OMS7 сегодня в 16:54 Разработка

Описание алгоритмов сортировки и сравнение их производительнос

Алгоритмы, C++

Из песочницы

Вступление

На эту тему написано уже немало статей. Однако я еще не видел статьи, в которой сравниваются все основные сортировки на большом чи тестов разного типа и размера. Кроме того, далеко не везде выложены реализации и описание набора тестов. Это приводит к тому, что мо возникнуть сомнения в правильности исследования. Однако цель моей работы состоит не только в том, чтобы определить, какие сортиров работают быстрее всего (в целом это и так известно). В первую очередь мне было интересно исследовать алгоритмы, оптимизировать их, они работали как можно быстрее. Работая над этим, мне удалось придумать эффективную формулу для сортировки Шелла.

Во многом статья посвящена тому, как написать все алгоритмы и протестировать их. Если говорить о самом программировании, то иногда возникнуть совершенно неожиданные трудности (во многом благодаря оптимизатору C++). Однако не менее трудно решить, какие именно в каких количествах нужно сделать. Коды всех алгоритмов, которые выложены в данной статье, написаны мной. Доступны и результаты за на всех тестах. Единственное, что я не могу показать — это сами тесты, поскольку они весят почти 140 ГБ. При малейшем подозрении я пр и код, соответствующий тесту, и сам тест. Надеюсь, что статья Вам понравится.

Описание основных сортировок и их реализация

Я постараюсь кратно и понятно описать сортировки и указать асимптотику, хотя последнее в рамках данной статьи не очень важно (интере узнать реальное время работы). О потреблении памяти в дальнейшем ничего писать не буду, замечу только, что сортировки, использующи непростые структуры данных (как, например, сортировка деревом), обычно потребляют ее в больших количествах, а остальные сортировк худшем случае только создают вспомогательный массив. Также существует понятие стабильности (устойчивости) сортировки. Это значит, относительный порядок элементов при их равенстве не меняется. Это тоже в рамках данной статьи неважно (в конце концов, можно прост прицепить к элементу его индекс), однако в одном месте пригодится.

Сортировка пузырьком / Bubble sort

Будем идти по массиву слева направо. Если текущий элемент больше следующего, меняем их местами. Делаем так, пока массив не будет отсортирован. Заметим, что после первой итерации самый большой элемент будет находиться в конце массива, на правильном месте. Пос двух итераций на правильном месте будут стоять два наибольших элемента, и так далее. Очевидно, не более чем после n итераций масси отсортирован. Таким образом, асимптотика в худшем и среднем случае – O(n^2), в лучшем случае – O(n).

Реализация:

```
void bubblesort(int* l, int* r) {
	int sz = r - l;
	if (sz <= 1) return;
	bool b = true;
	while (b) {
		b = false;
		for (int* i = l; i + 1 < r; i++) {
			if (*i > *(i + 1)) {
				swap(*i, *(i + 1));
				b = true;
			}
		}
		r--;
	}
}
```

Шейкерная сортировка / Shaker sort

(также известна как сортировка перемешиванием и коктейльная сортировка). Заметим, что сортировка пузырьком работает медленно на те которых маленькие элементы стоят в конце (их еще называют «черепахами»). Такой элемент на каждом шаге алгоритма будет сдвигаться на одну позицию влево. Поэтому будем идти не только слева направо, но и справа налево. Будем поддерживать два указателя begin и end, обозначающих, какой отрезок массива еще не отсортирован. На очередной итерации при достижении end вычитаем из него единицу и дви: справа налево, аналогично, при достижении begin прибавляем единицу и двигаемся слева направо. Асимптотика у алгоритма такая же, ка сортировки пузырьком, однако реальное время работы лучше.

Реализация:

```
void shakersort(int* l, int* r) {
	int sz = r - l;
	if (sz <= 1) return;
	bool b = true;
	int* beg = l - 1;
	int* end = r - 1;
	while (b) {
		b = false;
		beg++;
		for (int* i = beg; i < end; i++) {
			if (*i > *(i + 1)) {
				swap(*i, *(i + 1));
				b = true;
			}
		}
		if (!b) break;
		end--;
		for (int* i = end; i > beg; i--) {
			if (*i < *(i - 1)) {
				swap(*i, *(i - 1));
				b = true;
			}
		}
	}
}
```

Сортировка расческой / Comb sort

Еще одна модификация сортировки пузырьком. Для того, чтобы избавиться от «черепах», будем переставлять элементы, стоящие на расс Зафиксируем его и будем идти слева направо, сравнивая элементы, стоящие на этом расстоянии, переставляя их, если необходимо. Очев это позволит «черепахам» быстро добраться в начало массива. Оптимально изначально взять расстояние равным длине массива, а далее делить его на некоторый коэффициент, равный примерно 1.247. Когда расстояние станет равно единице, выполняется сортировка пузырьк лучшем случае асимптотика равна O(nlogn), в худшем – O(n^2). Какая асимптотика в среднем мне не очень понятно, на практике похоже на O(nlogn).

Реализация:

```
void combsort(int* l, int* r) {
	int sz = r - l;
	if (sz <= 1) return;
	double k = 1.2473309;
	int step = sz - 1;
	while (step > 1) {
		for (int* i = l; i + step < r; i++) {
			if (*i > *(i + step))
				swap(*i, *(i + step));
		}
		step /= k;
	}
	bool b = true;
	while (b) {
		b = false;
		for (int* i = l; i + 1 < r; i++) {
			if (*i > *(i + 1)) {
				swap(*i, *(i + 1));
				b = true;
			}
```

```
                }
        }
}
```

Об этих сортировках (пузырьком, шейкерной и расческой) также можно почитать здесь.

Сортировка вставками / Insertion sort

Создадим массив, в котором после завершения алгоритма будет лежать ответ. Будем поочередно вставлять элементы из исходного масси чтобы элементы в массиве-ответе всегда были отсортированы. Асимптотика в среднем и худшем случае – $O(n^2)$, в лучшем – $O(n)$. Реализо алгоритм удобнее по-другому (создавать новый массив и реально что-то вставлять в него относительно сложно): просто сделаем так, чтобı отсортирован был некоторый префикс исходного массива, вместо вставки будем менять текущий элемент с предыдущим, пока они стоят в неправильном порядке.

Реализация:

```
void insertionsort(int* l, int* r) {
        for (int *i = l + 1; i < r; i++) {
                int* j = i;
                while (j > l && *(j - 1) > *j) {
                        swap(*(j - 1), *j);
                        j--;
                }
        }
}
```

Сортировка Шелла / Shellsort

Используем ту же идею, что и сортировка с расческой, и применим к сортировке вставками. Зафиксируем некоторое расстояние. Тогда эле массива разобьются на классы – в один класс попадают элементы, расстояние между которыми кратно зафиксированному расстоянию. Отсортируем сортировкой вставками каждый класс. В отличие от сортировки расческой, неизвестен оптимальный набор расстояний. Суще довольно много последовательностей с разными оценками. Последовательность Шелла – первый элемент равен длине массива, каждый следующий вдвое меньше предыдущего. Асимптотика в худшем случае – $O(n^2)$. Последовательность Хиббарда – 2^n — 1, асимптотика в худ случае – $O(n^{1,5})$, последовательность Седжвика (формула нетривиальна, можете ее посмотреть по ссылке ниже) — $O(n^{4/3})$, Пратта (все произведения степеней двойки и тройки) — $O(n\log^2 n)$. Отмечу, что все эти последовательности нужно рассчитать только до размера масси запускать от большего от меньшему (иначе получится просто сортировка вставками). Также я провел дополнительное исследование и протестировал разные последовательности вида $s_i = a * s_{i-1} + k * s_{i-1}$ (отчасти это было навеяно эмпирической последовательностью Ци одной из лучших последовательностей расстояний для небольшого количества элементов). Наилучшими оказались последовательности с коэффициентами a = 3, k = 1/3; a = 4, k = 1/4 и a = 4, k = -1/5.

Несколько полезных ссылок:

Сортировка Шелла в русскоязычной Википедии
Сортировка Шелла в англоязычной Википедии
Статья на Хабре

Реализации:

```
void shellsort(int* l, int* r) {
        int sz = r - l;
        int step = sz / 2;
        while (step >= 1) {
                for (int *i = l + step; i < r; i++) {
                        int *j = i;
                        int *diff = j - step;
                        while (diff >= l && *diff > *j) {
                                swap(*diff, *j);
                                j = diff;
                                diff = j - step;
                        }
                }
                step /= 2;
```

```
        }
}
void shellsorthib(int* l, int* r) {
        int sz = r - l;
        if (sz <= 1) return;
        int step = 1;
        while (step < sz) step <<= 1;
        step >>= 1;
        step--;
        while (step >= 1) {
                for (int *i = l + step; i < r; i++) {
                        int *j = i;
                        int *diff = j - step;
                        while (diff >= l && *diff > *j) {
                                swap(*diff, *j);
                                j = diff;
                                diff = j - step;
                        }
                }
                step /= 2;
        }
}
int steps[100];
void shellsortsedgwick(int* l, int* r) {
        int sz = r - l;
        steps[0] = 1;
        int q = 1;
        while (steps[q - 1] * 3 < sz) {
                if (q % 2 == 0)
                        steps[q] = 9 * (1 << q) - 9 * (1 << (q / 2)) + 1;
                else
                        steps[q] = 8 * (1 << q) - 6 * (1 << ((q + 1) / 2)) + 1;
                q++;
        }
        q--;
        for (; q >= 0; q--) {
                int step = steps[q];
                for (int *i = l + step; i < r; i++) {
                        int *j = i;
                        int *diff = j - step;
                        while (diff >= l && *diff > *j) {
                                swap(*diff, *j);
                                j = diff;
                                diff = j - step;
                        }
                }
        }
}
void shellsortpratt(int* l, int* r) {
        int sz = r - l;
        steps[0] = 1;
        int cur = 1, q = 1;
        for (int i = 1; i < sz; i++) {
                int cur = 1 << i;
                if (cur > sz / 2) break;
                for (int j = 1; j < sz; j++) {
                        cur *= 3;
                        if (cur > sz / 2) break;
                        steps[q++] = cur;
                }
        }
        insertionsort(steps, steps + q);
        q--;
        for (; q >= 0; q--) {
                int step = steps[q];
                for (int *i = l + step; i < r; i++) {
                        int *j = i;
                        int *diff = j - step;
                        while (diff >= l && *diff > *j) {
                                swap(*diff, *j);
                                j = diff;
```

```
                                diff = j - step;
                        }
                }
        }
}
void myshell1(int* l, int* r) {
        int sz = r - l, q = 1;
        steps[0] = 1;
        while (steps[q - 1] < sz) {
                int s = steps[q - 1];
                steps[q++] = s * 4 + s / 4;
        }
        q--;
        for (; q >= 0; q--) {
                int step = steps[q];
                for (int *i = l + step; i < r; i++) {
                        int *j = i;
                        int *diff = j - step;
                        while (diff >= l && *diff > *j) {
                                swap(*diff, *j);
                                j = diff;
                                diff = j - step;
                        }
                }
        }
}
void myshell2(int* l, int* r) {
        int sz = r - l, q = 1;
        steps[0] = 1;
        while (steps[q - 1] < sz) {
                int s = steps[q - 1];
                steps[q++] = s * 3 + s / 3;
        }
        q--;
        for (; q >= 0; q--) {
                int step = steps[q];
                for (int *i = l + step; i < r; i++) {
                        int *j = i;
                        int *diff = j - step;
                        while (diff >= l && *diff > *j) {
                                swap(*diff, *j);
                                j = diff;
                                diff = j - step;
                        }
                }
        }
}
void myshell3(int* l, int* r) {
        int sz = r - l, q = 1;
        steps[0] = 1;
        while (steps[q - 1] < sz) {
                int s = steps[q - 1];
                steps[q++] = s * 4 - s / 5;
        }
        q--;
        for (; q >= 0; q--) {
                int step = steps[q];
                for (int *i = l + step; i < r; i++) {
                        int *j = i;
                        int *diff = j - step;
                        while (diff >= l && *diff > *j) {
                                swap(*diff, *j);
                                j = diff;
                                diff = j - step;
                        }
                }
        }
}
```

Сортировка деревом / Tree sort

Будем вставлять элементы в двоичное дерево поиска. После того, как все элементы вставлены достаточно обойти дерево в глубину и полу отсортированный массив. Если использовать сбалансированное дерево, например красно-черное, асимптотика будет равна O(nlogn) в худ среднем и лучшем случае. В реализации использован контейнер multiset.

Здесь можно почитать про деревья поиска:

[Википедия]
[Статья на Хабре]
[И ещё статья на Хабре]

Реализация:

```
void treesort(int* l, int* r) {
        multiset<int> m;
        for (int *i = l; i < r; i++)
                m.insert(*i);
        for (int q : m)
                *l = q, l++;
}
```

Гномья сортировка / Gnome sort

Алгоритм похож на сортировку вставками. Поддерживаем указатель на текущий элемент, если он больше предыдущего или он первый — смещаем указатель на позицию вправо, иначе меняем текущий и предыдущий элементы местами и смещаемся влево.

Реализация:

```
void gnomesort(int* l, int* r) {
        int *i = l;
        while (i < r) {
                if (i == l || *(i - 1) <= *i) i++;
                else swap(*(i - 1), *i), i--;
        }
}
```

Сортировка выбором / Selection sort

На очередной итерации будем находить минимум в массиве после текущего элемента и менять его с ним, если надо. Таким образом, посл итерации первые i элементов будут стоять на своих местах. Асимптотика: O(n^2) в лучшем, среднем и худшем случае. Нужно отметить, что сортировку можно реализовать двумя способами – сохраняя минимум и его индекс или просто переставляя текущий элемент с рассматриваемым, если они стоят в неправильном порядке. Первый способ оказался немного быстрее, поэтому он и реализован.

Реализация:

```
void selectionsort(int* l, int* r) {
        for (int *i = l; i < r; i++) {
                int minz = *i, *ind = i;
                for (int *j = i + 1; j < r; j++) {
                        if (*j < minz) minz = *j, ind = j;
                }
                swap(*i, *ind);
        }
}
```

Пирамидальная сортировка / Heapsort

Развитие идеи сортировки выбором. Воспользуемся структурой данных «куча» (или «пирамида», откуда и название алгоритма). Она позво получать минимум за O(1), добавляя элементы и извлекая минимум за O(logn). Таким образом, асимптотика O(nlogn) в худшем, среднем и лучшем случае. Реализовывал кучу я сам, хотя в C++ и есть контейнер priority_queue, поскольку этот контейнер довольно медленный.

Почитать про кучу можно здесь:

[Википедия](#)
[Статья на Хабре](#)

Реализация:

```
template <class T>
class heap {
public:
	int size() {
		return n;
	}
	int top() {
		return h[0];
	}
	bool empty() {
		return n == 0;
	}
	void push(T a) {
		h.push_back(a);
		SiftUp(n);
		n++;
	}
	void pop() {
		n--;
		swap(h[n], h[0]);
		h.pop_back();
		SiftDown(0);
	}
	void clear() {
		h.clear();
		n = 0;
	}
	T operator [] (int a) {
		return h[a];
	}
private:
	vector<T> h;
	int n = 0;
	void SiftUp(int a) {
		while (a) {
			int p = (a - 1) / 2;
			if (h[p] > h[a]) swap(h[p], h[a]);
			else break;
			a--; a /= 2;
		}
	}
	void SiftDown(int a) {
		while (2 * a + 1 < n) {
			int l = 2 * a + 1, r = 2 * a + 2;
			if (r == n) {
				if (h[l] < h[a]) swap(h[l], h[a]);
				break;
			}
			else if (h[l] <= h[r]) {
				if (h[l] < h[a]) {
					swap(h[l], h[a]);
					a = l;
				}
				else break;
			}
			else if (h[r] < h[a]) {
				swap(h[r], h[a]);
				a = r;
			}
```

```
                        else break;
                }
        }
};
void heapsort(int* l, int* r) {
        heap<int> h;
        for (int *i = l; i < r; i++) h.push(*i);
        for (int *i = l; i < r; i++) {
                *i = h.top();
                h.pop();
        }
}
```

Быстрая сортировка / Quicksort

Выберем некоторый опорный элемент. После этого перекинем все элементы, меньшие его, налево, а большие – направо. Рекурсивно вызс от каждой из частей. В итоге получим отсортированный массив, так как каждый элемент меньше опорного стоял раньше каждого большего опорного. Асимптотика: O(nlogn) в среднем и лучшем случае, $O(n^2)$. Наихудшая оценка достигается при неудачном выборе опорного элеме Моя реализация этого алгоритма совершенно стандартна, идем одновременно слева и справа, находим пару элементов, таких, что левый элемент больше опорного, а правый меньше, и меняем их местами. Помимо чистой быстрой сортировки, участвовала в сравнении и сорти переходящая при малом количестве элементов на сортировку вставками. Константа подобрана тестированием, а сортировка вставками — наилучшая сортировка, подходящая для этой задачи (хотя не стоит из-за этого думать, что она самая быстрая из квадратичных).

Реализация:

```
void quicksort(int* l, int* r) {
        if (r - l <= 1) return;
        int z = *(l + (r - l) / 2);
        int* ll = l, *rr = r - 1;
        while (ll <= rr) {
                while (*ll < z) ll++;
                while (*rr > z) rr--;
                if (ll <= rr) {
                        swap(*ll, *rr);
                        ll++;
                        rr--;
                }
        }
        if (l < rr) quicksort(l, rr + 1);
        if (ll < r) quicksort(ll, r);
}
void quickinssort(int* l, int* r) {
        if (r - l <= 32) {
                insertionsort(l, r);
                return;
        }
        int z = *(l + (r - l) / 2);
        int* ll = l, *rr = r - 1;
        while (ll <= rr) {
                while (*ll < z) ll++;
                while (*rr > z) rr--;
                if (ll <= rr) {
                        swap(*ll, *rr);
                        ll++;
                        rr--;
                }
        }
        if (l < rr) quickinssort(l, rr + 1);
        if (ll < r) quickinssort(ll, r);
}
```

Сортировка слиянием / Merge sort

Сортировка, основанная на парадигме «разделяй и властвуй». Разделим массив пополам, рекурсивно отсортируем части, после чего выпо. процедуру слияния: поддерживаем два указателя, один на текущий элемент первой части, второй – на текущий элемент второй части. Из з двух элементов выбираем минимальный, вставляем в ответ и сдвигаем указатель, соответствующий минимуму. Слияние работает за O(n), уровней всего logn, поэтому асимптотика O(nlogn). Эффективно заранее создать временный массив и передать его в качестве аргумента функции. Эта сортировка рекурсивна, как и быстрая, а потому возможен переход на квадратичную при небольшом числе элементов.

Реализация:

```
void merge(int* l, int* m, int* r, int* temp) {
        int *cl = l, *cr = m, cur = 0;
        while (cl < m && cr < r) {
                if (*cl < *cr) temp[cur++] = *cl, cl++;
                else temp[cur++] = *cr, cr++;
        }
        while (cl < m) temp[cur++] = *cl, cl++;
        while (cr < r) temp[cur++] = *cr, cr++;
        cur = 0;
        for (int* i = l; i < r; i++)
                 *i = temp[cur++];
}
void _mergesort(int* l, int* r, int* temp) {
        if (r - l <= 1) return;
        int *m = l + (r - l) / 2;
        _mergesort(l, m, temp);
        _mergesort(m, r, temp);
        merge(l, m, r, temp);
}
void mergesort(int* l, int* r) {
        int* temp = new int[r - l];
        _mergesort(l, r, temp);
        delete temp;
}
void _mergeinssort(int* l, int* r, int* temp) {
        if (r - l <= 32) {
                insertionsort(l, r);
                return;
        }
        int *m = l + (r - l) / 2;
        _mergeinssort(l, m, temp);
        _mergeinssort(m, r, temp);
        merge(l, m, r, temp);
}
void mergeinssort(int* l, int* r) {
        int* temp = new int[r - l];
        _mergeinssort(l, r, temp);
        delete temp;
}
```

Сортировка подсчетом / Counting sort

Создадим массив размера r – l, где l – минимальный, а r – максимальный элемент массива. После этого пройдем по массиву и подсчитаем количество вхождений каждого элемента. Теперь можно пройти по массиву значений и выписать каждое число столько раз, сколько нужно. Асимптотика – O(n + r — l). Можно модифицировать этот алгоритм, чтобы он стал стабильным: для этого определим место, где должно сто очередное число (это просто префиксные суммы в массиве значений) и будем идти по исходному массиву слева направо, ставя элемент н правильное место и увеличивая позицию на 1. Эта сортировка не тестировалась, поскольку большинство тестов содержало достаточно бо числа, не позволяющие создать массив требуемого размера. Однако она, тем не менее, пригодилась.

