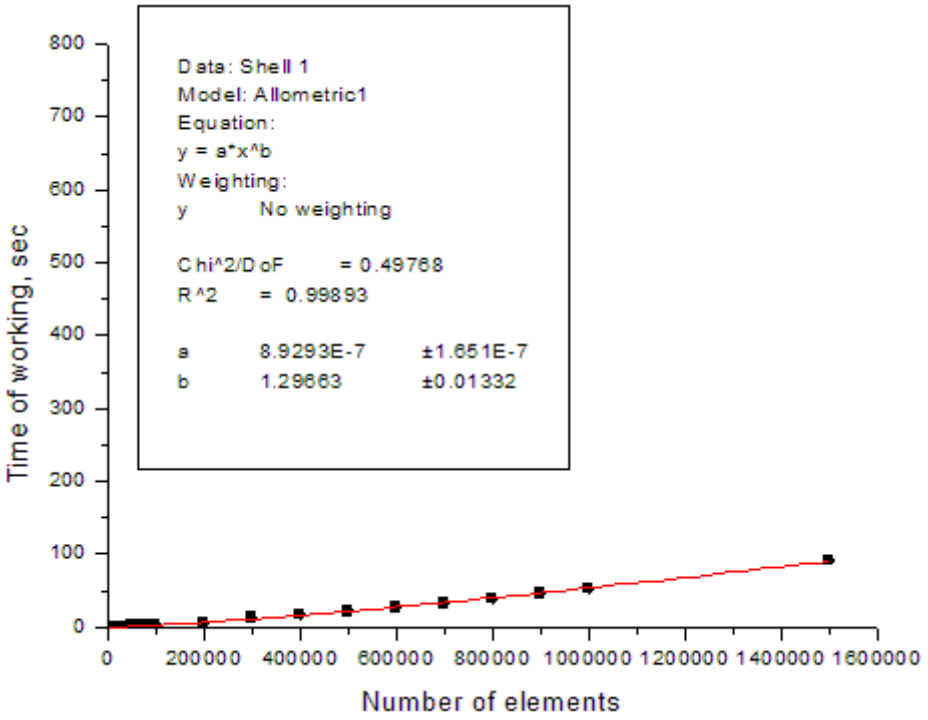
Приложение 1. Графики результатов тестирования