Блочная сортировка / Bucket sort

(также известна как корзинная и карманная сортировка). Пусть l – минимальный, а r – максимальный элемент массива. Разобьем элементы блоки, в первом будут элементы от l до l + k, во втором – от l + k до l + 2k и т.д., где k = (r – l) / количество блоков. В общем-то, если количес блоков равно двум, то данный алгоритм превращается в разновидность быстрой сортировки. Асимптотика этого алгоритма неясна, время р зависит и от входных данных, и от количества блоков. Утверждается, что на удачных данных время работы линейно. Реализация этого алг оказалась одной из самых трудных задач. Можно сделать это так: просто создавать новые массивы, рекурсивно их сортировать и склеиват Однако такой подход все же довольно медленный и меня не устроил. В эффективной реализации используется несколько идей:

1) Не будем создавать новых массивов. Для этого воспользуемся техникой сортировки подсчетом – подсчитаем количество элементов в ка

блоке, префиксные суммы и, таким образом, позицию каждого элемента в массиве.

2) Не будем запускаться из пустых блоков. Занесем индексы непустых блоков в отдельный массив и запустимся только от них.

3) Проверим, отсортирован ли массив. Это не ухудшит время работы, так как все равно нужно сделать проход с целью нахождения миниму максимума, однако позволит алгоритму ускориться на частично отсортированных данных, ведь элементы вставляются в новые блоки в том порядке, что и в исходном массиве.

4) Поскольку алгоритм получился довольно громоздким, при небольшом количестве элементов он крайне неэффективен. До такой степени переход на сортировку вставками ускоряет работу примерно в 10 раз.

Осталось только понять, какое количество блоков нужно выбрать. На рандомизированных тестах мне удалось получить следующую оценку блоков для 10^7 элементов и 3000 для 10^8. Подобрать формулу не удалось – время работы ухудшалось в несколько раз.

Реализация:

```
void _newbucketsort(int* l, int* r, int* temp) {
        if (r - l <= 64) {
                insertionsort(l, r);
                return;
        }
        int minz = *l, maxz = *l;
        bool is_sorted = true;
        for (int *i = l + 1; i < r; i++) {
                minz = min(minz, *i);
                maxz = max(maxz, *i);
                if (*i < *(i - 1)) is_sorted = false;
        }
        if (is_sorted) return;
        int diff = maxz - minz + 1;
        int numbuckets;
        if (r - l <= 1e7) numbuckets = 1500;
        else numbuckets = 3000;
        int range = (diff + numbuckets - 1) / numbuckets;
        int* cnt = new int[numbuckets + 1];
        for (int i = 0; i <= numbuckets; i++)
                cnt[i] = 0;
        int cur = 0;
        for (int* i = l; i < r; i++) {
                temp[cur++] = *i;
                int ind = (*i - minz) / range;
                cnt[ind + 1]++;
        }
        int sz = 0;
        for (int i = 1; i <= numbuckets; i++)
                if (cnt[i]) sz++;
        int* run = new int[sz];
        cur = 0;
        for (int i = 1; i <= numbuckets; i++)
                if (cnt[i]) run[cur++] = i - 1;
        for (int i = 1; i <= numbuckets; i++)
                cnt[i] += cnt[i - 1];
        cur = 0;
        for (int *i = l; i < r; i++) {
                int ind = (temp[cur] - minz) / range;
                *(l + cnt[ind]) = temp[cur];
                cur++;
                cnt[ind]++;
        }
        for (int i = 0; i < sz; i++) {
                int r = run[i];
                if (r != 0) _newbucketsort(l + cnt[r - 1], l + cnt[r], temp);
                else _newbucketsort(l, l + cnt[r], temp);
        }
        delete run;
        delete cnt;
}
void newbucketsort(int* l, int* r) {
```

```
        int *temp = new int[r - l];
        _newbucketsort(l, r, temp);
        delete temp;
}
```

Поразрядная сортировка / Radix sort

(также известна как цифровая сортировка). Существует две версии этой сортировки, в которых, на мой взгляд, мало общего, кроме идеи воспользоваться представлением числа в какой-либо системе счисления (например, двоичной).

LSD (least significant digit):

Представим каждое число в двоичном виде. На каждом шаге алгоритма будем сортировать числа таким образом, чтобы они были отсортир по первым k * i битам, где k – некоторая константа. Из данного определения следует, что на каждом шаге достаточно стабильно сортироват элементы по новым k битам. Для этого идеально подходит сортировка подсчетом (необходимо 2^k памяти и времени, что немного при удачн выборе константы). Асимптотика: O(n), если считать, что числа фиксированного размера (а в противном случае нельзя было бы считать, ч сравнение двух чисел выполняется за единицу времени). Реализация довольно проста.

Реализация:

```
int digit(int n, int k, int N, int M) {
        return (n >> (N * k) & (M - 1));
}
void _radixsort(int* l, int* r, int N) {
        int k = (32 + N - 1) / N;
        int M = 1 << N;
        int sz = r - l;
        int* b = new int[sz];
        int* c = new int[M];
        for (int i = 0; i < k; i++) {
                for (int j = 0; j < M; j++)
                        c[j] = 0;
                for (int* j = l; j < r; j++)
                        c[digit(*j, i, N, M)]++;
                for (int j = 1; j < M; j++)
                        c[j] += c[j - 1];
                for (int* j = r - 1; j >= l; j--)
                        b[--c[digit(*j, i, N, M)]] = *j;
                int cur = 0;
                for (int* j = l; j < r; j++)
                        *j = b[cur++];
        }
        delete b;
        delete c;
}
void radixsort(int* l, int* r) {
        _radixsort(l, r, 8);
}
```

MSD (most significant digit):

На самом деле, некоторая разновидность блочной сортировки. В один блок будут попадать числа с равными k битами. Асимптотика такая : и у LSD версии. Реализация очень похожа на блочную сортировку, но проще. В ней используется функция digit, определенная в реализаци версии.

Реализация:

```
void _radixsortmsd(int* l, int* r, int N, int d, int* temp) {
        if (d == -1) return;
        if (r - l <= 32) {
                insertionsort(l, r);
                return;
        }
```

```
        int M = 1 << N;
        int* cnt = new int[M + 1];
        for (int i = 0; i <= M; i++)
                cnt[i] = 0;
        int cur = 0;
        for (int* i = l; i < r; i++) {
                temp[cur++] = *i;
                cnt[digit(*i, d, N, M) + 1]++;
        }
        int sz = 0;
        for (int i = 1; i <= M; i++)
                if (cnt[i]) sz++;
        int* run = new int[sz];
        cur = 0;
        for (int i = 1; i <= M; i++)
                if (cnt[i]) run[cur++] = i - 1;
        for (int i = 1; i <= M; i++)
                cnt[i] += cnt[i - 1];
        cur = 0;
        for (int *i = l; i < r; i++) {
                int ind = digit(temp[cur], d, N, M);
                *(l + cnt[ind]) = temp[cur];
                cur++;
                cnt[ind]++;
        }
        for (int i = 0; i < sz; i++) {
                int r = run[i];
                if (r != 0) _radixsortmsd(l + cnt[r - 1], l + cnt[r], N, d - 1, temp);
                else _radixsortmsd(l, l + cnt[r], N, d - 1, temp);
        }
        delete run;
        delete cnt;
}
void radixsortmsd(int* l, int* r) {
        int* temp = new int[r - l];
        _radixsortmsd(l, r, 8, 3, temp);
        delete temp;
}
```

Битонная сортировка / Bitonic sort:

Идея данного алгоритма заключается в том, что исходный массив преобразуется в битонную последовательность – последовательность, ко сначала возрастает, а потом убывает. Ее можно эффективно отсортировать следующим образом: разобьем массив на две части, создадим массива, в первый добавим все элементы, равные минимуму из соответственных элементов каждой из двух частей, а во второй – равные максимуму. Утверждается, что получатся две битонные последовательности, каждую из которых можно рекурсивно отсортировать тем же образом, после чего можно склеить два массива (так как любой элемент первого меньше или равен любого элемента второго). Для того, чт преобразовать исходный массив в битонную последовательность, сделаем следующее: если массив состоит из двух элементов, можно про завершиться, иначе разделим массив пополам, рекурсивно вызовем от половинок алгоритм, после чего отсортируем первую часть по поря, вторую в обратном порядке и склеим. Очевидно, получится битонная последовательность. Асимптотика: O(nlog^{2}n), поскольку при построен битонной последовательности мы использовали сортировку, работающую за O(nlogn), а всего уровней было logn. Также заметим, что разм массива должен быть равен степени двойки, так что, возможно, придется его дополнять фиктивными элементами (что не влияет на асимпт

Реализация:

```
void bitseqsort(int* l, int* r, bool inv) {
        if (r - l <= 1) return;
        int *m = l + (r - l) / 2;
        for (int *i = l, *j = m; i < m && j < r; i++, j++) {
                if (inv ^ (*i > *j)) swap(*i, *j);
        }
        bitseqsort(l, m, inv);
        bitseqsort(m, r, inv);
}
void makebitonic(int* l, int* r) {
        if (r - l <= 1) return;
        int *m = l + (r - l) / 2;
```

```
        makebitonic(l, m);
        bitseqsort(l, m, 0);
        makebitonic(m, r);
        bitseqsort(m, r, 1);
}
void bitonicsort(int* l, int* r) {
        int n = 1;
        int inf = *max_element(l, r) + 1;
        while (n < r - l) n *= 2;
        int* a = new int[n];
        int cur = 0;
        for (int *i = l; i < r; i++)
                a[cur++] = *i;
        while (cur < n) a[cur++] = inf;
        makebitonic(a, a + n);
        bitseqsort(a, a + n, 0);
        cur = 0;
        for (int *i = l; i < r; i++)
                *i = a[cur++];
        delete a;
}
```

Timsort

Гибридная сортировка, совмещающая сортировку вставками и сортировку слиянием. Разобьем элементы массива на несколько подмассив небольшого размера, при этом будем расширять подмассив, пока элементы в нем отсортированы. Отсортируем подмассивы сортировкой вставками, пользуясь тем, что она эффективно работает на отсортированных массивах. Далее будем сливать подмассивы как в сортировк слиянием, беря их примерно равного размера (иначе время работы приблизится к квадратичному). Для этого удобного хранить подмассивы стеке, поддерживая инвариант — чем дальше от вершины, тем больше размер, и сливать подмассивы на верхушке только тогда, когда раз третьего по отдаленности от вершины подмассива больше или равен сумме их размеров. Асимптотика: O(n) в лучшем случае и O(nlogn) в среднем и худшем случае. Реализация нетривиальна, твердой уверенности в ней у меня нет, однако время работы она показала довольно неплохое и согласующееся с моими представлениями о том, как должна работать эта сортировка.

Подробнее timsort описан здесь:

Здесь
Здесь

Реализация:

```
void _timsort(int* l, int* r, int* temp) {
        int sz = r - l;
        if (sz <= 64) {
                insertionsort(l, r);
                return;
        }
        int minrun = sz, f = 0;
        while (minrun >= 64) {
                f |= minrun & 1;
                minrun >>= 1;
        }
        minrun += f;
        int* cur = l;
        stack<pair<int, int*>> s;
        while (cur < r) {
                int* c1 = cur;
                while (c1 < r - 1 && *c1 <= *(c1 + 1)) c1++;
                int* c2 = cur;
                while (c2 < r - 1 && *c2 >= *(c2 + 1)) c2++;
                if (c1 >= c2) {
                        c1 = max(c1, cur + minrun - 1);
                        c1 = min(c1, r - 1);
                        insertionsort(cur, c1 + 1);
                        s.push({ c1 - cur + 1, cur });
                        cur = c1 + 1;
                }
```

```
                else {
                        c2 = max(c2, cur + minrun - 1);
                        c2 = min(c2, r - 1);
                        reverse(cur, c2 + 1);
                        insertionsort(cur, c2 + 1);
                        s.push({ c2 - cur + 1, cur });
                        cur = c2 + 1;
                }
                while (s.size() >= 3) {
                        pair<int, int*> x = s.top();
                        s.pop();
                        pair<int, int*> y = s.top();
                        s.pop();
                        pair<int, int*> z = s.top();
                        s.pop();
                        if (z.first >= x.first + y.first && y.first >= x.first) {
                                s.push(z);
                                s.push(y);
                                s.push(x);
                                break;
                        }
                        else if (z.first >= x.first + y.first) {
                                merge(y.second, x.second, x.second + x.first, temp);
                                s.push(z);
                                s.push({ x.first + y.first, y.second });
                        }
                        else {
                                merge(z.second, y.second, y.second + y.first, temp);
                                s.push({ z.first + y.first, z.second });
                                s.push(x);
                        }
                }
        }
        while (s.size() != 1) {
                pair<int, int*> x = s.top();
                s.pop();
                pair<int, int*> y = s.top();
                s.pop();
                if (x.second < y.second) swap(x, y);
                merge(y.second, x.second, x.second + x.first, temp);
                s.push({ y.first + x.first, y.second });
        }
}
void timsort(int* l, int* r) {
        int* temp = new int[r - l];
        _timsort(l, r, temp);
        delete temp;
}
```

Тестирование

Железо и система

Процессор: Intel Core i7-3770 CPU 3.40 GHz
ОЗУ: 8 ГБ
Тестирование проводилось на почти чистой системе Windows 10 x64, установленной за несколько дней до запуска. Использованная IDE – Microsoft Visual Studio 2015.

Тесты

Все тесты поделены на четыре группы. Первая группа – массив случайных чисел по разным модулям (10, 1000, 10^5, 10^7 и 10^9). Вторая груп массив, разбивающийся на несколько отсортированных подмассивов. Фактически брался массив случайных чисел по модулю 10^9, а далее отсортировывались подмассивы размера, равного минимуму из длины оставшегося суффикса и случайного числа по модулю некоторой константы. Последовательность констант – 10, 100, 1000 и т.д. вплоть до размера массива. Третья группа – изначально отсортированный м случайных чисел с некоторым числом «свопов» — перестановок двух случайных элементов. Последовательность количеств свопов такая и в предыдущей группе. Наконец, последняя группа состоит из нескольких тестов с полностью отсортированным массивом (в прямом и обр порядке), нескольких тестов с исходным массивом натуральных чисел от 1 до n, в котором несколько чисел заменены на случайное, и тест большим количеством повторений одного элемента (10%, 25%, 50%, 75% и 90%). Таким образом, тесты позволяют посмотреть, как сортиро работают на случайных и частично отсортированных массивах, что выглядит наиболее существенным. Четвертая группа во многом направ против сортировок с линейным временем работы, которые любят последовательности случайных чисел. В конце статьи есть ссылка на фа котором подробно описаны все тесты.

Размер входных данных

Было бы довольно глупо сравнивать, например, сортировку с линейным временем работы и квадратичную, и запускать их на тестах одного размера. Поэтому каждая из групп тестов делится еще на четыре группы, размера 10^5, 10^6, 10^7и 10^8 элементов. Сортировки были разбиты группы, в первой – квадратичные (сортировка пузырьком, вставками, выбором, шейкерная и гномья), во второй – нечто среднее между логарифмическим временем и квадратом, (битонная, несколько видов сортировки Шелла и сортировка деревом), в третьей все остальные. то, возможно, удивит, что сортировка деревом попала не в третью группу, хотя ее асимптотика и O(nlogn), но, к сожалению, ее константа оч велика. Сортировки первой группы тестировались на тестах с 10^5элементов, второй группы – на тестах с 10^6и 10^7, третьей – на тестах с 10 Именно такие размеры данных позволяют как-то увидеть рост времени работы, при меньших размерах слишком велика погрешность, при больших алгоритм работает слишком долго (или же недостаток оперативной памяти). С первой группой я не стал заморачиваться, чтобы н нарушать десятикратное увеличение (10^4 элементов для квадратичных сортировок слишком мало), в конце концов, сами по себе они представляют мало интереса.

Как проводилось тестирование

На каждом тесте было производилось 20 запусков, итоговое время работы – среднее по получившимся значениям. Почти все результаты б получены после одного запуска программы, однако из-за нескольких ошибок в коде и системных глюков (все же тестирование продолжало почти неделю чистого времени) некоторые сортировки и тесты пришлось впоследствии перетестировать.

Тонкости реализации

Возможно, кого-то удивит, что в реализации самого процесса тестирования я не использовал указатели на функции, что сильно сократило Оказалось, что это заметно замедляет работу алгоритма (примерно на 5-10%). Поэтому я использовал отдельный вызов каждой функции (конечно, не отразилось бы на относительной скорости, но… все же хочется улучшить и абсолютную). По той же причине были заменены ве на обычные массивы, не были использованы шаблоны и функции-компараторы. Все это более актуально для промышленного использова алгоритма, нежели его тестирования.

Результаты

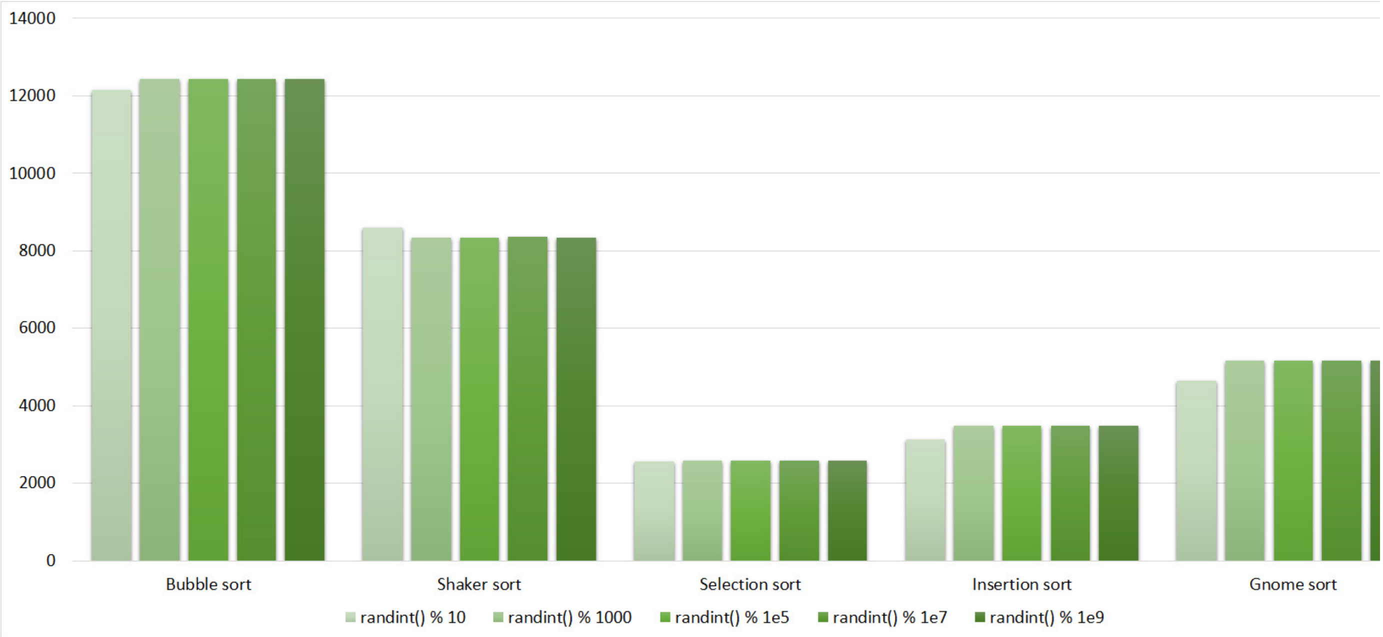
Все результаты доступны в нескольких видах – три диаграммы (гистограмма, на которой видно изменение скорости при переходе к следую ограничению на одном типе тестов, график, изображающий то же самое, но иногда более наглядно, и гистограмма, на которой видно, кака сортировка лучше всего работает на каком-то типе тестов) и таблицы, на которых они основаны. Третья группа была разделена еще на три а то мало что было бы понятно. Впрочем, и так далеко не все диаграммы удачны (в полезности третьего типа диаграмм я вообще сильно сомневаюсь), но, надеюсь, каждый сможет найти наиболее подходящую для понимания.

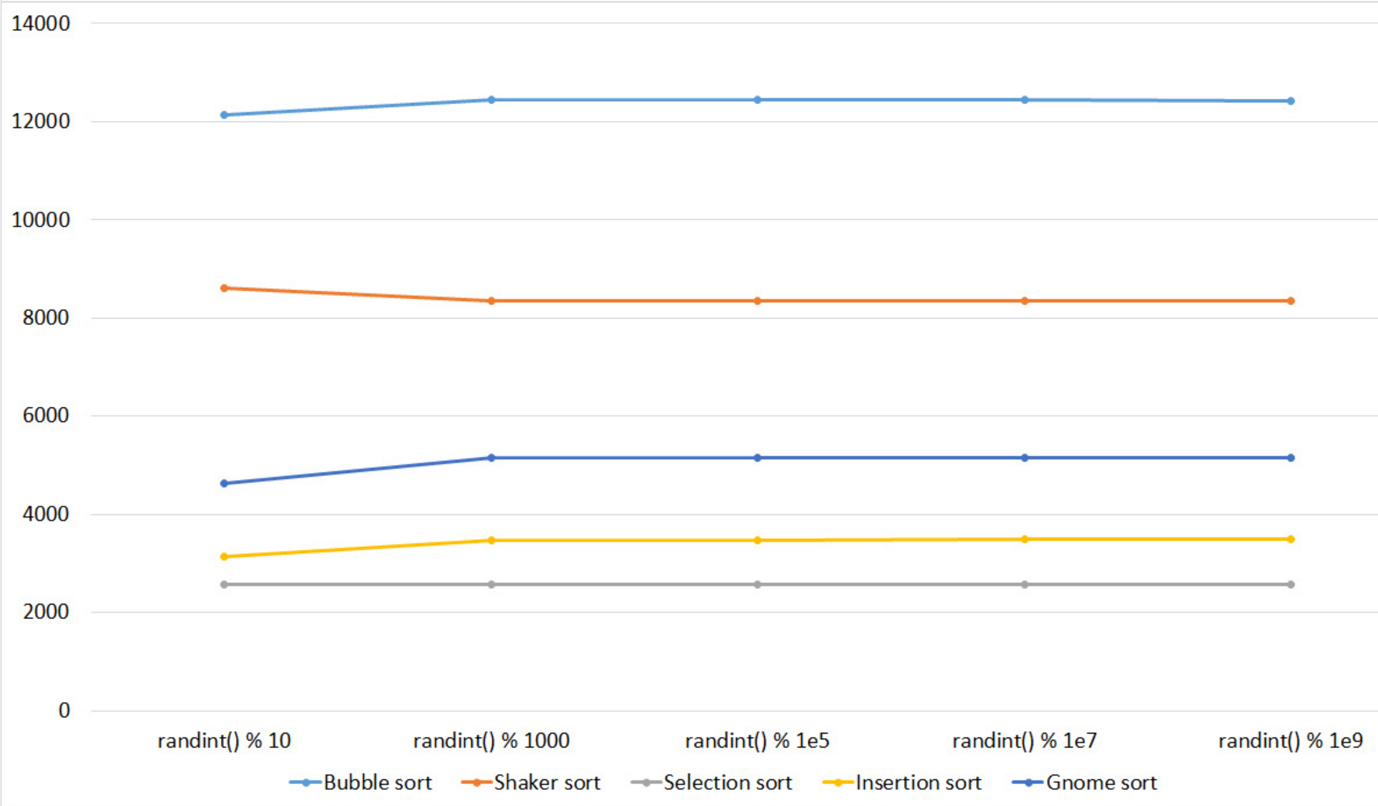
Поскольку картинок очень много, они скрыты спойлерами. Немного комментариев по поводу обозначений. Сортировки названы так, как вы если это сортировка Шелла, то в скобочках указан автор последовательности, к названиям сортировок, переходящих на сортировку вставк приписано Ins (для компактности). В диаграммах у второй группы тестов обозначена возможная длина отсортированных подмассивов, у тр группы — количество свопов, у четвертой — количество замен. Общий результат рассчитывался как среднее по четырем группам.

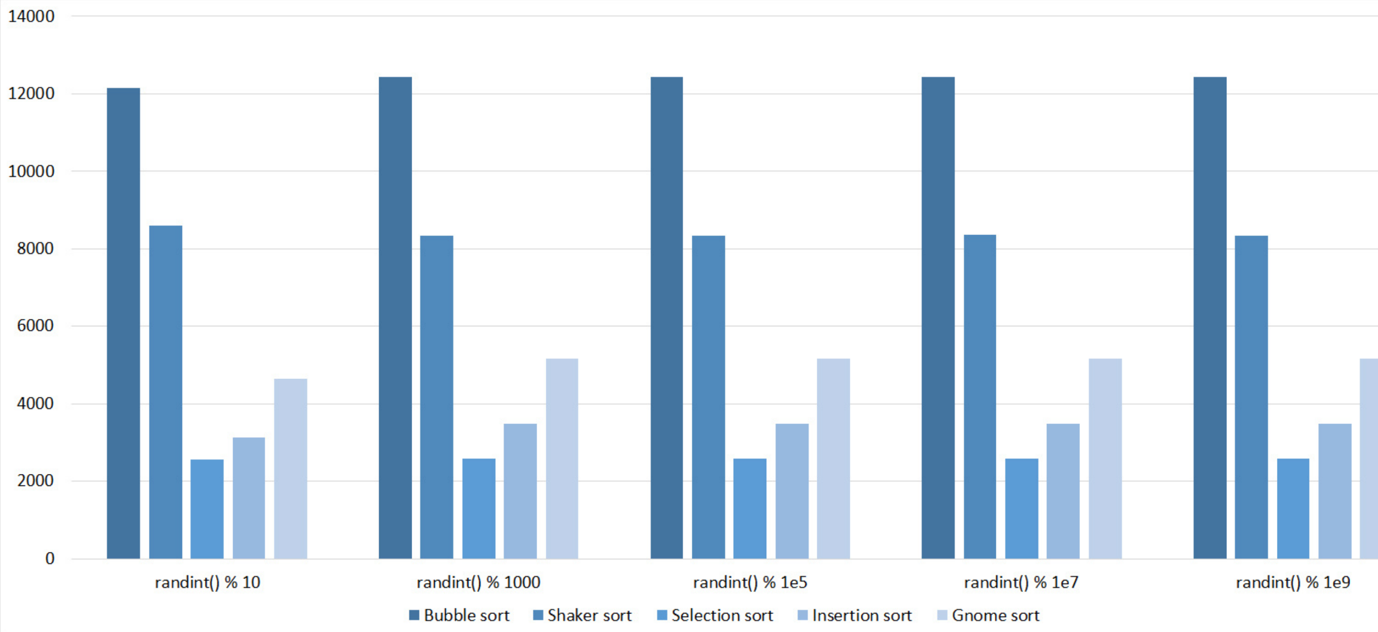
Первая группа сортировок

Массив случайных чисел

Таблицы


14000
12000
10000
8000
6000
4000
2000
0
Bubble sort
Shaker sort
Selection sort
Insertion sort
Gnome sort
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9


14000
12000
10000
8000
6000
4000
2000
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
Bubble sort
Shaker sort
Selection sort
Insertion sort
Gnome sort



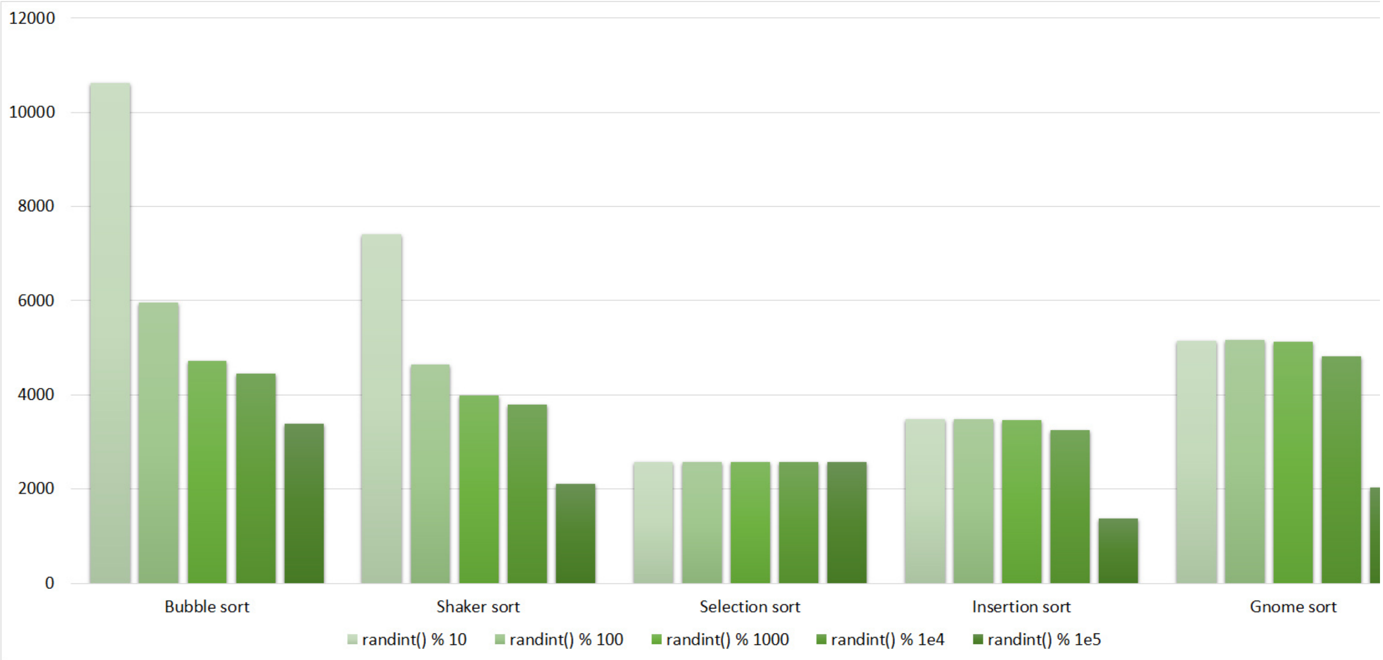
1e5	randint() % 10	randint() % 1000	randint() % 1e5	randint() % 1e7	randint() % 1e9	Ито
Bubble sort	12140,27	12435,47	12434,23	12439,46	12432,81	1237
Shaker sort	8603,16	8345,15	8342,92	8351,89	8339,06	8396
Selection sort	2567,94	2572,81	2577,14	2577,34	2577,04	2574
Insertion sort	3133,05	3479,45	3478,69	3486,73	3482,74	3412
Gnome sort	4634,51	5155,16	5158,38	5163,62	5159,04	5054

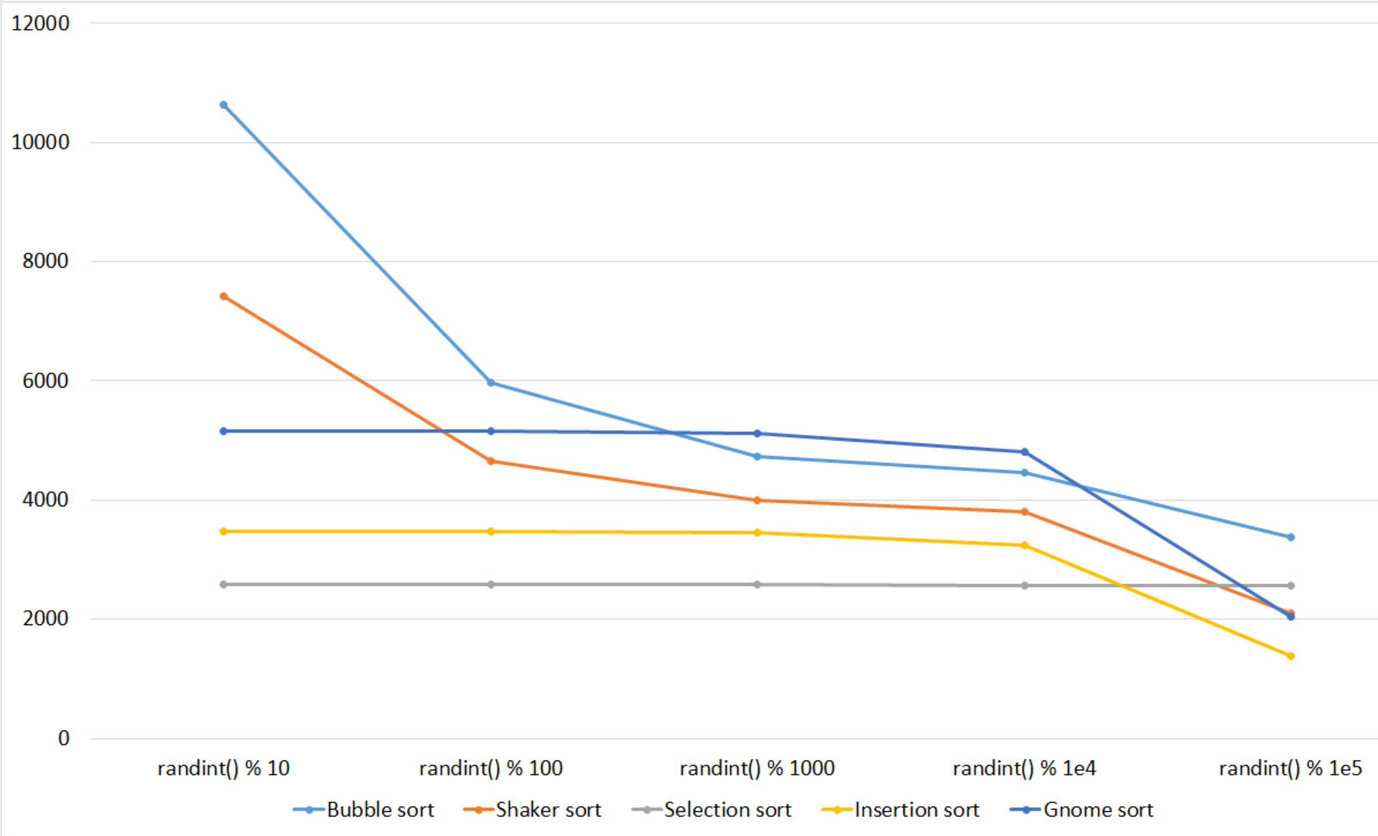
1e5	Итог
Selection sort	2574,45
Insertion sort	3412,13
Gnome sort	5054,14
Shaker sort	8396,44
Bubble sort	12376,45

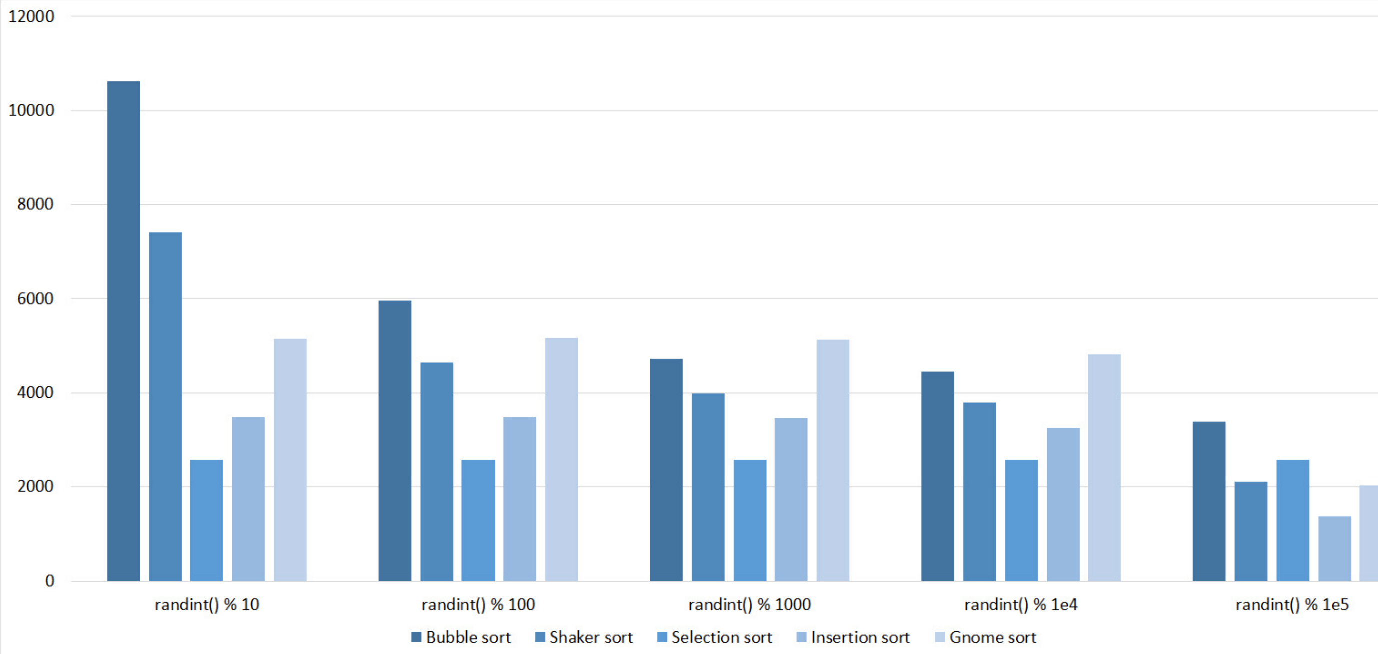
Совсем скучные результаты, даже частичная отсортированность при небольшом модуле почти незаметна.