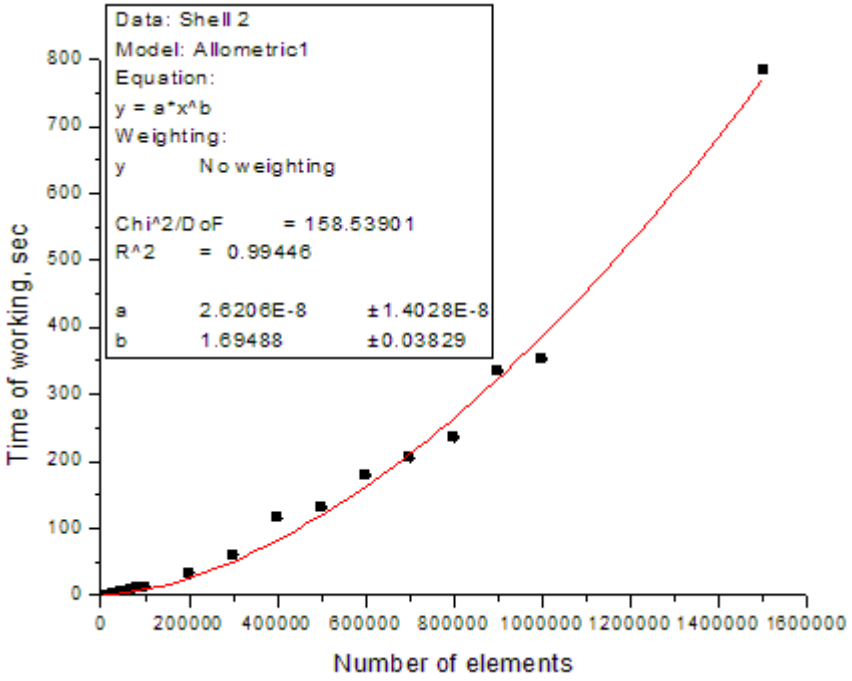


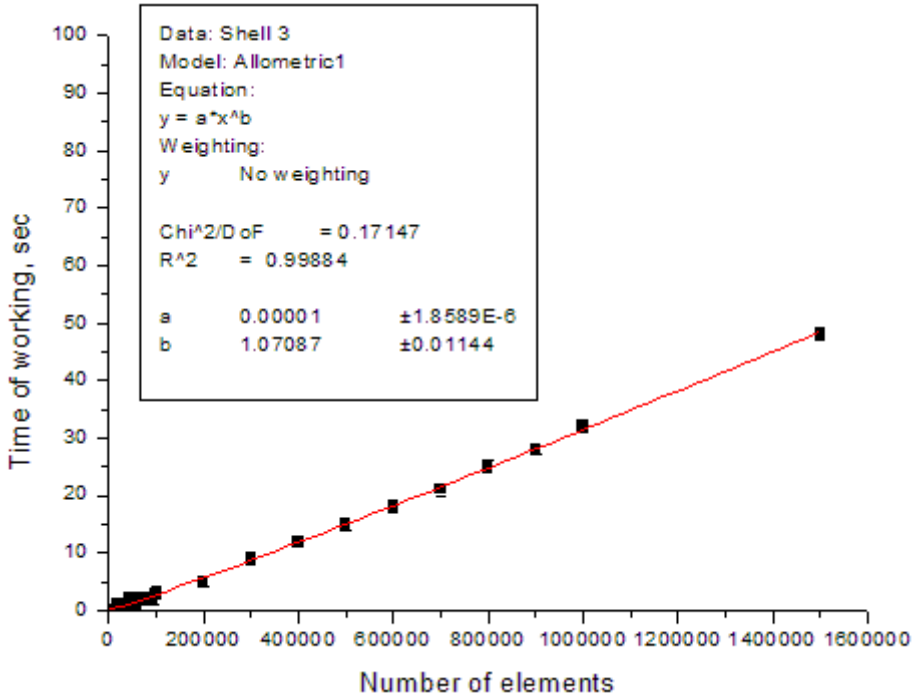
Приложение 1. Графики результатов тестирования