Частично отсортированный массив

Таблицы


12000
10000
8000
6000
4000
2000
0
Bubble sort
Shaker sort
Selection sort
Insertion sort
Gnome sort
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5


12000
10000
8000
6000
4000
2000
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
Bubble sort
Shaker sort
Selection sort
Insertion sort
Gnome sort



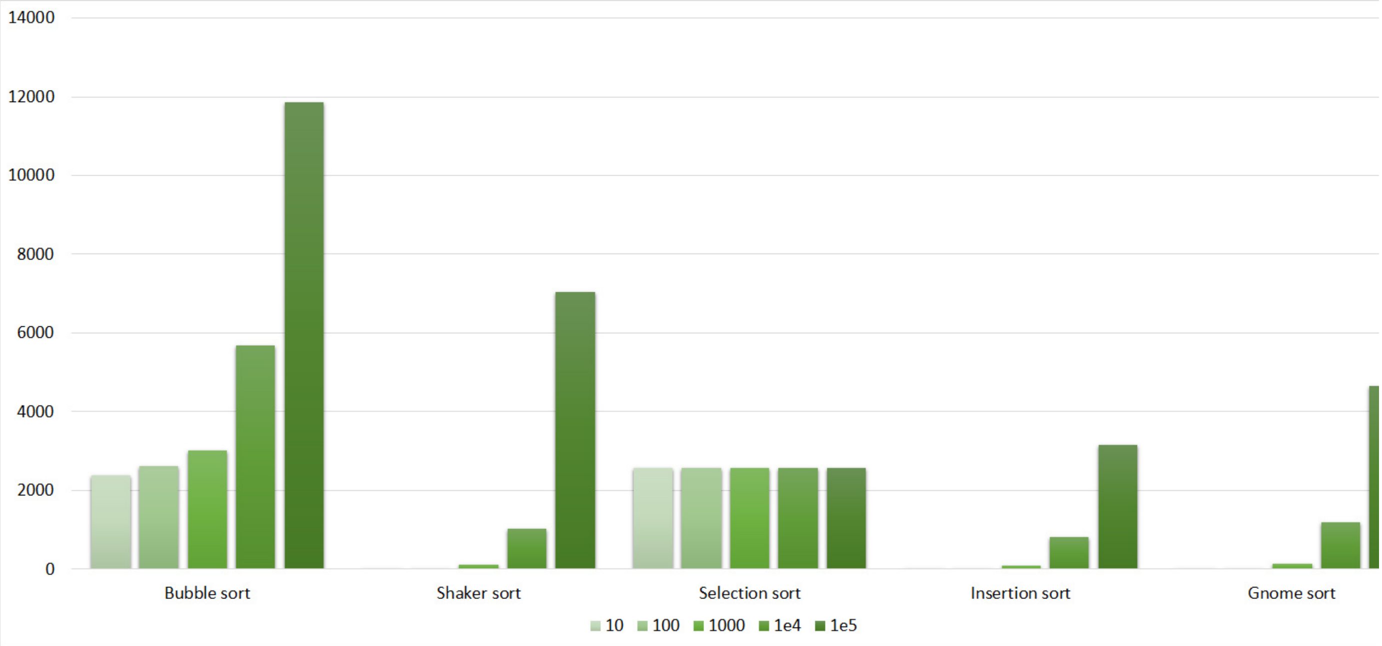
1e5	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	Итог
Bubble sort	10627,09	5969	4730,7	4451,56	3380,56	5831,78
Shaker sort	7420,33	4652,05	3992,53	3802,48	2105,07	4394,49
Selection sort	2578,27	2577,58	2576,65	2574,47	2573,24	2576,04
Insertion sort	3477,29	3482,53	3456,83	3247,58	1376,54	3008,15
Gnome sort	5153,24	5157,65	5120,61	4810,62	2040	4456,42

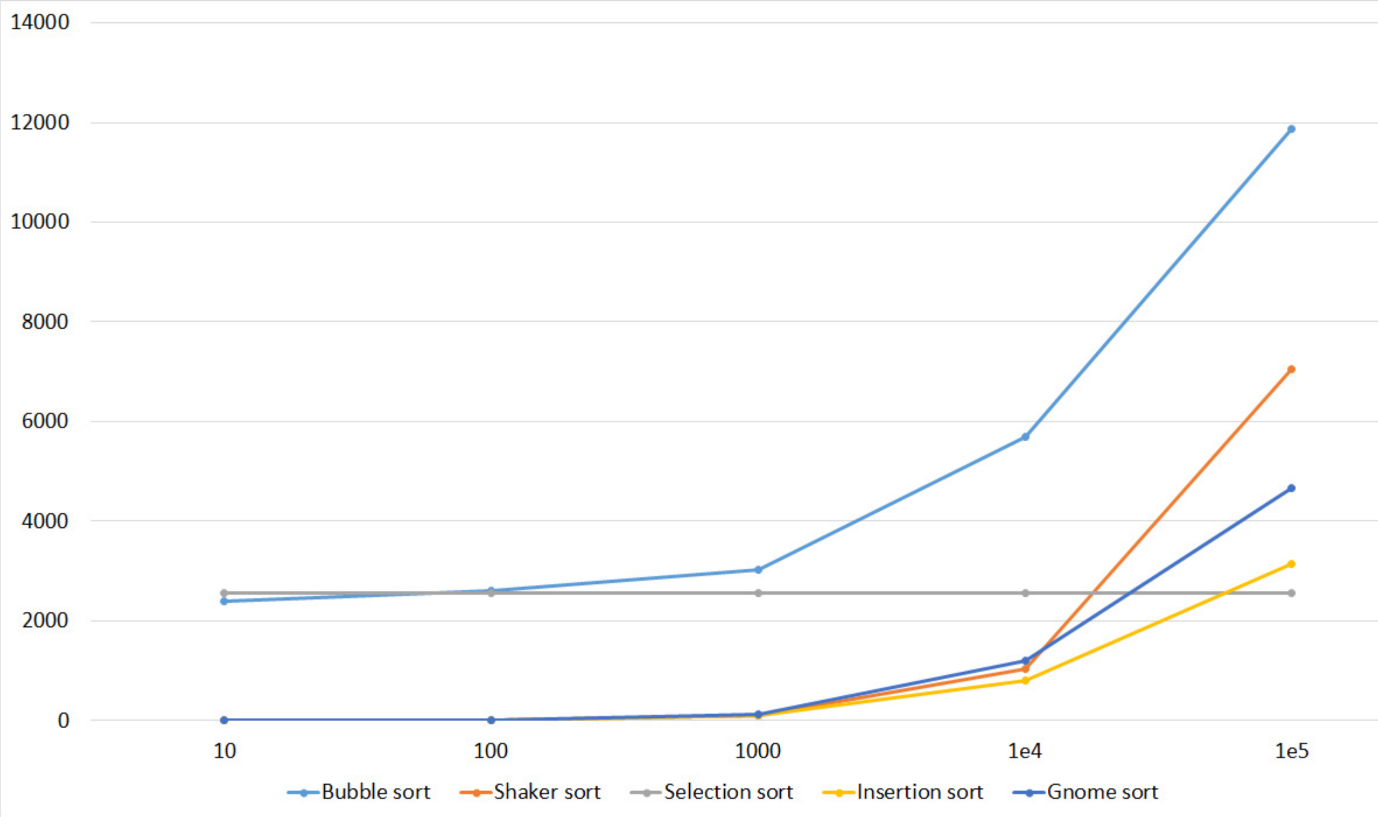
1e5	Итог
Selection sort	2576,042
Insertion sort	3008,154
Shaker sort	4394,492
Gnome sort	4456,424
Bubble sort	5831,782

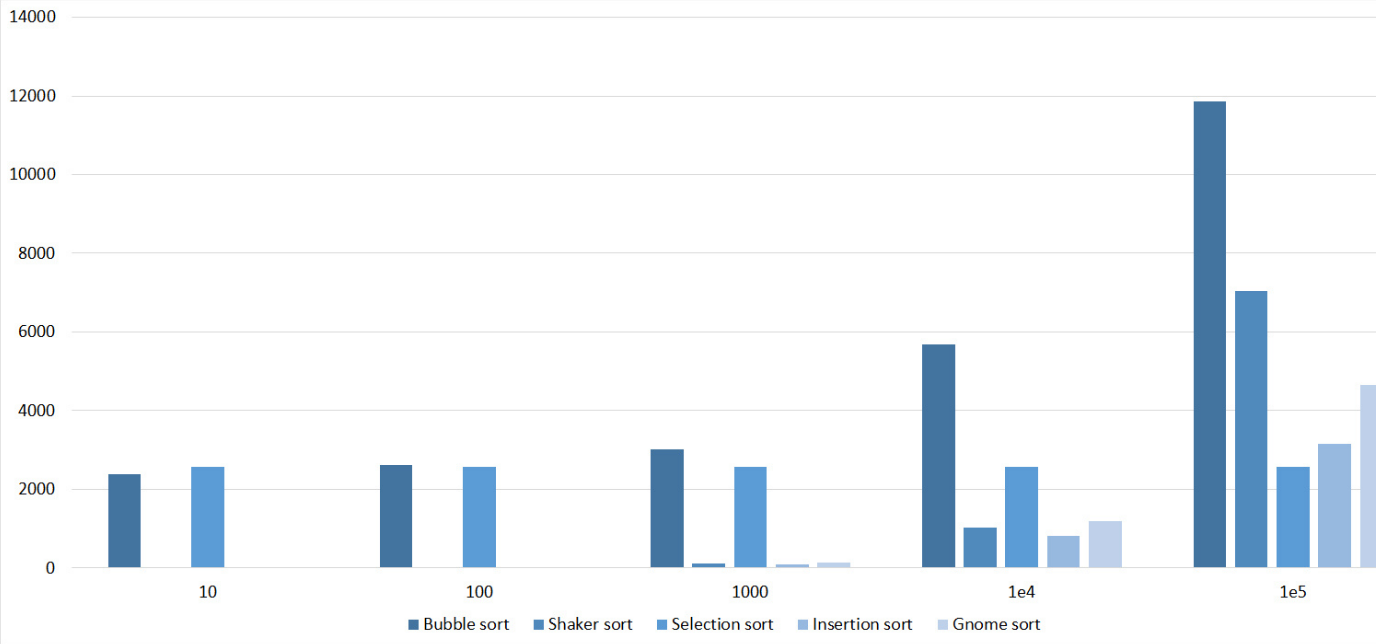
Уже гораздо интереснее. Обменные сортировки наиболее бурно отреагировали, шейкерная даже обогнала гномью. Сортировка вставками ускорилась только под самый конец. Сортировка выбором, конечно, работает совершенно также.

Свопы

Таблицы


14000
12000
10000
8000
6000
4000
2000
0
Bubble sort
Shaker sort
Selection sort
Insertion sort
Gnome sort
10
100
1000
1e4
1e5


14000
12000
10000
8000
6000
4000
2000
0
10
100
1000
1e4
1e5
Bubble sort
Shaker sort
Selection sort
Insertion sort
Gnome sort



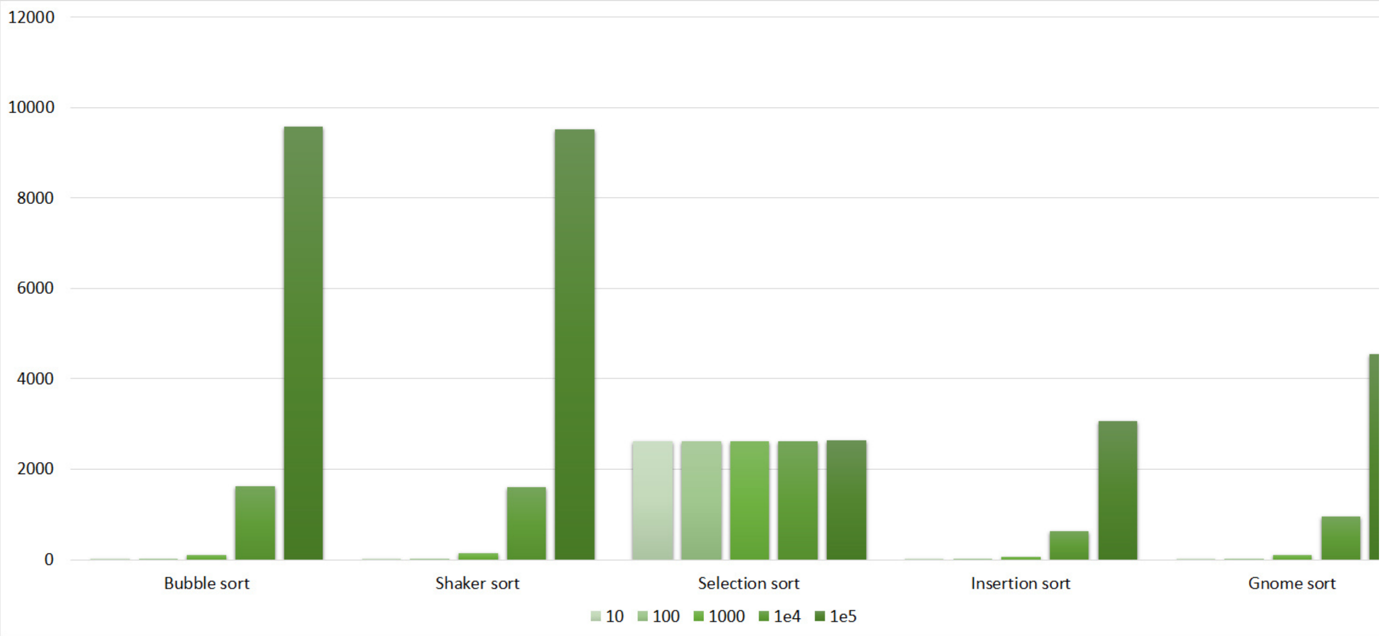
1e5	10	100	1000	1e4	1e5	Итог
Bubble sort	2388,17	2615,87	3026,07	5686,58	11870,28	5117,394
Shaker sort	1,58	11,56	104,65	1030,79	7041,87	1638,09
Selection sort	2567,16	2567	2566,27	2569,89	2569,77	2568,018
Insertion sort	0,71	9,23	91,14	803,23	3146,29	810,12
Gnome sort	1,84	15,24	135,28	1193,07	4656,42	1200,37

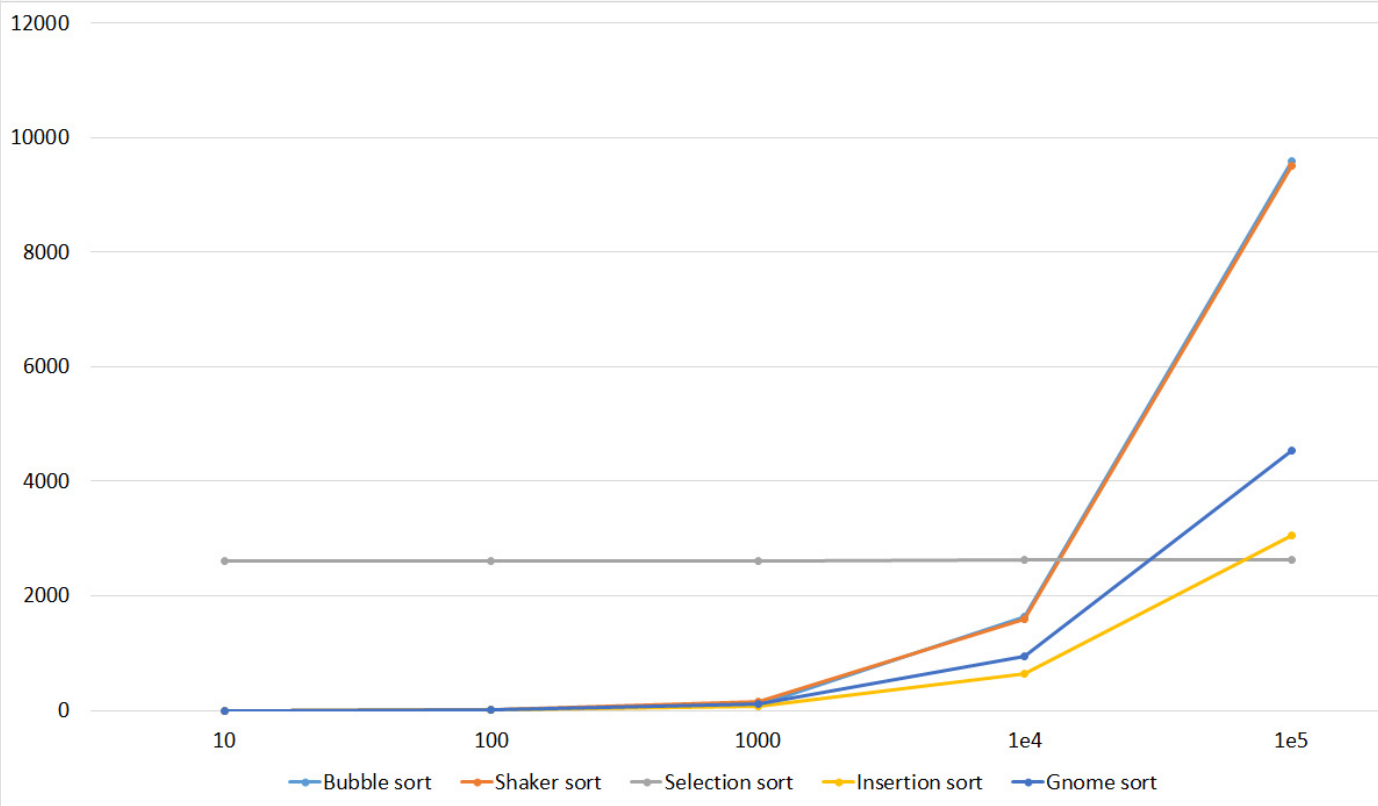
1e5	Итог
Insertion sort	810,12
Gnome sort	1200,37
Shaker sort	1638,09
Selection sort	2568,018
Bubble sort	5117,394

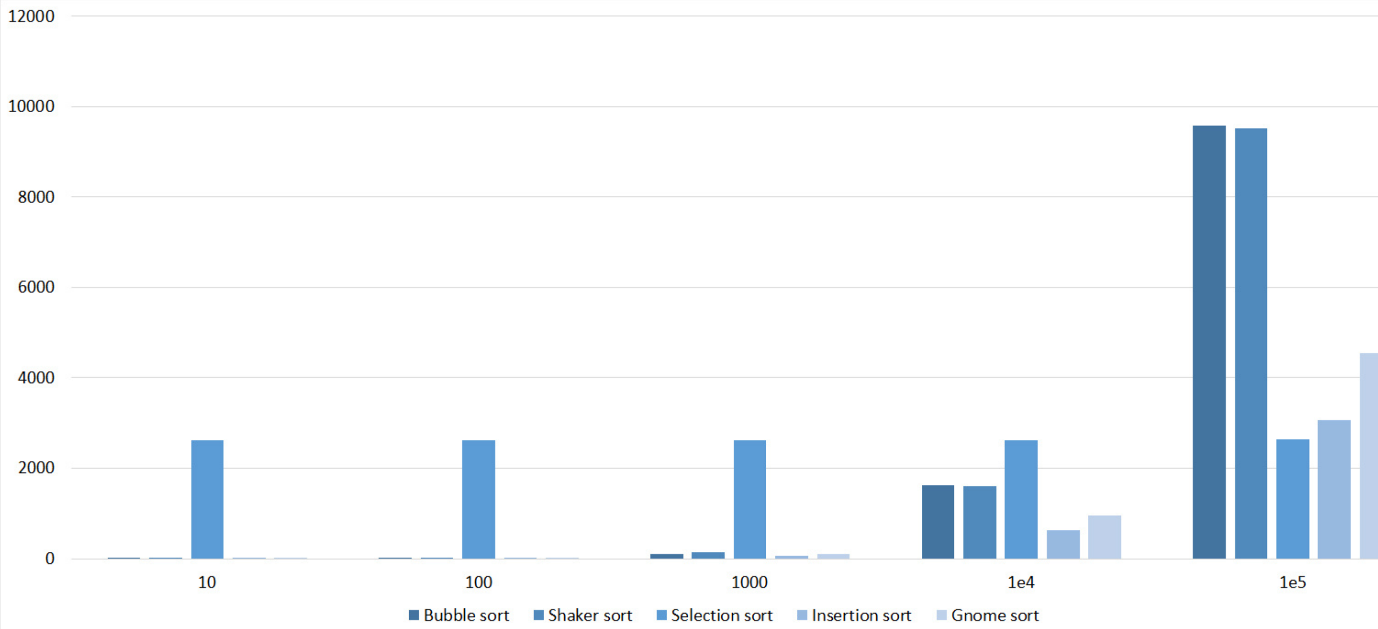
Здесь наконец-то проявила себя сортировка вставками, хотя рост скорости у шейкерной примерно такой же. Здесь проявилась слабость сортировки пузырьком — достаточно одного свопа, перемещающего маленький элемент в конец, и она уже работает медленно. Сортировк выбором оказалась почти в конце.

Изменения в перестановке

▾ Таблицы


12000
10000
8000
6000
4000
2000
0
Bubble sort
Shaker sort
Selection sort
Insertion sort
Gnome sort
10 100 1000 1e4 1e5


12000
10000
8000
6000
4000
2000
0
10
100
1000
1e4
1e5
Bubble sort
Shaker sort
Selection sort
Insertion sort
Gnome sort



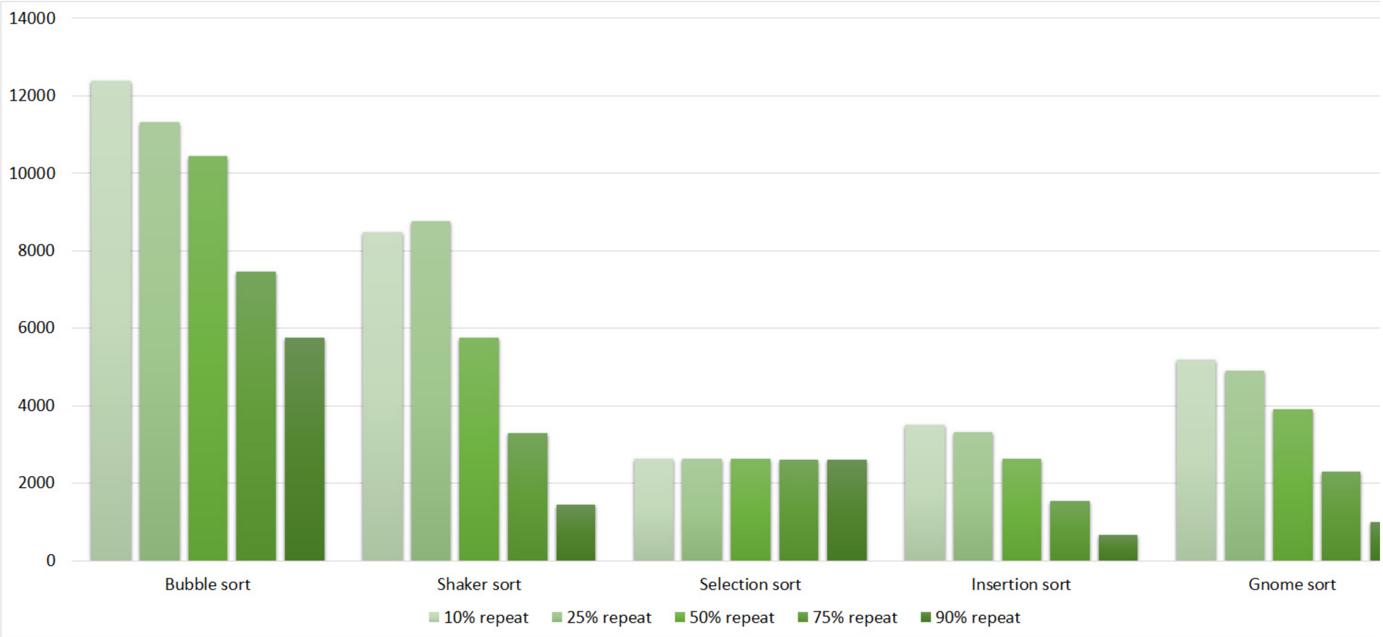
1e5	10	100	1000	1e4	1e5	Итог
Bubble sort	1,03	9,50	93,45	1631,20	9588,78	2264,7
Shaker sort	0,70	14,48	148,30	1603,70	9512,20	2255,8
Selection sort	2609,65	2610,08	2613,30	2619,70	2633,75	2617,3
Insertion sort	0,55	6,65	69,35	639,35	3064,55	756,0
Gnome sort	0,90	11,85	105,45	950,25	4542,25	1122,1

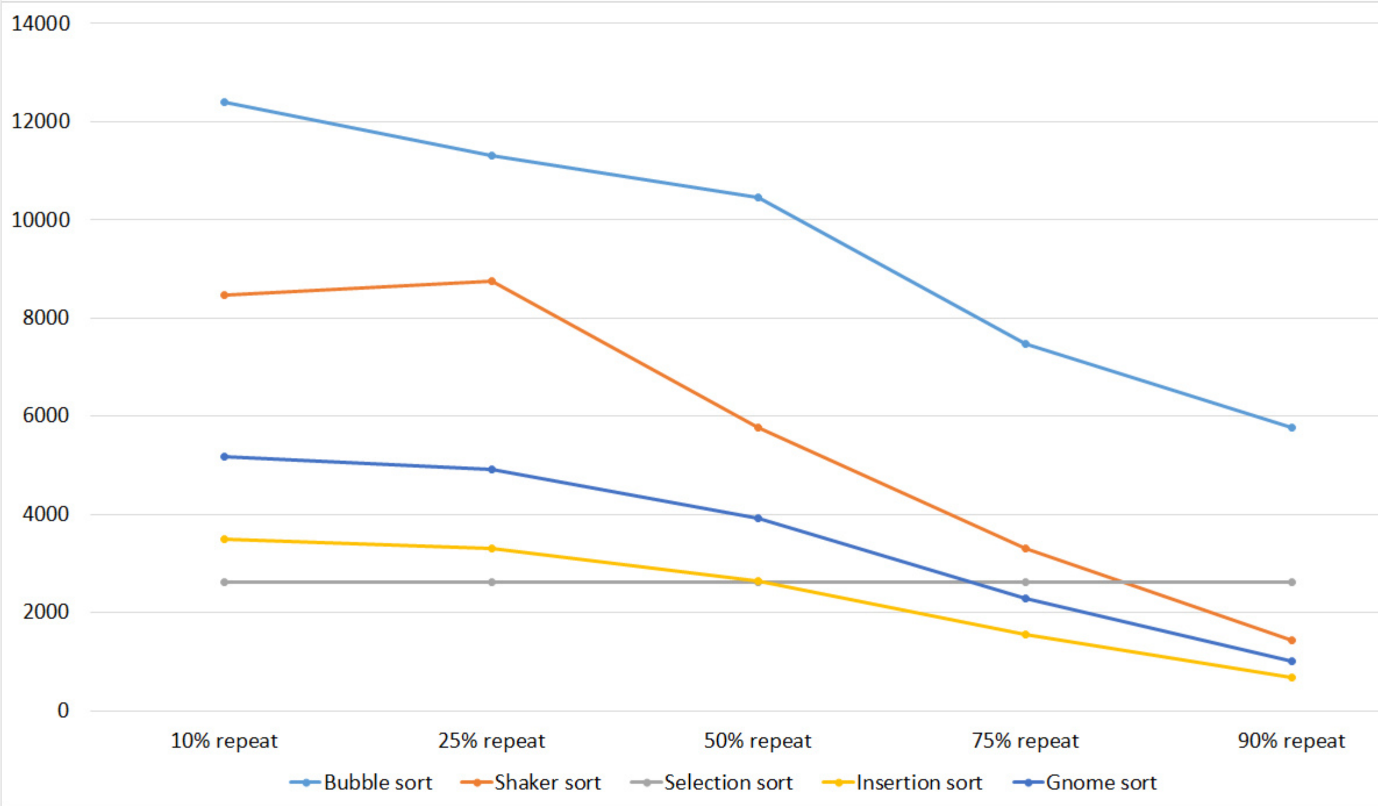
1e5	Итог
Insertion sort	756,09
Gnome sort	1122,14
Shaker sort	2255,88
Bubble sort	2264,79
Selection sort	2617,30

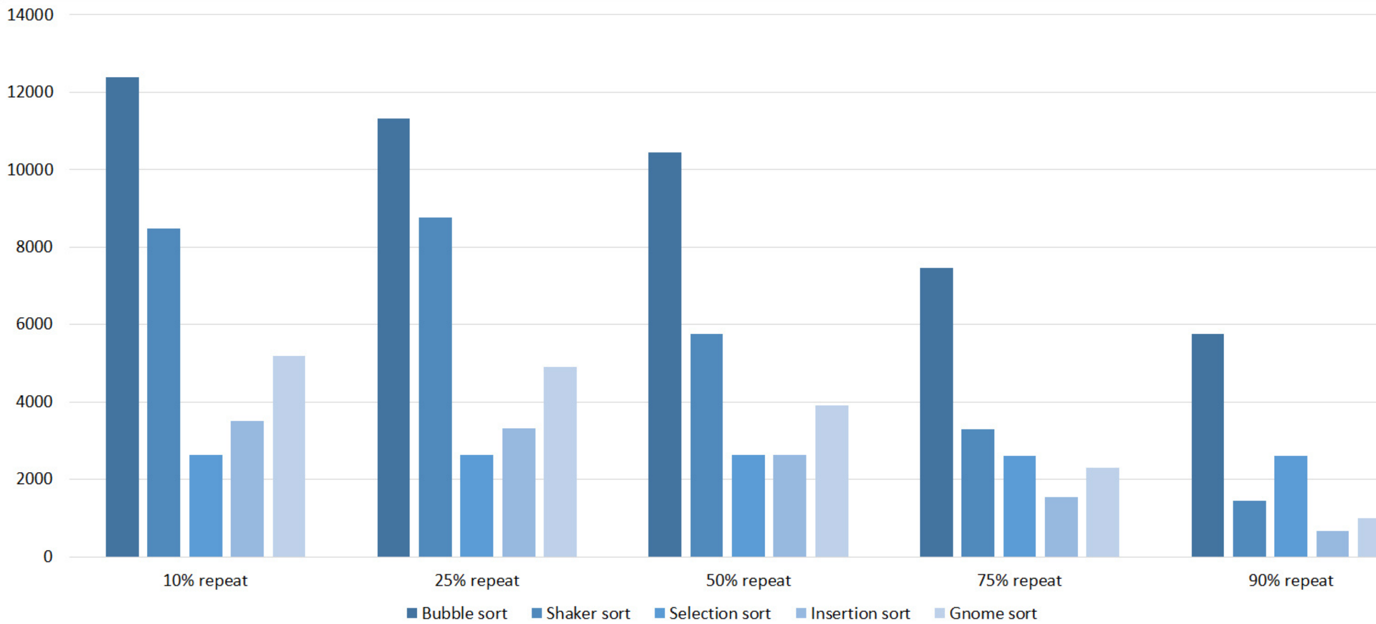
Группа почти ничем не отличается от предыдущей, поэтому результаты похожи. Однако сортировка пузырьком вырывается вперед, так как случайный элемент, вставленный в массив, скорее всего будет больше всех остальных, то есть за одну итерацию переместится в конец. Сортировка выбором стала аутсайдером.

Повторы

Таблицы


14000
12000
10000
8000
6000
4000
2000
0
Bubble sort
Shaker sort
Selection sort
Insertion sort
Gnome sort
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat


14000
12000
10000
8000
6000
4000
2000
0
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat
Bubble sort
Shaker sort
Selection sort
Insertion sort
Gnome sort



1e5	10% repeat	25% repeat	50% repeat	75% repeat	90% repeat	Итог
Bubble sort	12397,50	11317,97	10449,38	7469,30	5767,20	9480,2
Shaker sort	8478,05	8762,40	5756,82	3304,68	1436,70	5547,7
Selection sort	2621,18	2621,85	2618,15	2617,42	2617,45	2619,2
Insertion sort	3498,48	3311,17	2640,42	1540,88	671,27	2332,4
Gnome sort	5184,07	4913,47	3917,07	2286,83	999,50	3460,1

1e5	Итог
Insertion sort	2332,44
Selection sort	2619,21
Gnome sort	3460,19
Shaker sort	5547,73
Bubble sort	9480,27

Здесь все сортировки (кроме, конечно, сортировки выбором) работали почти одинаково, ускоряясь по мере увеличении количества повторс

Итоговые результаты

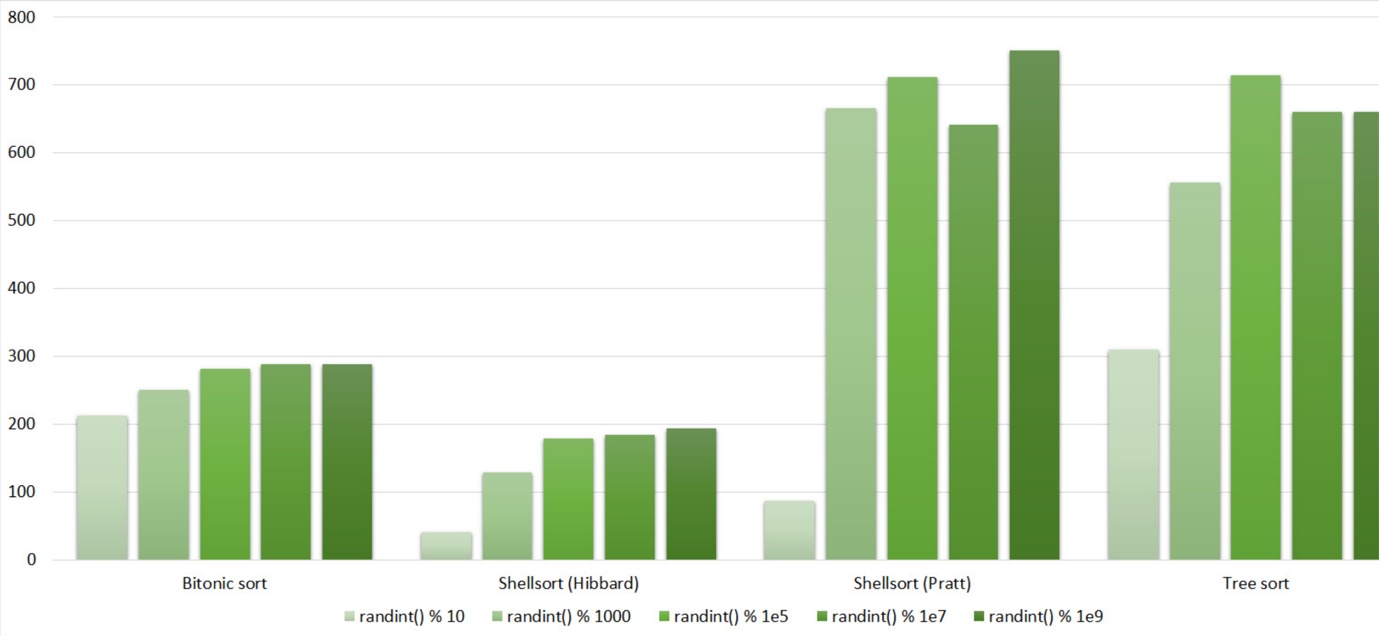
	1e5	Итог
1	Insertion sort	2337,07
2	Selection sort	2583,91
3	Gnome sort	3462,19
4	Shaker sort	4544,39
5	Bubble sort	7116,68

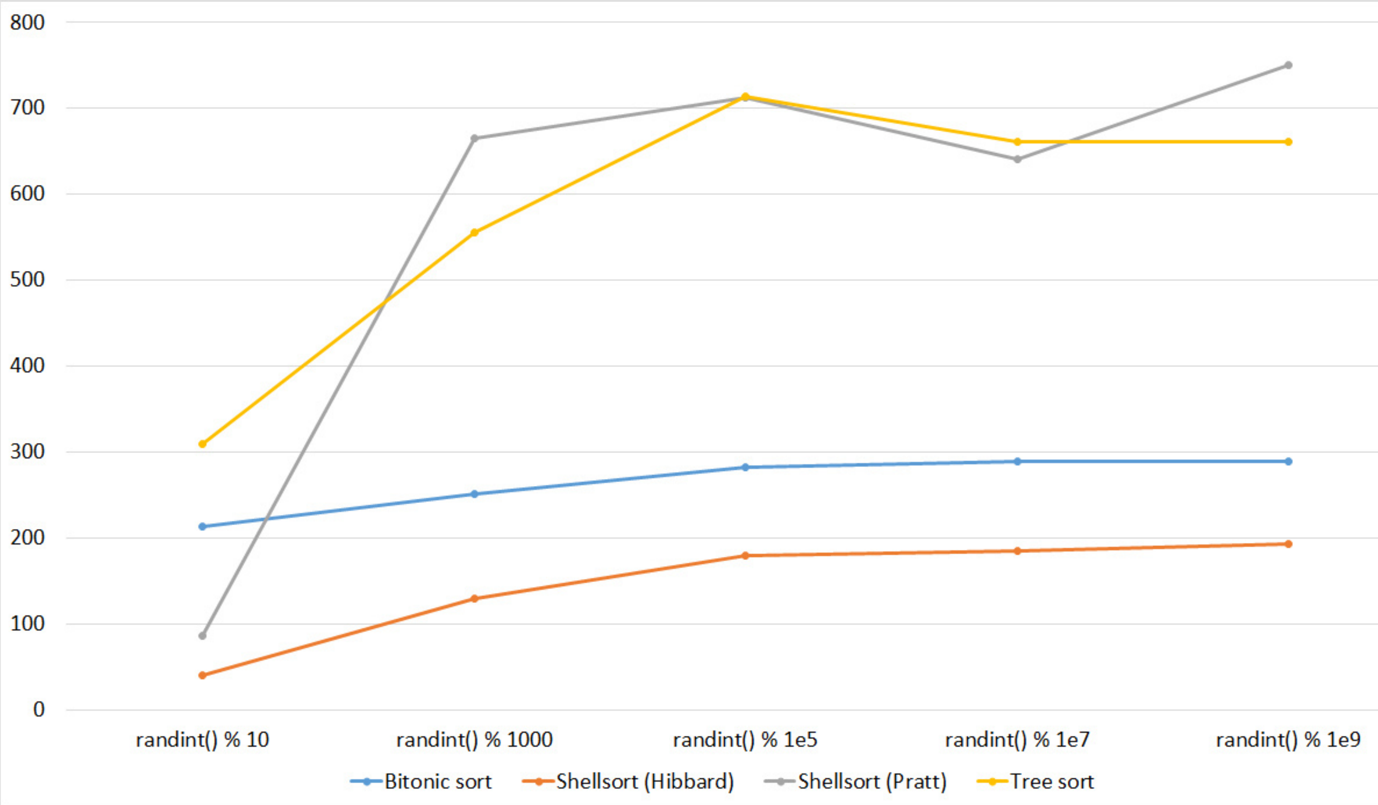
За счет своего абсолютного безразличия к массиву, сортировка выбором, работавшая быстрее всех на случайных данных, все же проигра
сортировке вставками. Гномья сортировка оказалась заметно хуже последней, из-за чего ее практическое применение сомнительно. Шейк
пузырьковая сортировки оказались медленнее всех.

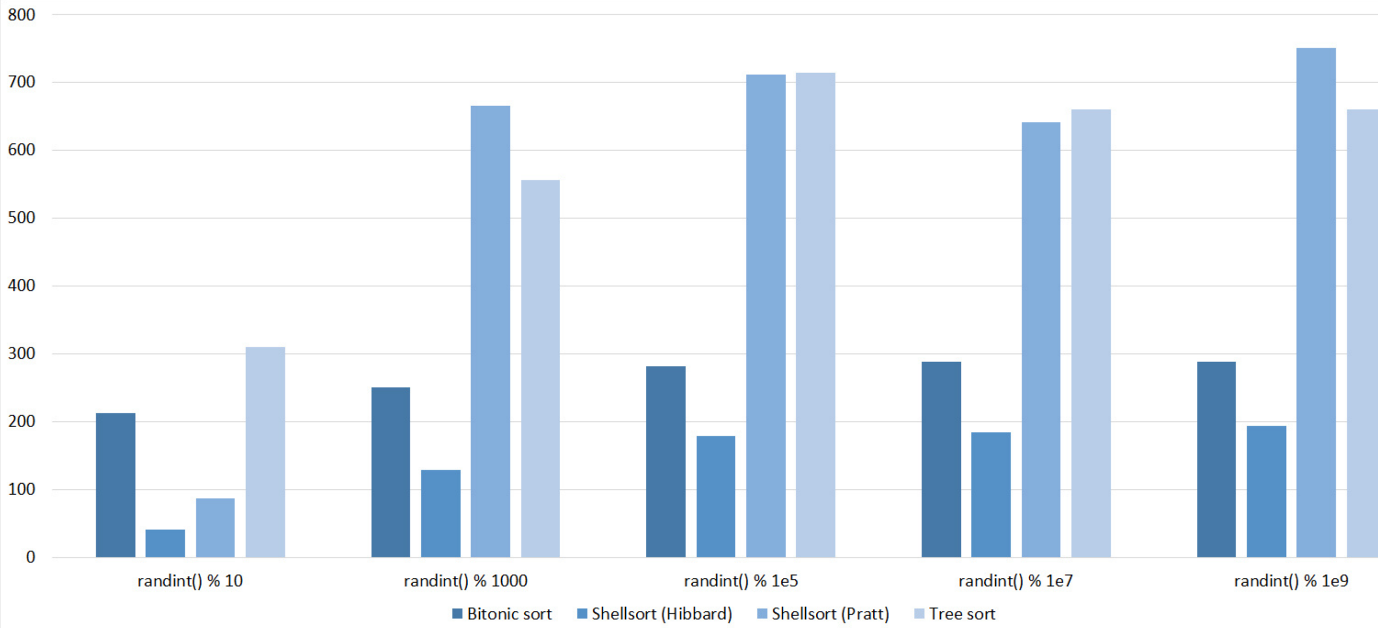
Вторая группа сортировок

Массив случайных чисел

Таблицы, 1e6 элементов


800
700
600
500
400
300
200
100
0
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9


800
700
600
500
400
300
200
100
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort