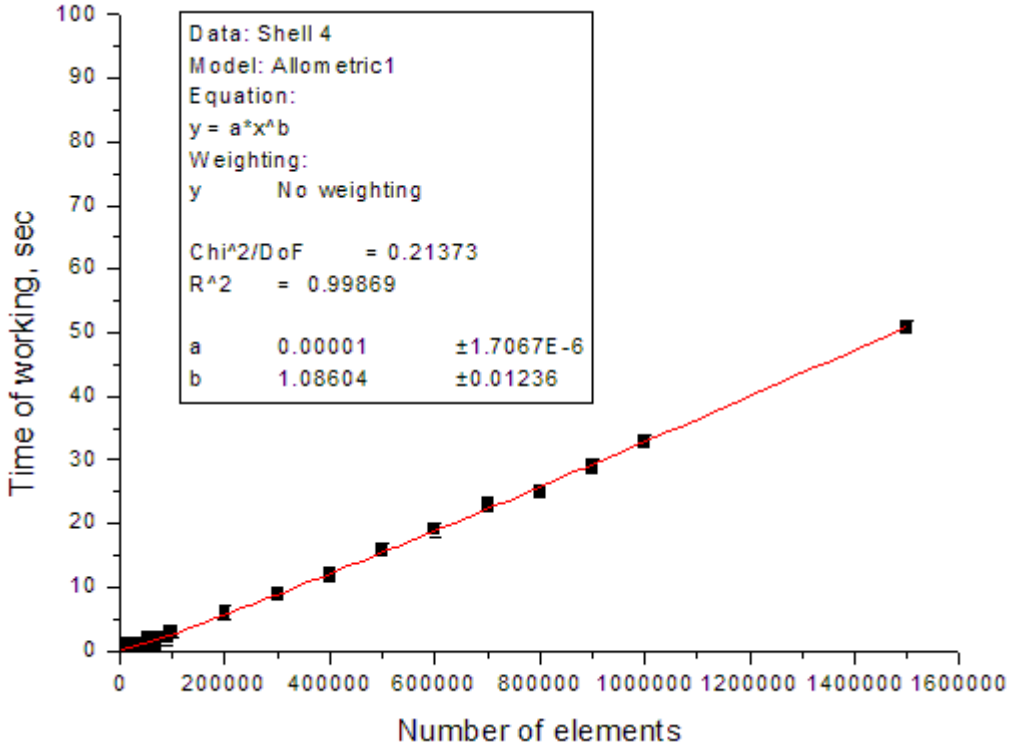


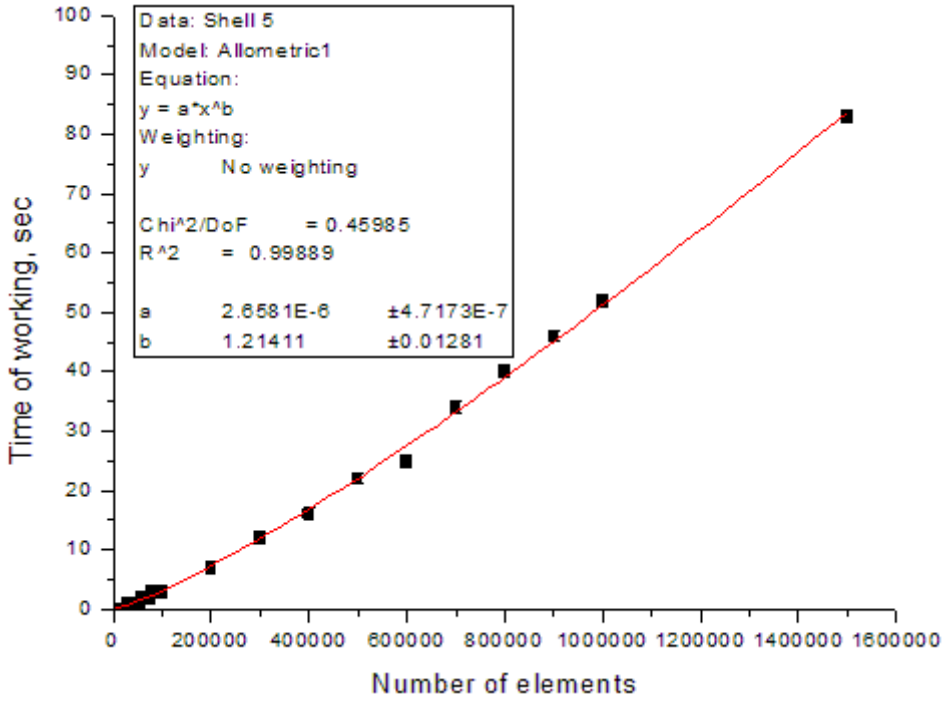
Приложение 1. Графики результатов тестирования