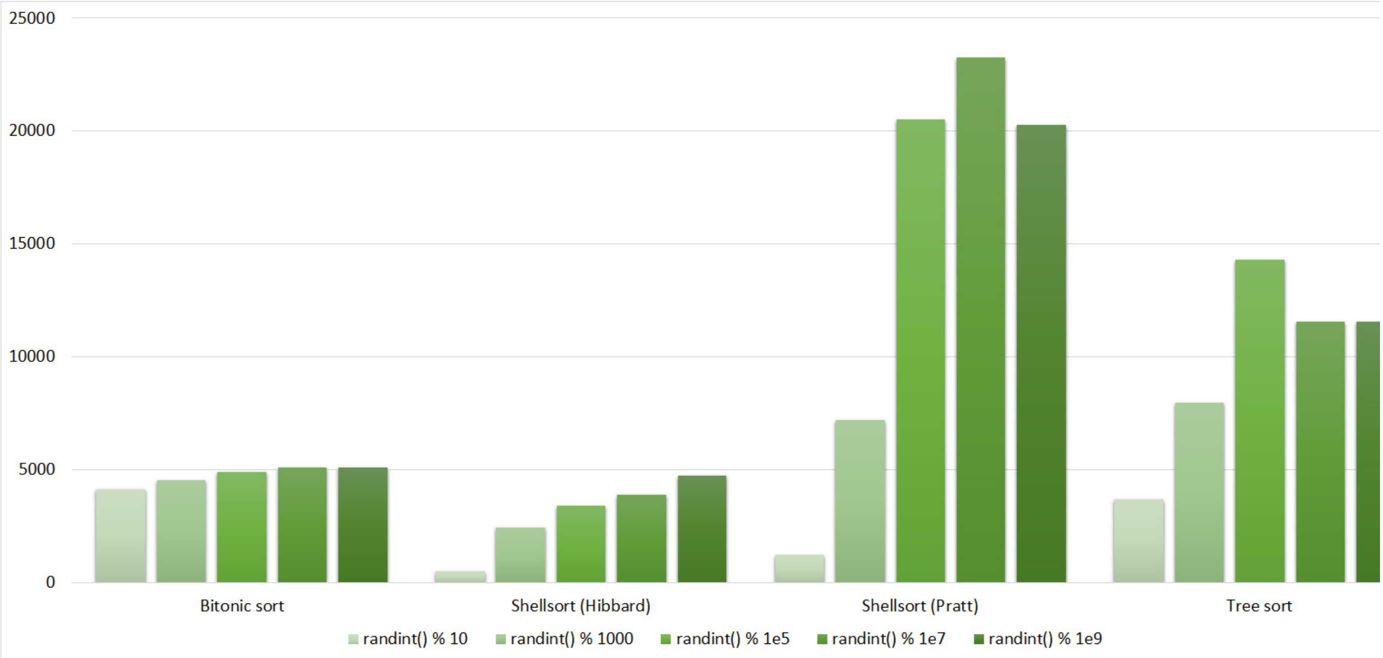


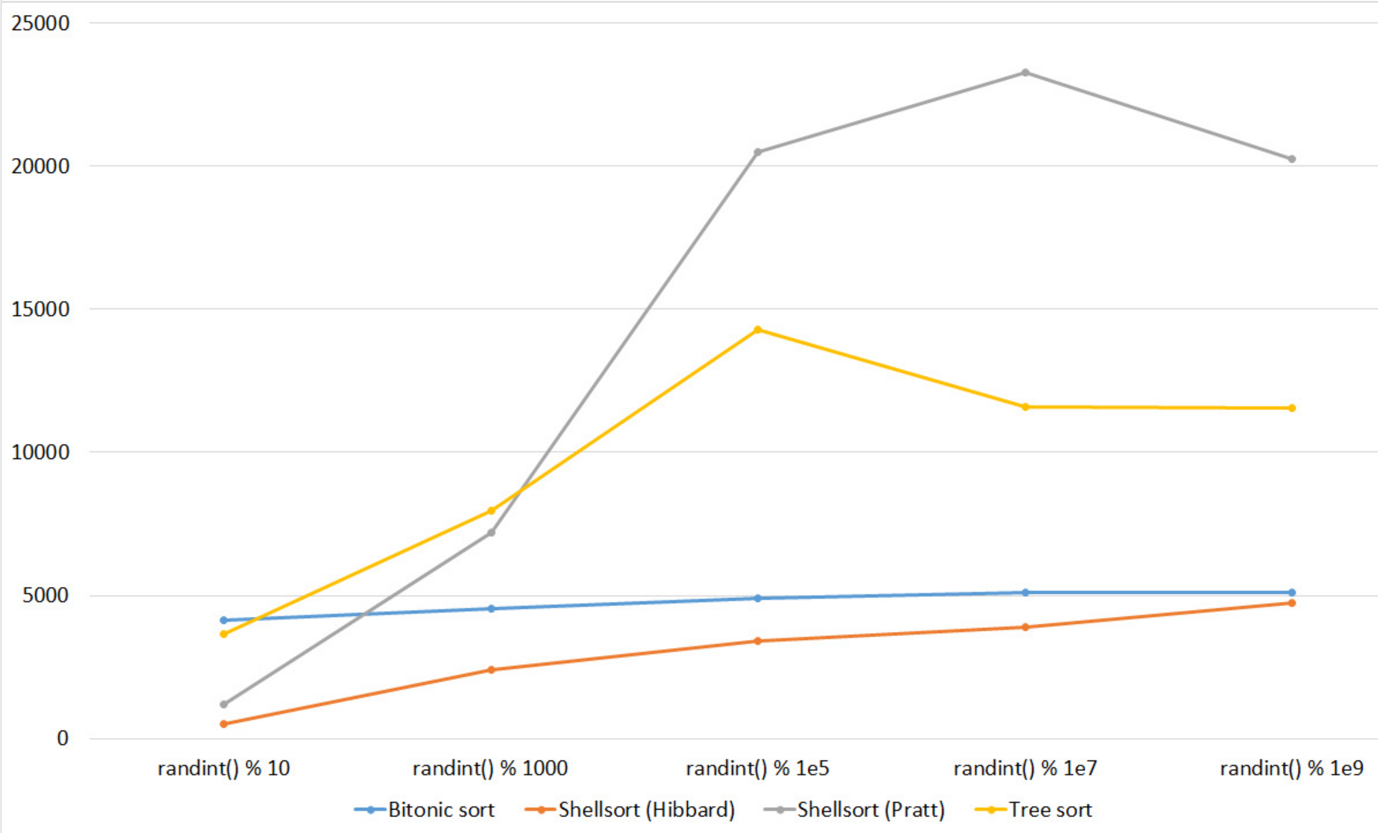
1e6	randint() % 10	randint() % 1000	randint() % 1e5	randint() % 1e7	randint() % 1e9	Итс
Bitonic sort	212,69	250,75	281,77	288,36	288,43	264,
Shellsort (Hibbard)	40,44	128,64	179,09	184,18	193,42	145,
Shellsort (Pratt)	86,22	665,72	712,22	641,15	750,44	571,
Tree sort	309,74	555,72	713,93	660,60	660,97	580,

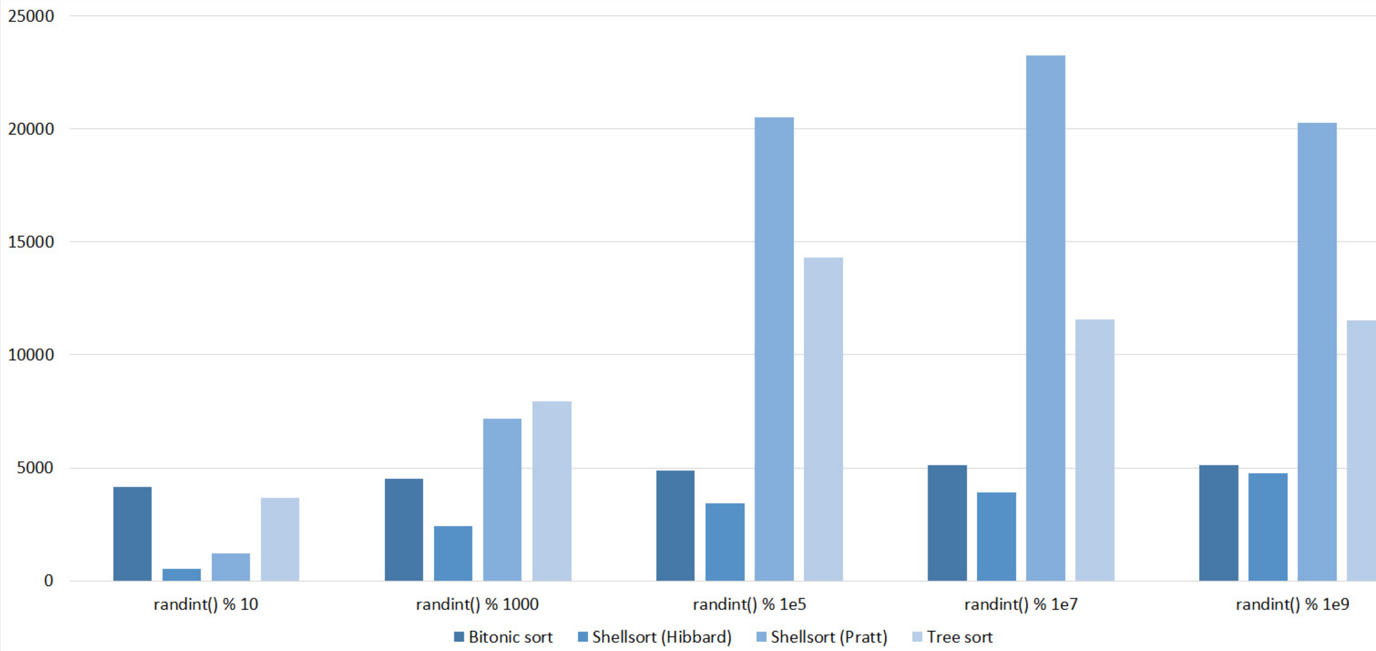
1e6	Итог
Shellsort (Hibbard)	145,15
Bitonic sort	264,40
Shellsort (Pratt)	571,15
Tree sort	580,19

Сортировка Шелла с последовательностью Пратта ведет себя совсем странно, остальные более менее ясно. Сортировка деревом любит частично отсортированные массивы, но не любит повторов, возможно, поэтому самое худшее время работы именно посередине.

Таблицы, 1e7 элементов


25000
20000
15000
10000
5000
0
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9


25000
20000
15000
10000
5000
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort



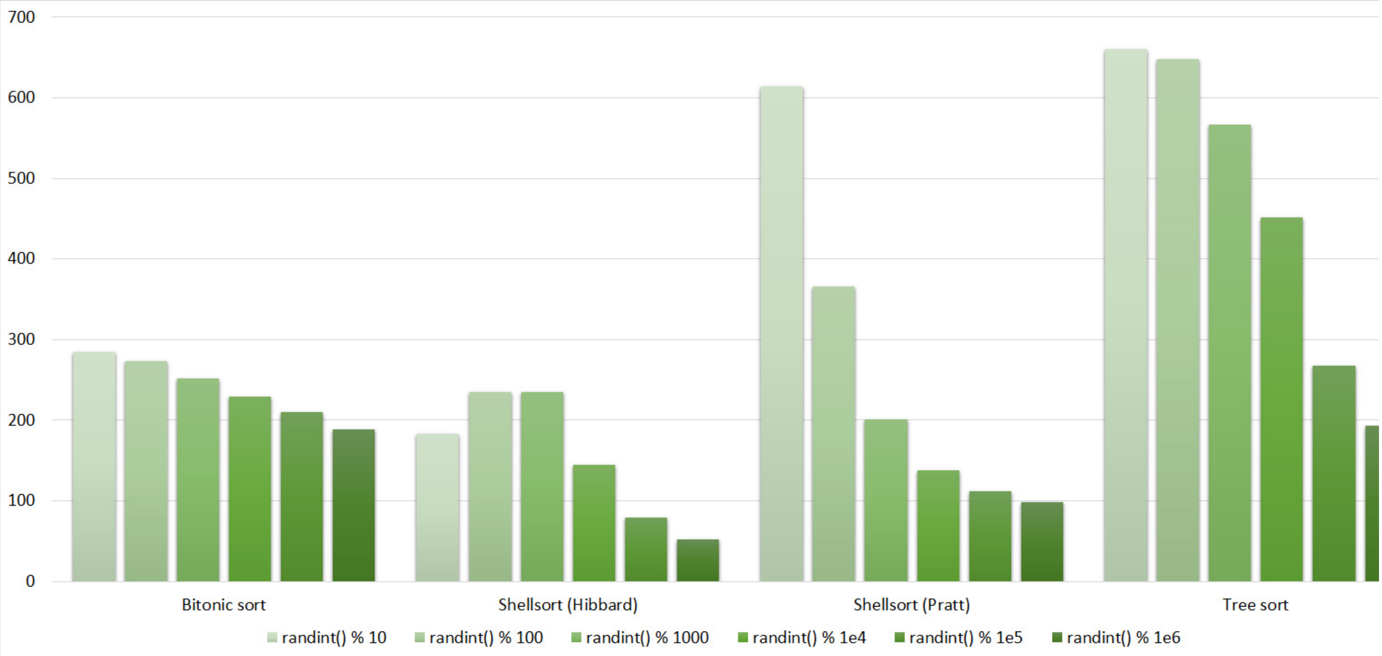
1e7	randint() % 10	randint() % 1000	randint() % 1e5	randint() % 1e7	randint() % 1e9	Итс
Bitonic sort	4142,81	4535,23	4898,90	5114,45	5112,47	4760
Shellsort (Hibbard)	513,70	2419,83	3423,91	3899,27	4743,87	2897
Shellsort (Pratt)	1204,63	7196,02	20504,91	23273,96	20267,04	1280
Tree sort	3668,15	7945,61	14303,92	11575,00	11550,99	4625

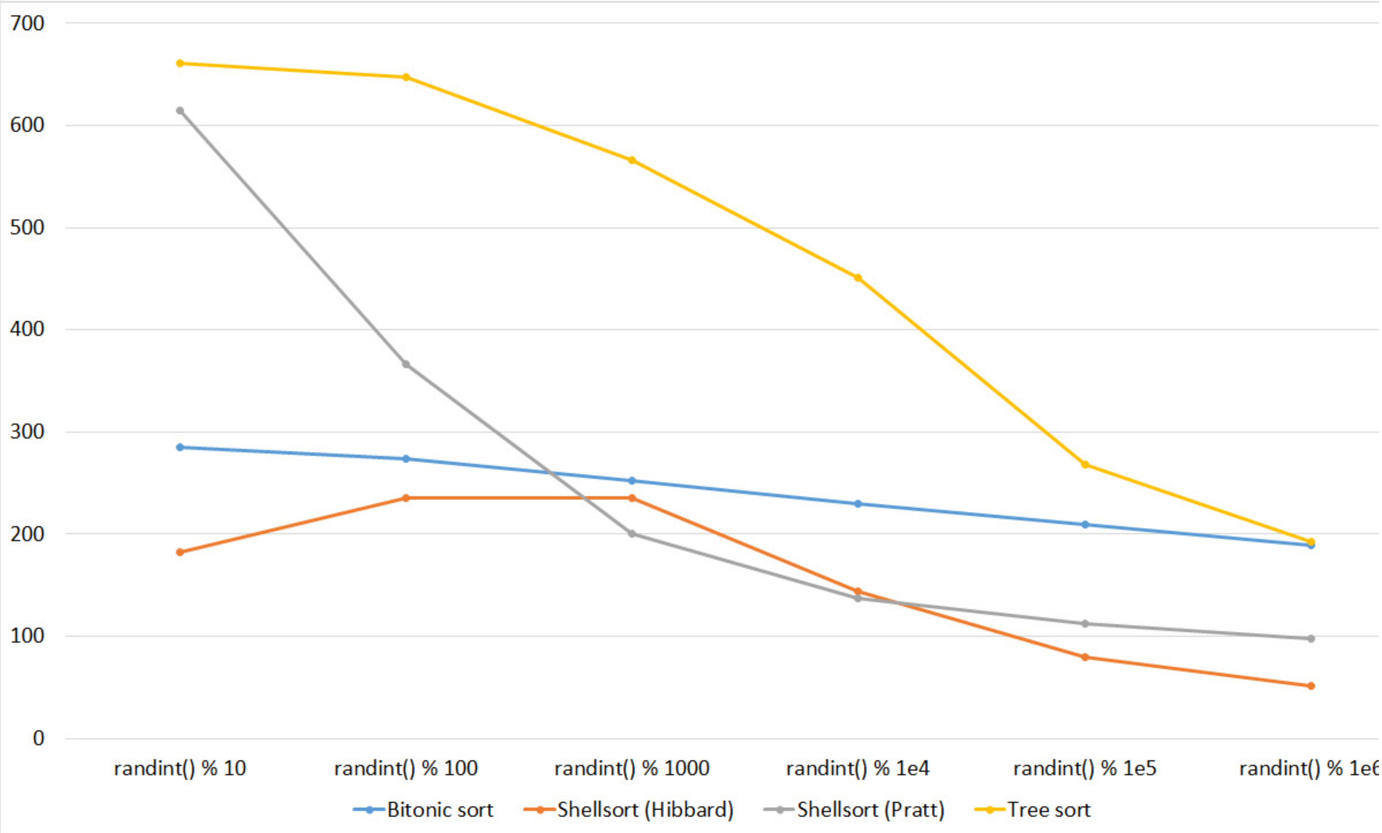
1e7	Итог
Shellsort (Hibbard)	2897,38
Tree sort	4625,20
Bitonic sort	4760,77
Shellsort (Pratt)	12809,18

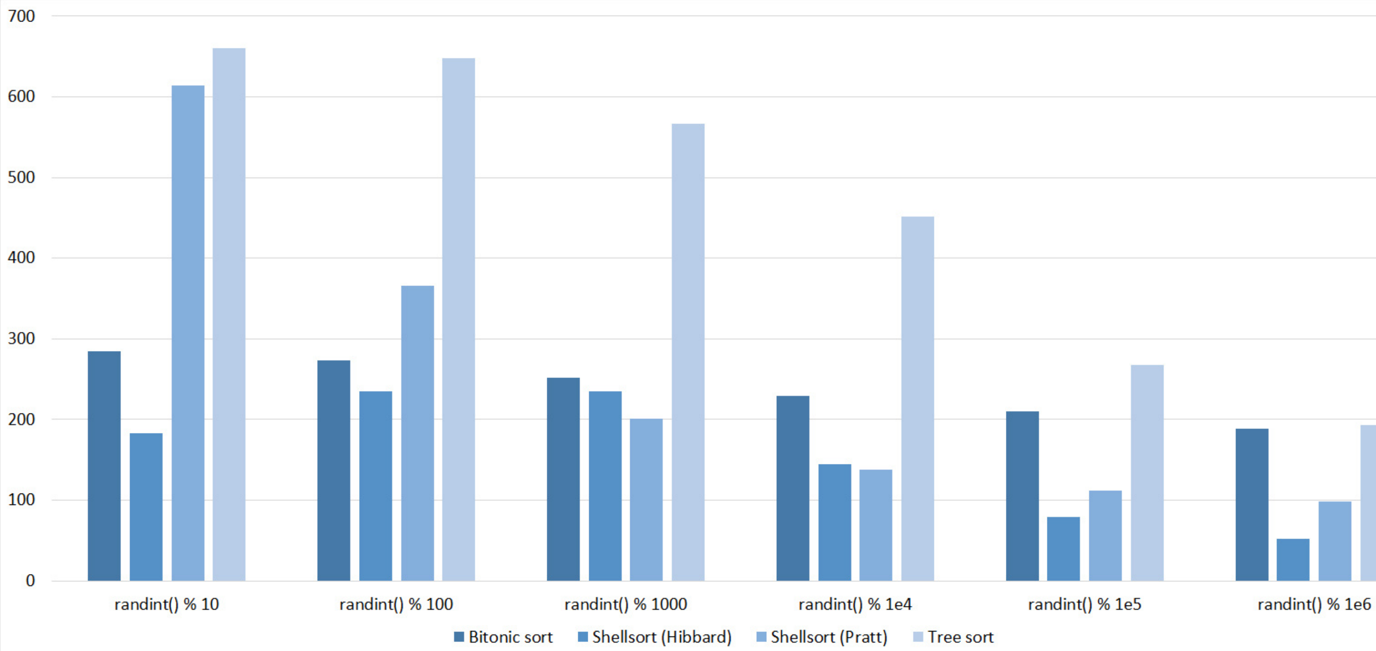
Все как прежде, только Шелл с Праттом усилился на второй группе из-за отсортированности. Также становится заметным влияние асимптс сортировка деревом оказывается на втором месте, в отличие от группы с меньшим числом элементов.

Частично отсортированный массив

Таблицы, 1e6 элементов


700
600
500
400
300
200
100
0
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6


700
600
500
400
300
200
100
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort

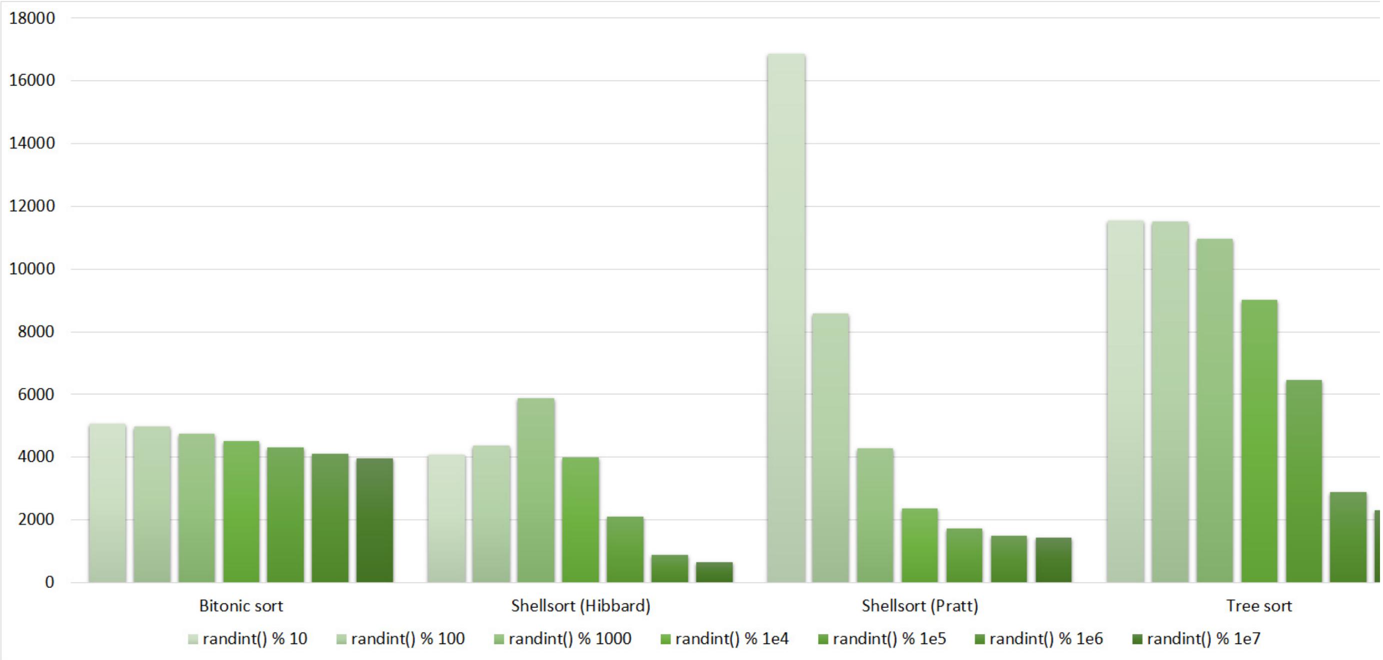


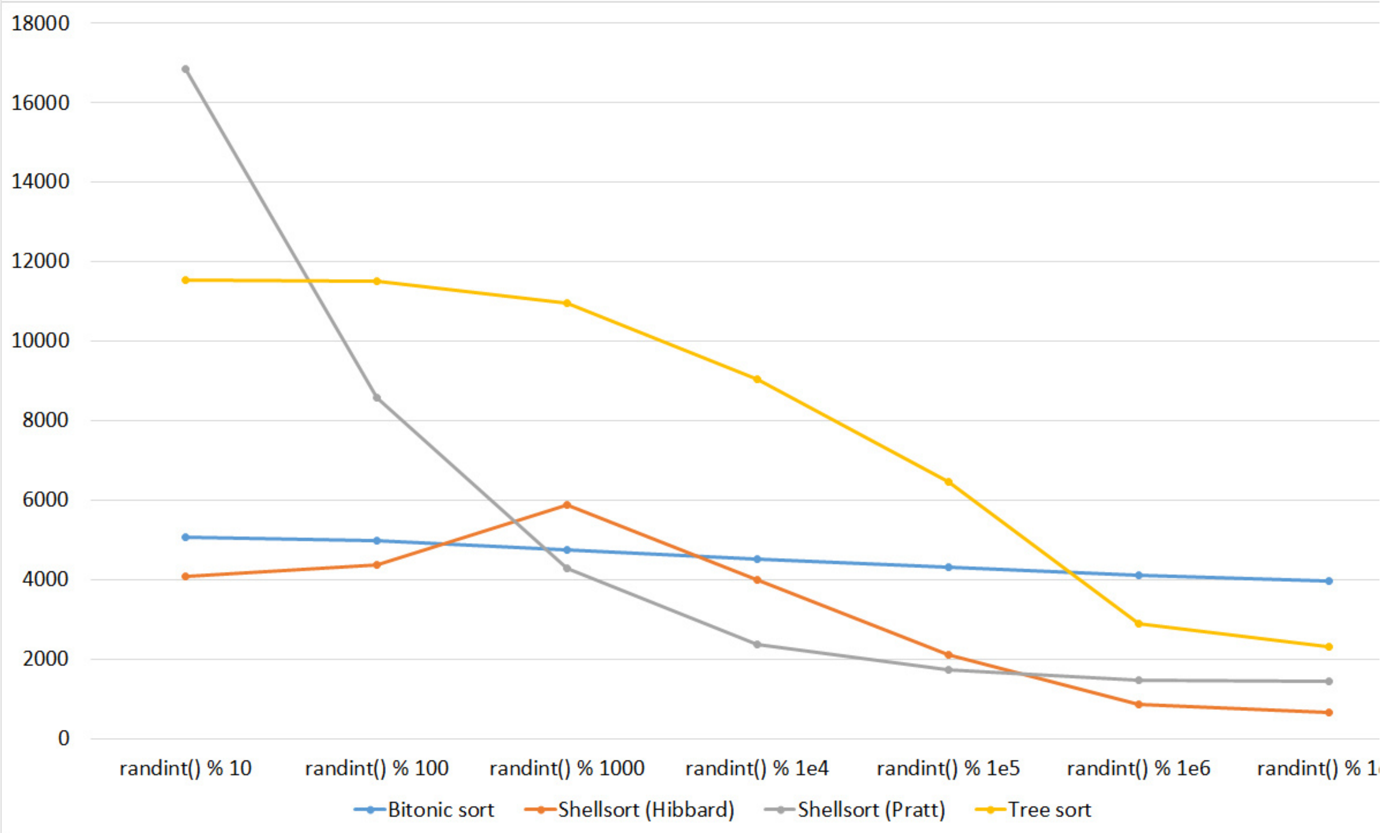
1e6	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	Ито
Bitonic sort	284,74	273,42	252,30	229,40	209,76	189,04	239,7
Shellsort (Hibbard)	182,39	235,38	235,24	144,30	79,25	51,84	154,7
Shellsort (Pratt)	614,20	366,26	200,46	137,63	112,23	98,25	254,8
Tree sort	660,62	647,91	566,22	451,39	267,69	192,94	464,4

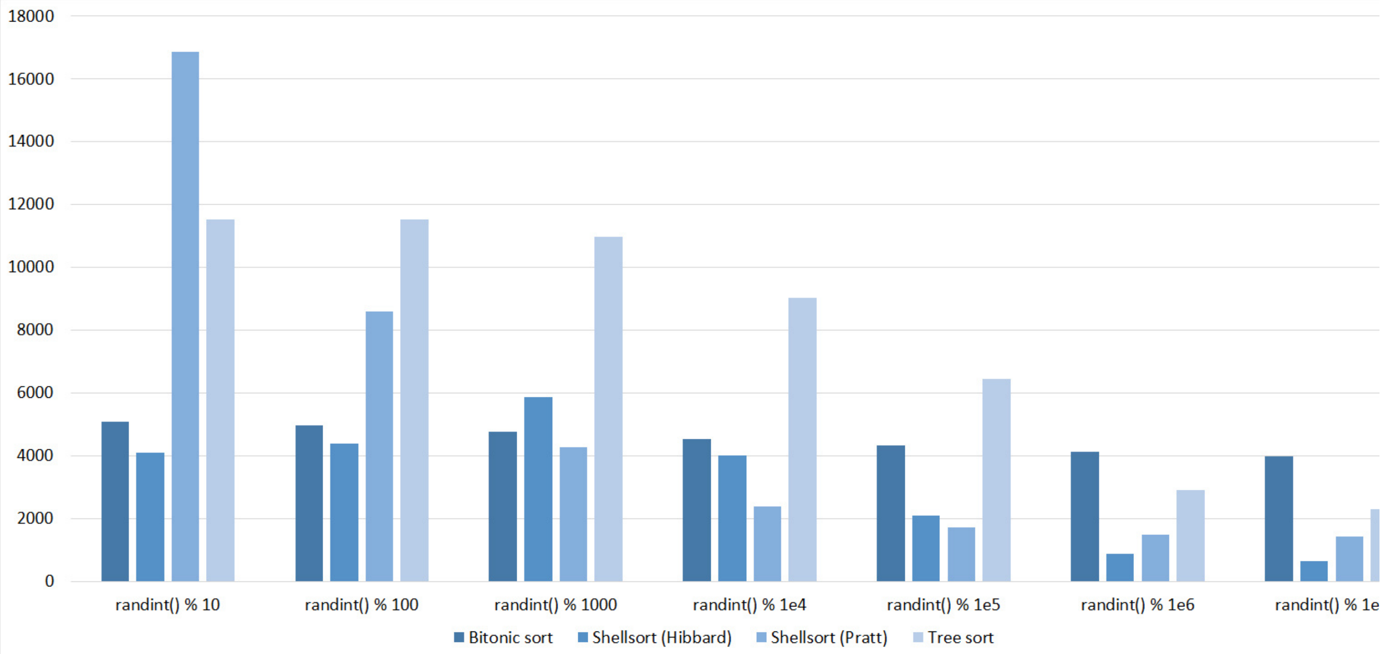
1e6	Итог
Shellsort (Hibbard)	154,73
Bitonic sort	239,78
Shellsort (Pratt)	254,84
Tree sort	464,46

Здесь понятным образом ведут себя все сортировки, кроме Шелла с Хиббардом, который почему-то не сразу начинает ускоряться.

Таблицы, 1e7 элементов


18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e7


18000
16000
14000
12000
10000
8000
6000
4000
2000
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort



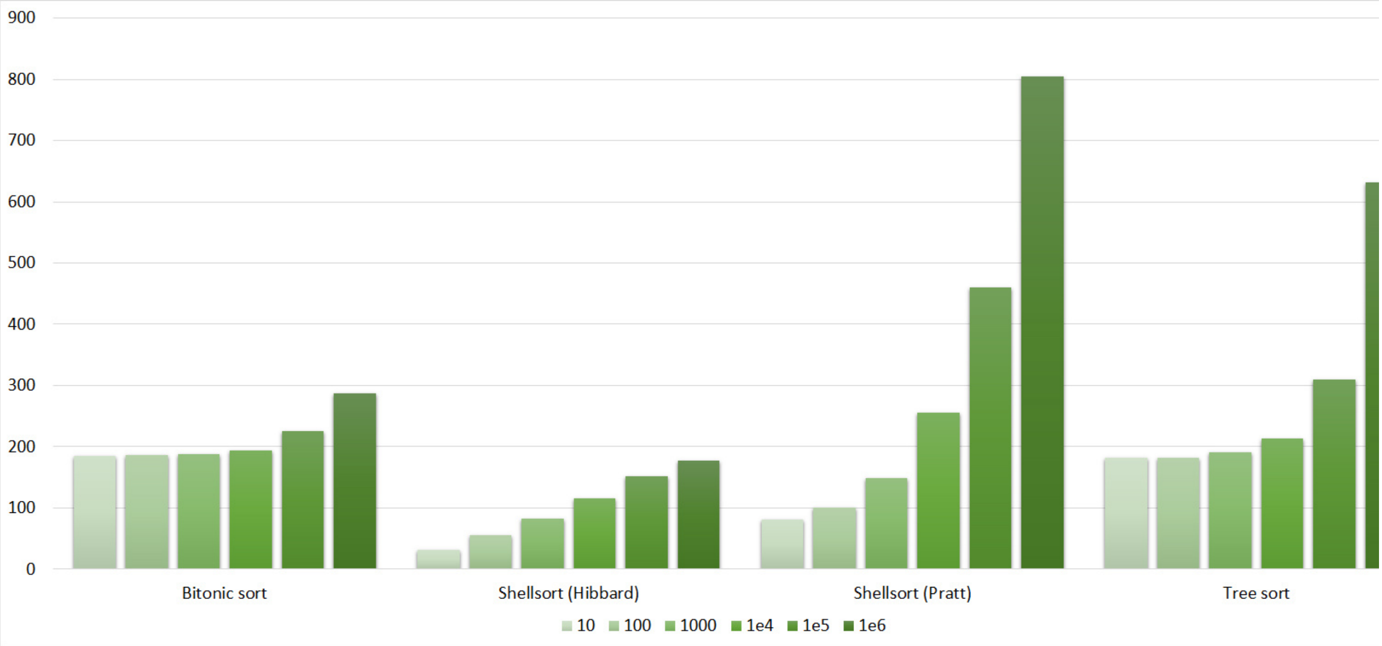
1e7	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	Ито
Bitonic sort	5079,56	4965,59	4752,93	4527,58	4316,46	4116,9	3970,69	4532,
Shellsort (Hibbard)	4081,35	4380,06	5870,64	3997,01	2099,58	867,4	654,58	3135,
Shellsort (Pratt)	16859,51	8587,82	4275,49	2376,36	1727,98	1480,82	1435,99	5249,
Tree sort	11538,4	11510,68	10971,49	9029,98	6447,13	2892,82	2303,1	7813,

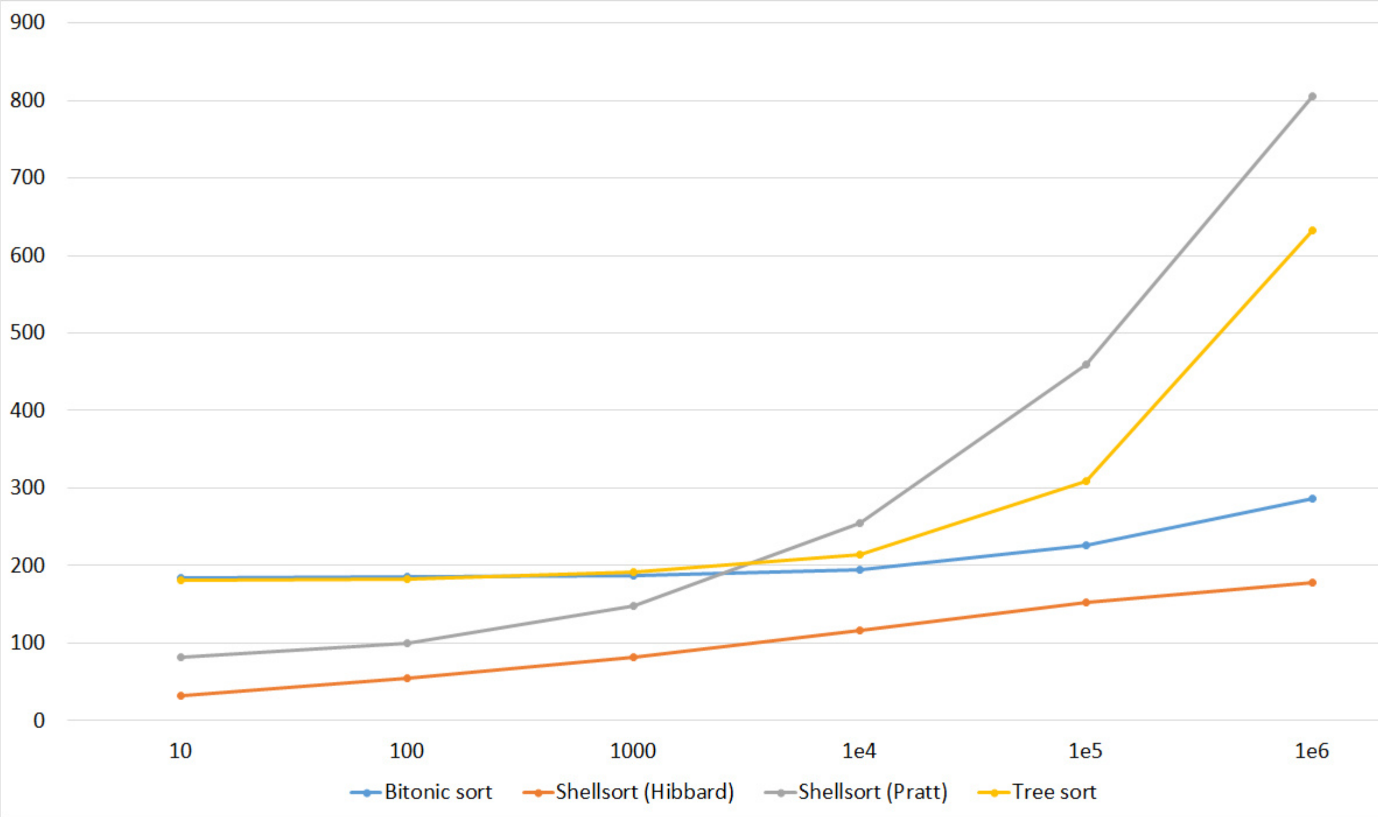
1e7	Итог
Shellsort (Hibbard)	3135,80
Bitonic sort	4532,82
Shellsort (Pratt)	5249,14
Tree sort	7813,37

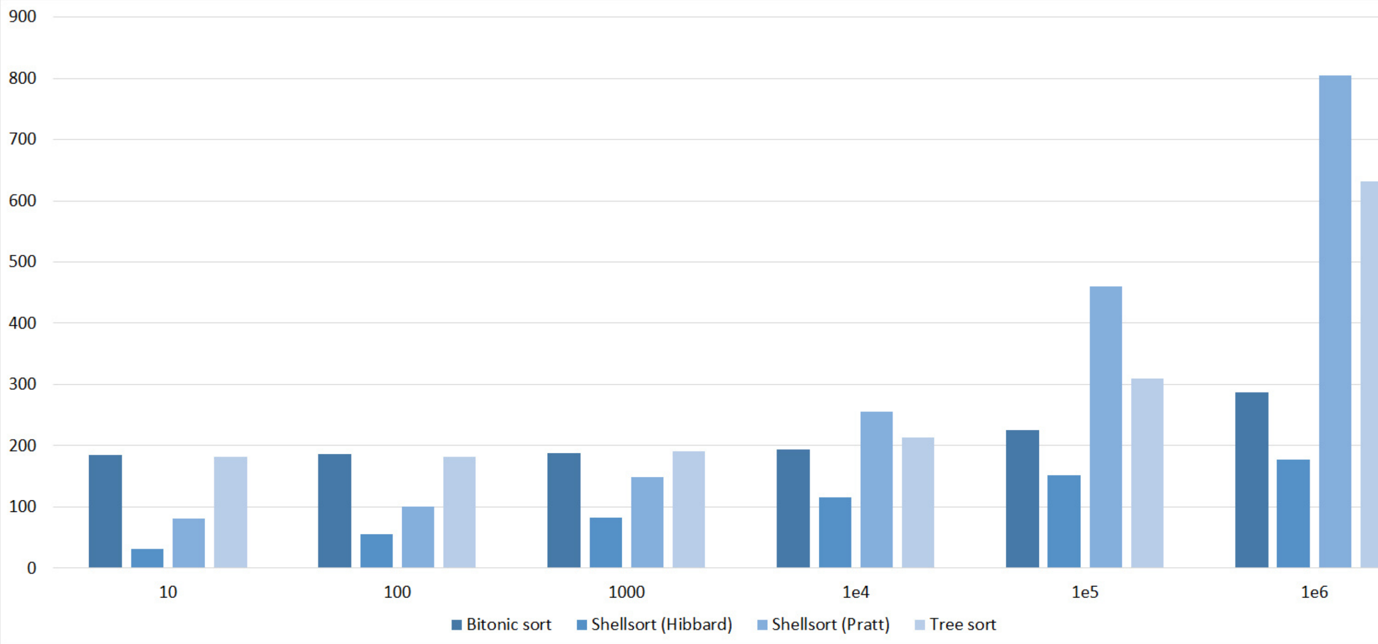
Здесь все, в общем, как и прежде. Даже асимптотика сортировки деревом не помогла ей вырваться с последнего места.