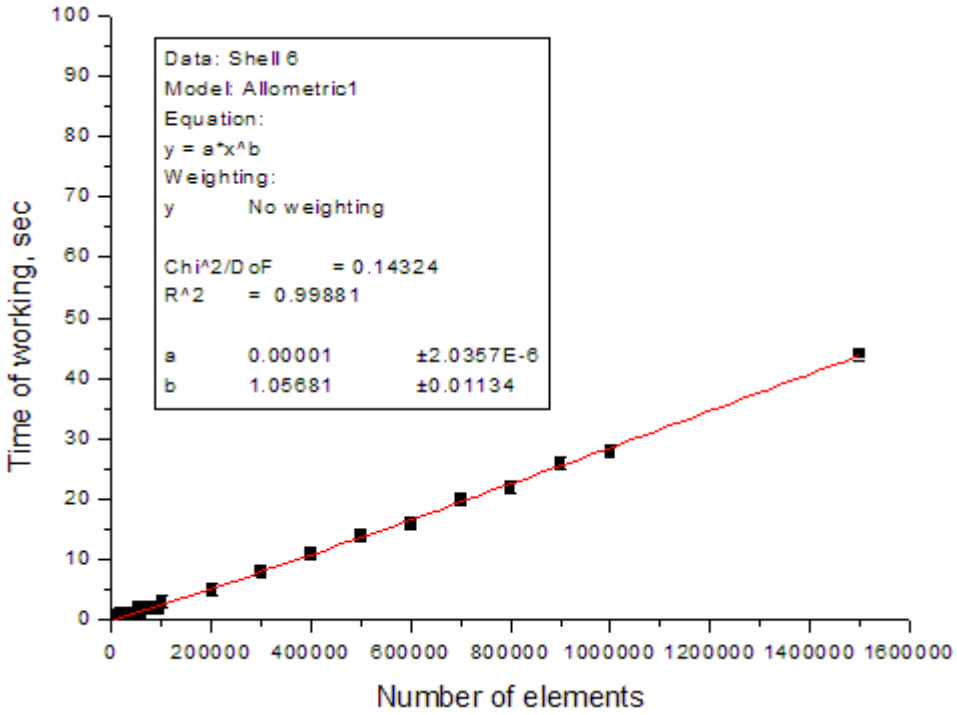


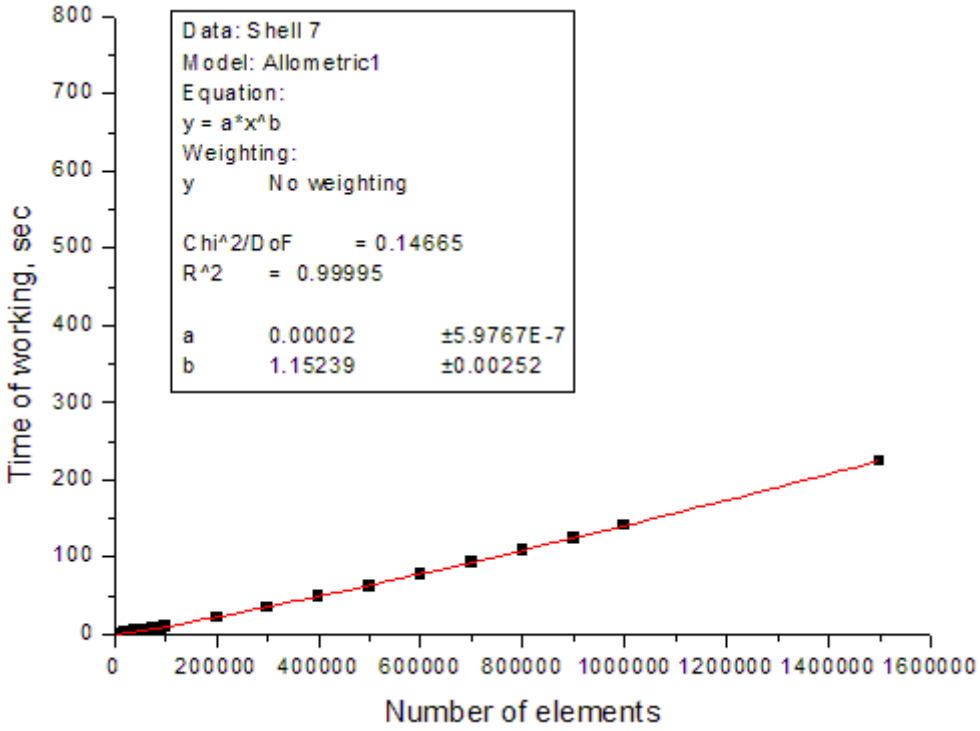
Приложение 1. Графики результатов тестирования



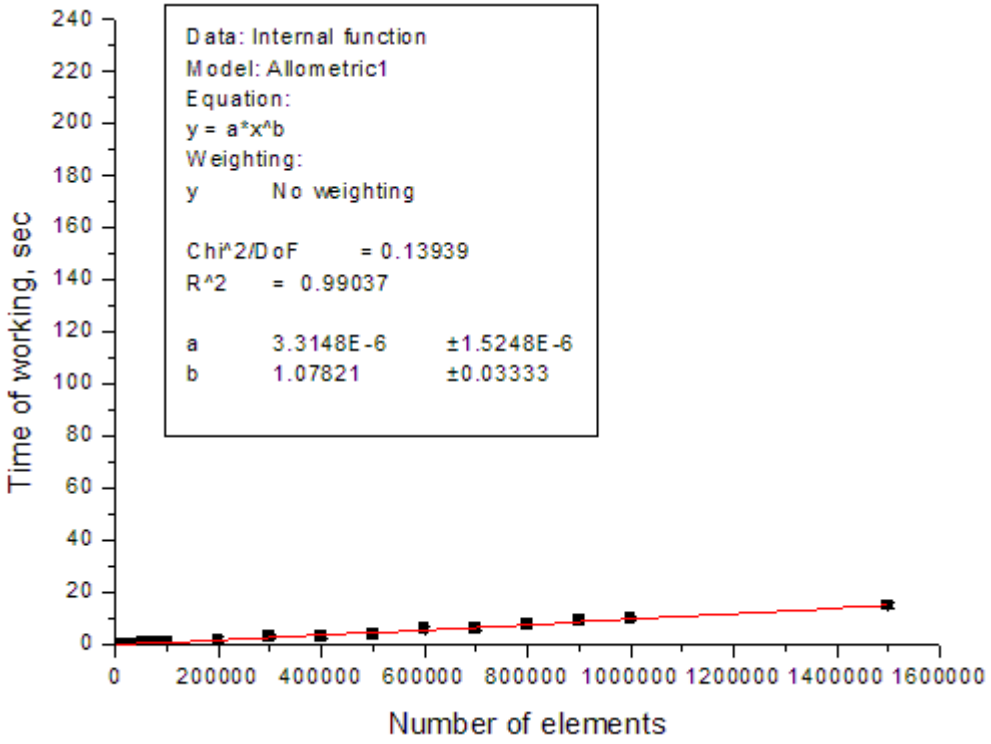


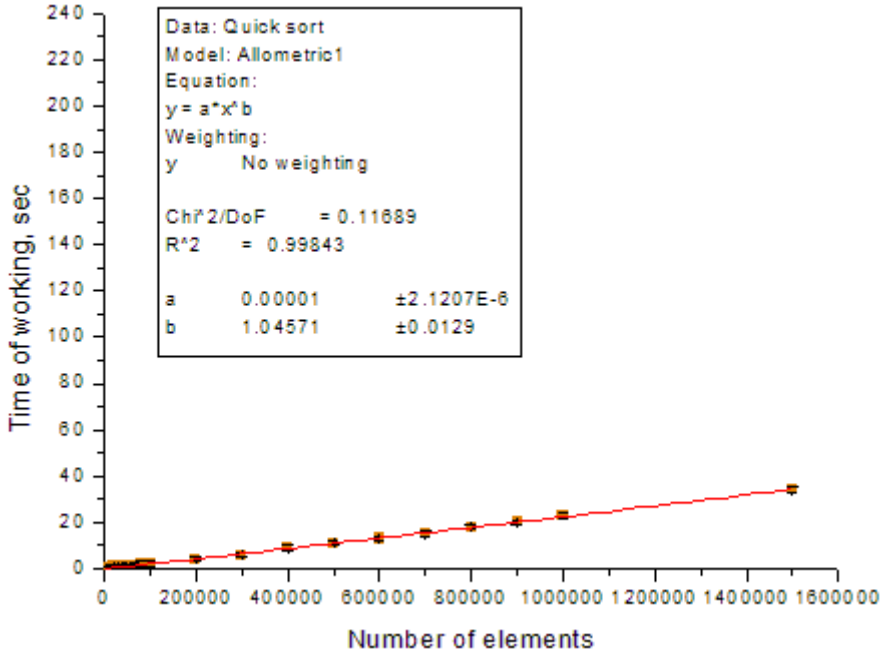
Приложение 1. Графики результатов тестирования



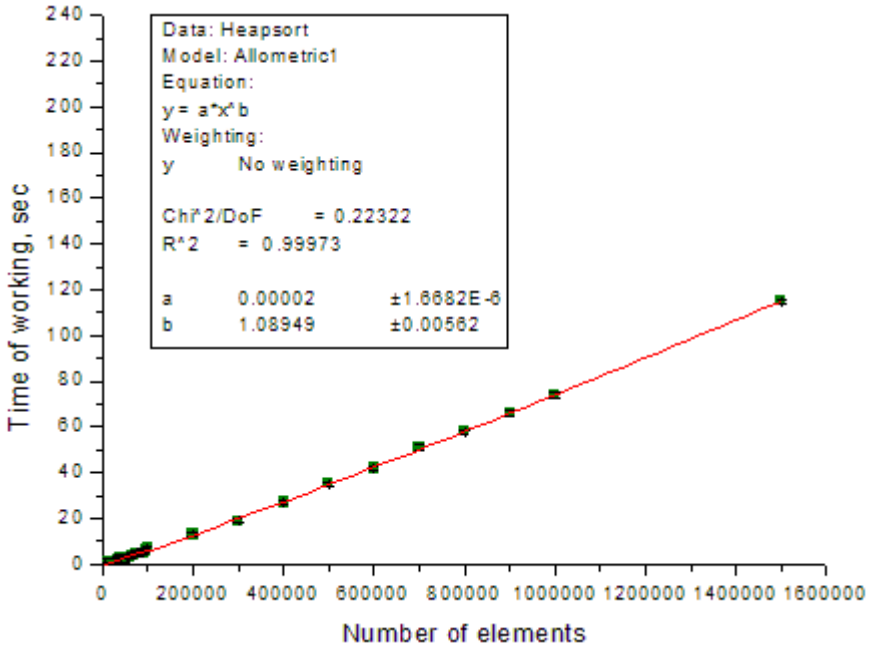


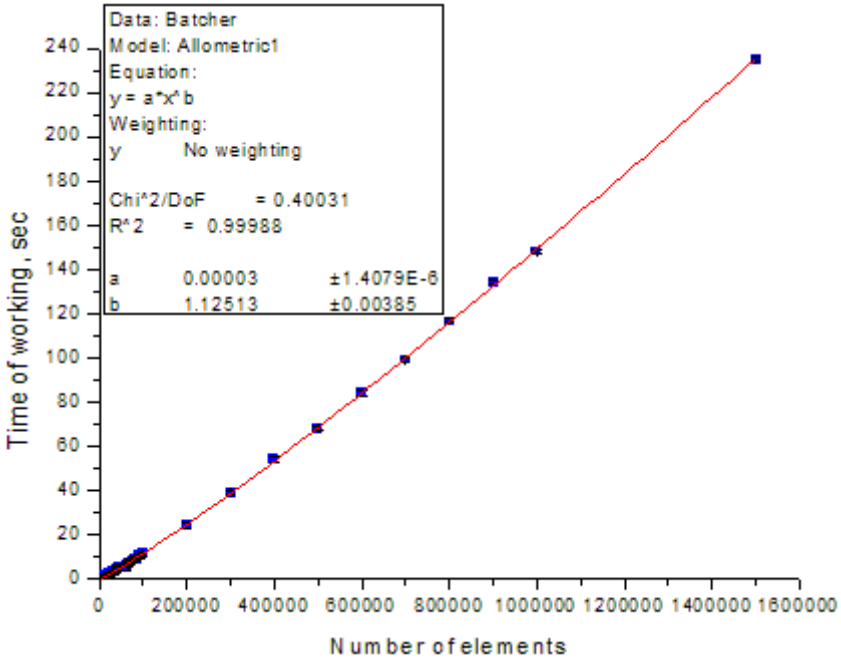
Приложение 1. Графики результатов тестирования





Приложение 1. Графики результатов тестирования





Приложение 1. Графики результатов тестирования