Свопы

Таблицы, 1e6 элементов


900
800
700
600
500
400
300
200
100
0
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort
10
100
1000
1e4
1e5
1e6

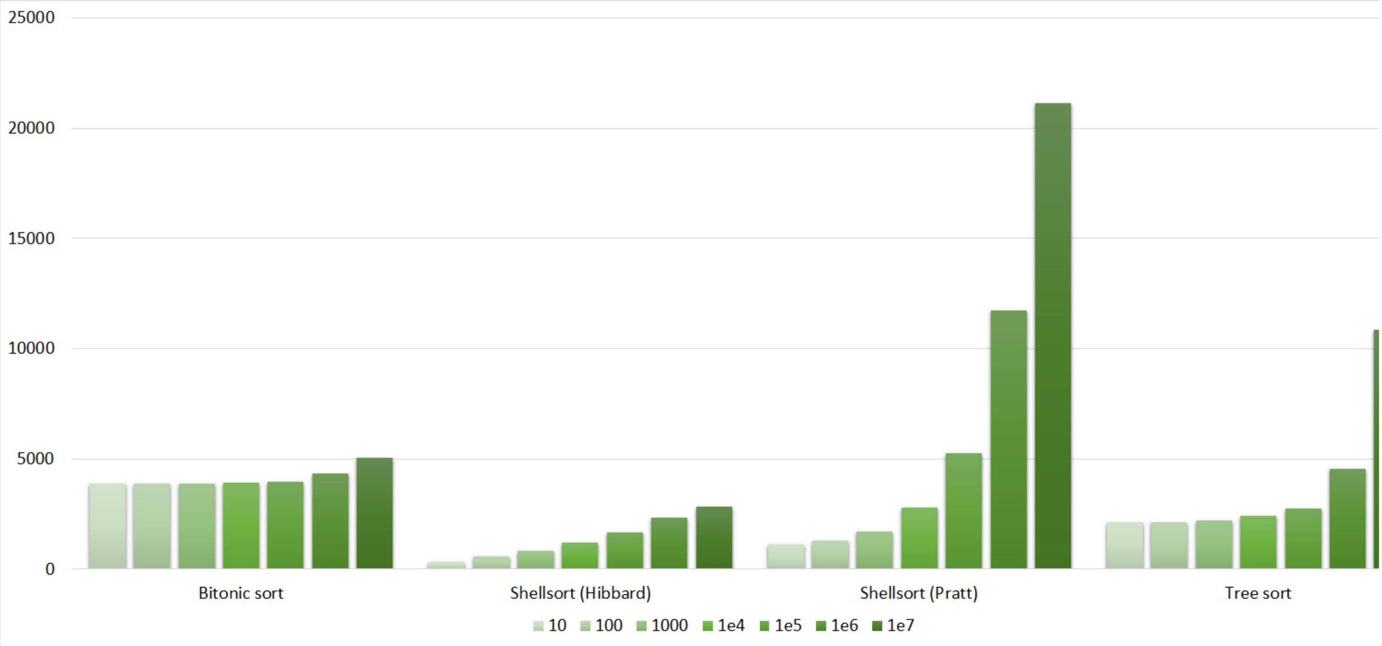

900
800
700
600
500
400
300
200
100
0
10
100
1000
1e4
1e5
1e6
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort

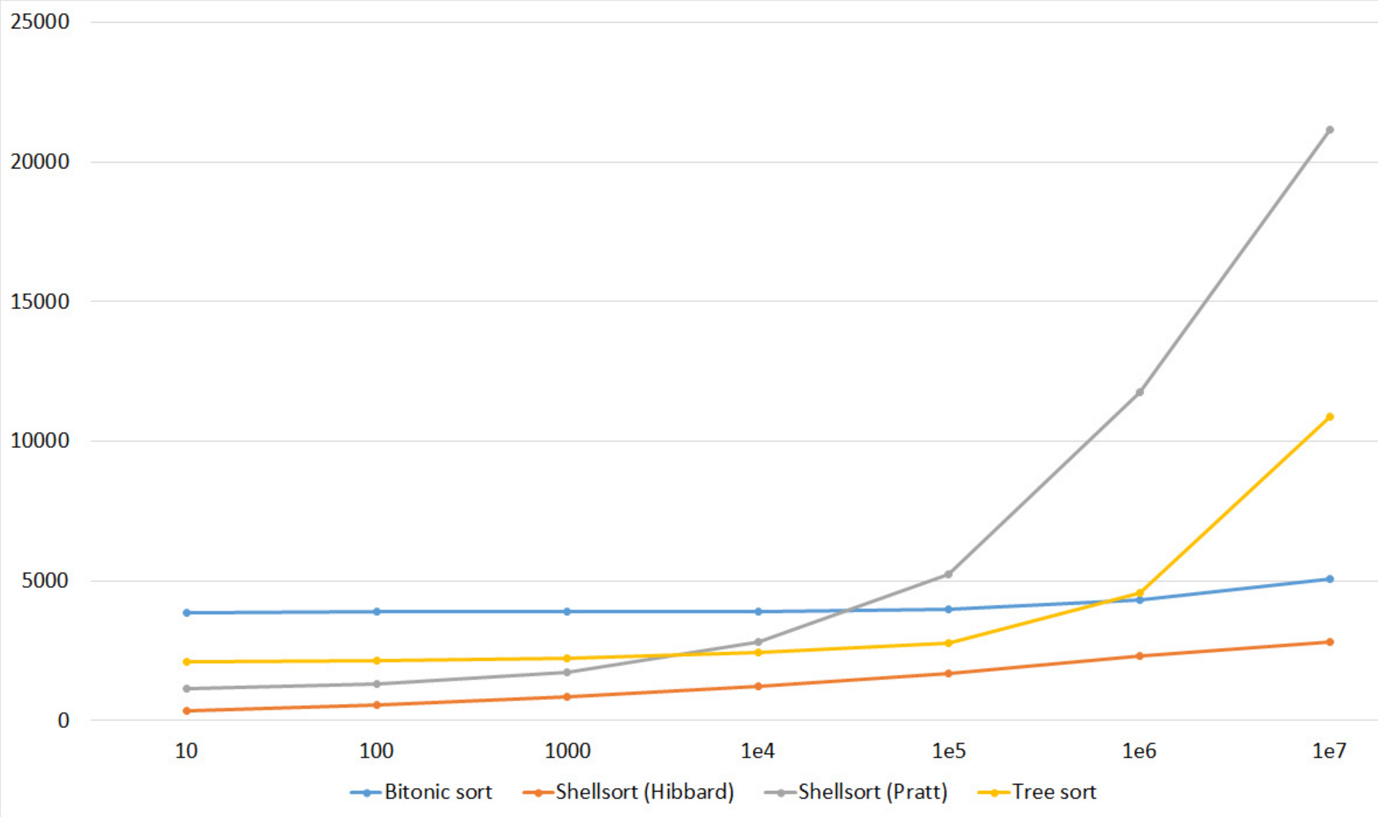


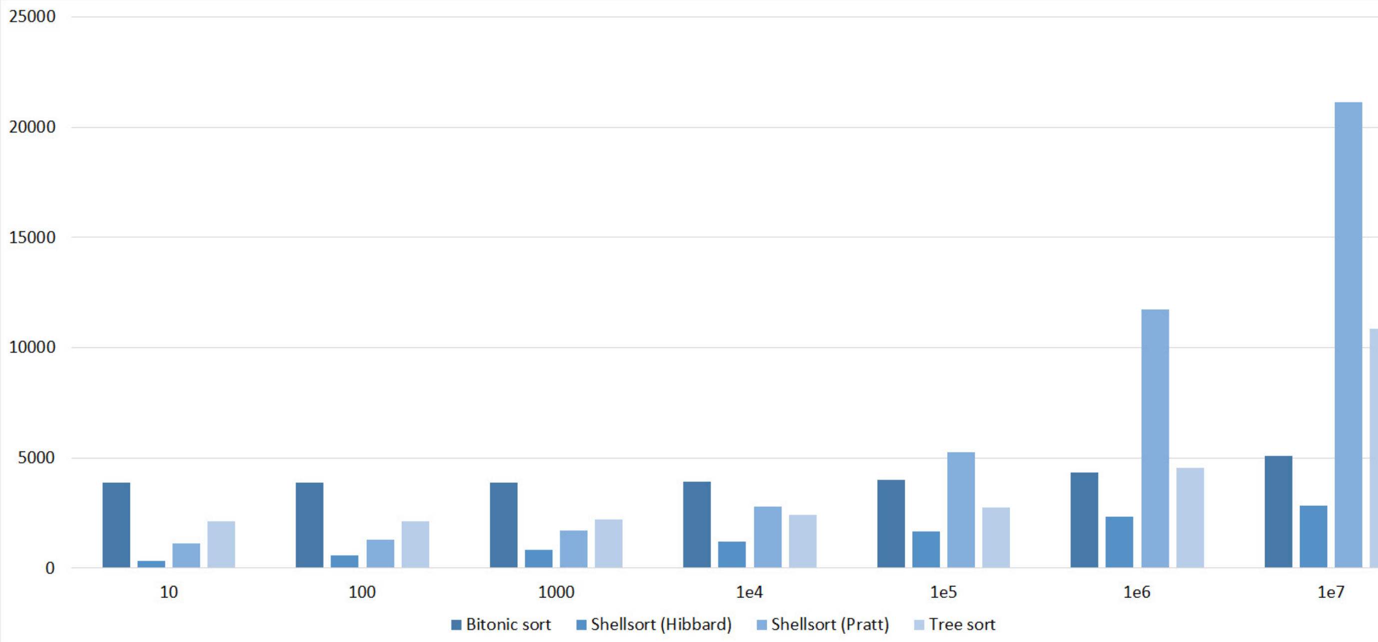
1e6	10	100	1000	1e4	1e5	1e6	Ито
Bitonic sort	184,63	185,82	187,81	194,21	225,52	286,94	210,
Shellsort (Hibbard)	31,73	55,1	82,11	115,86	152,18	177,53	102,
Shellsort (Pratt)	81,4	100,14	148,33	255,16	460,09	805,07	308,
Tree sort	181,55	182,52	191,25	213,58	309,39	631,73	285,

1e6	Итог
Shellsort (Hibbard)	102,42
Bitonic sort	210,82
Tree sort	285,00
Shellsort (Pratt)	308,37

Таблицы, 1e7 элементов


25000
20000
15000
10000
5000
0
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort
10 100 1000 1e4 1e5 1e6 1e7


25000
20000
15000
10000
5000
0
10 100 1000 1e4 1e5 1e6 1e7
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort



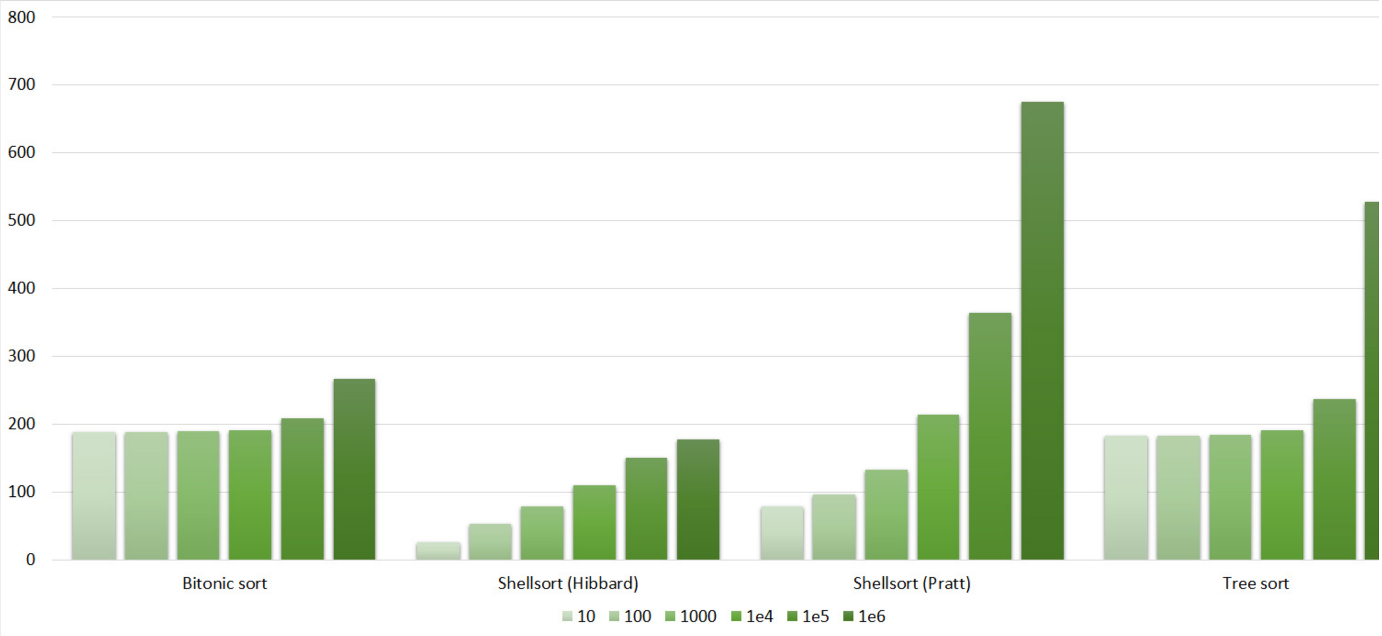
1e7	10	100	1000	1e4	1e5	1e6	1e7	Итог
Bitonic sort	3872,94	3882,51	3890,98	3909,03	3986,53	4326,05	5073,78	4134,55
Shellsort (Hibbard)	339,61	576,06	849,77	1210,91	1681,56	2317,66	2818,3	1399,12
Shellsort (Pratt)	1126,25	1309,64	1711,42	2799	5255,05	11740,64	21161,45	6443,35
Tree sort	2122,91	2135,33	2221,78	2425,01	2759,77	4570,11	10867,81	3871,82

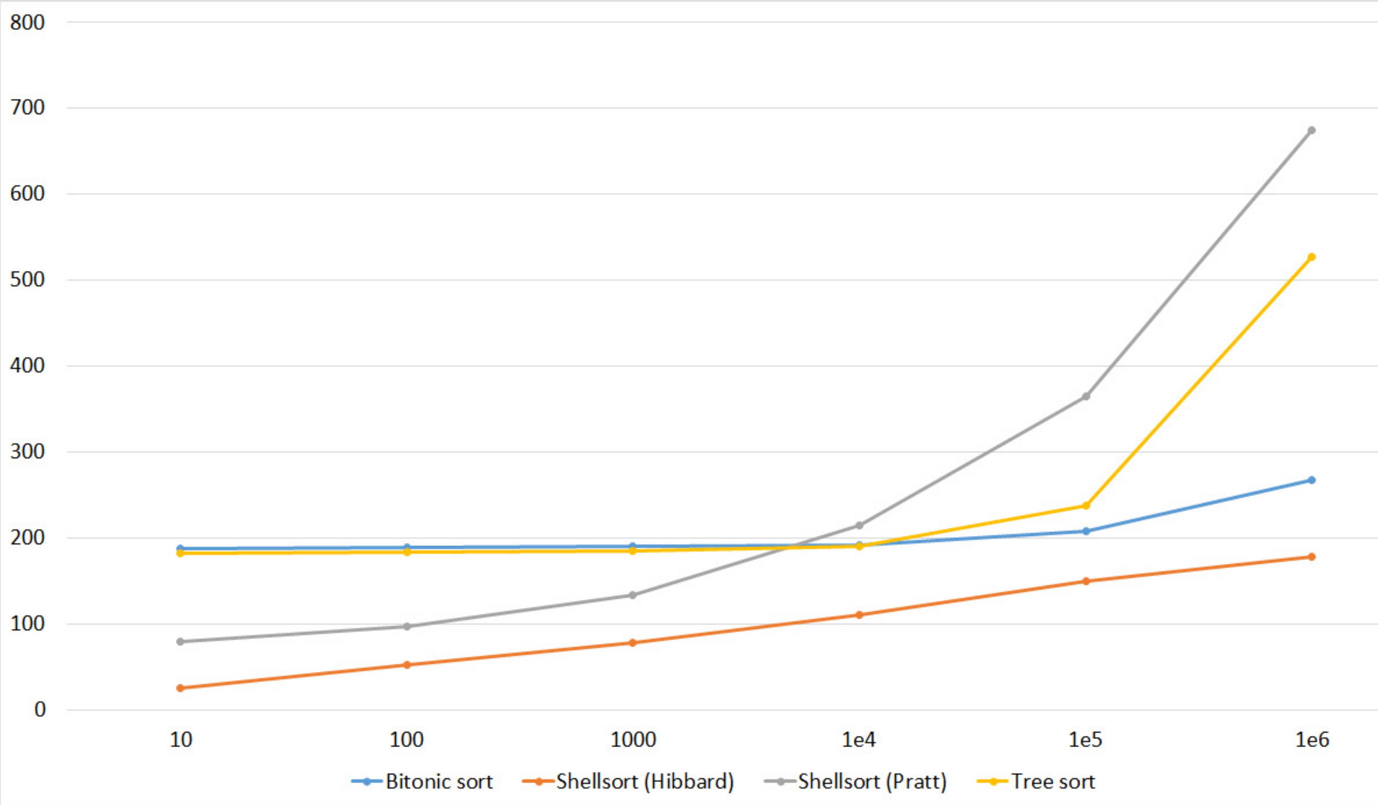
1e7	Итог
Shellsort (Hibbard)	1399,12
Tree sort	3871,82
Bitonic sort	4134,55
Shellsort (Pratt)	6443,35

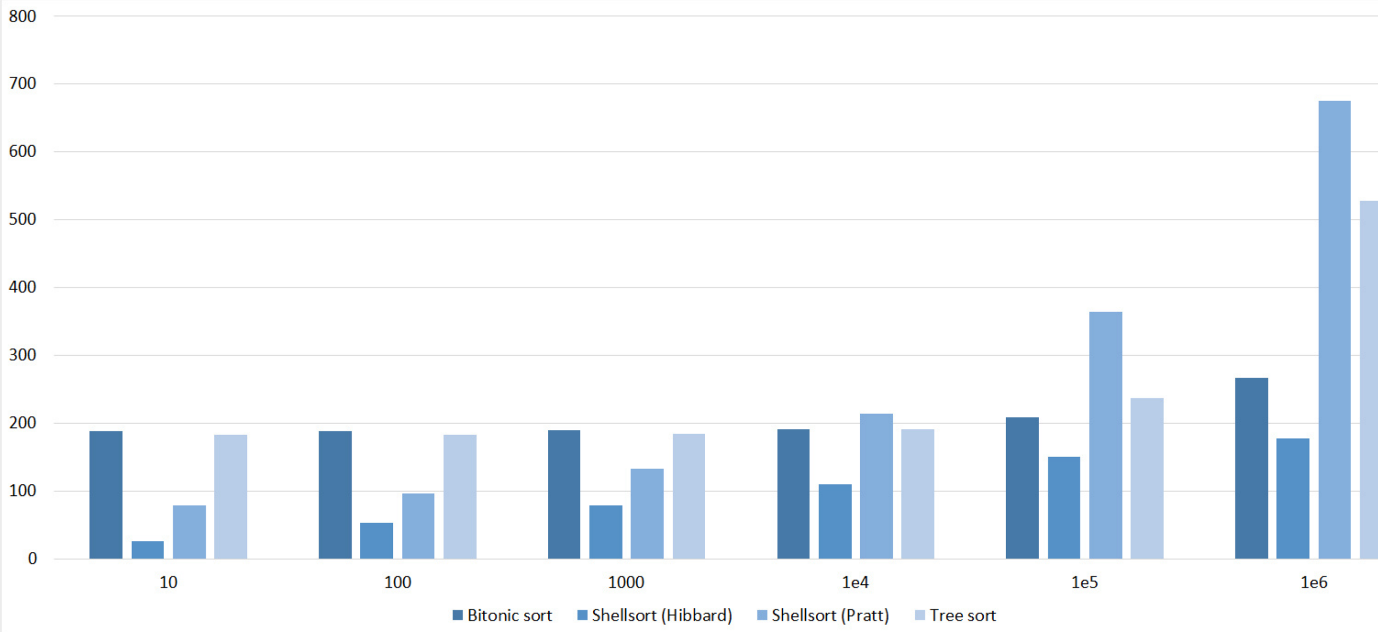
Здесь заметно, что у сортировок Шелла большая зависимость от частичной отсортированности, так как они ведут себя практически линейн остальные две только сильно падают на последних группах.

Изменения в перестановке

Таблицы, 1e6 элементов


800
700
600
500
400
300
200
100
0
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort
10 100 1000 1e4 1e5 1e6

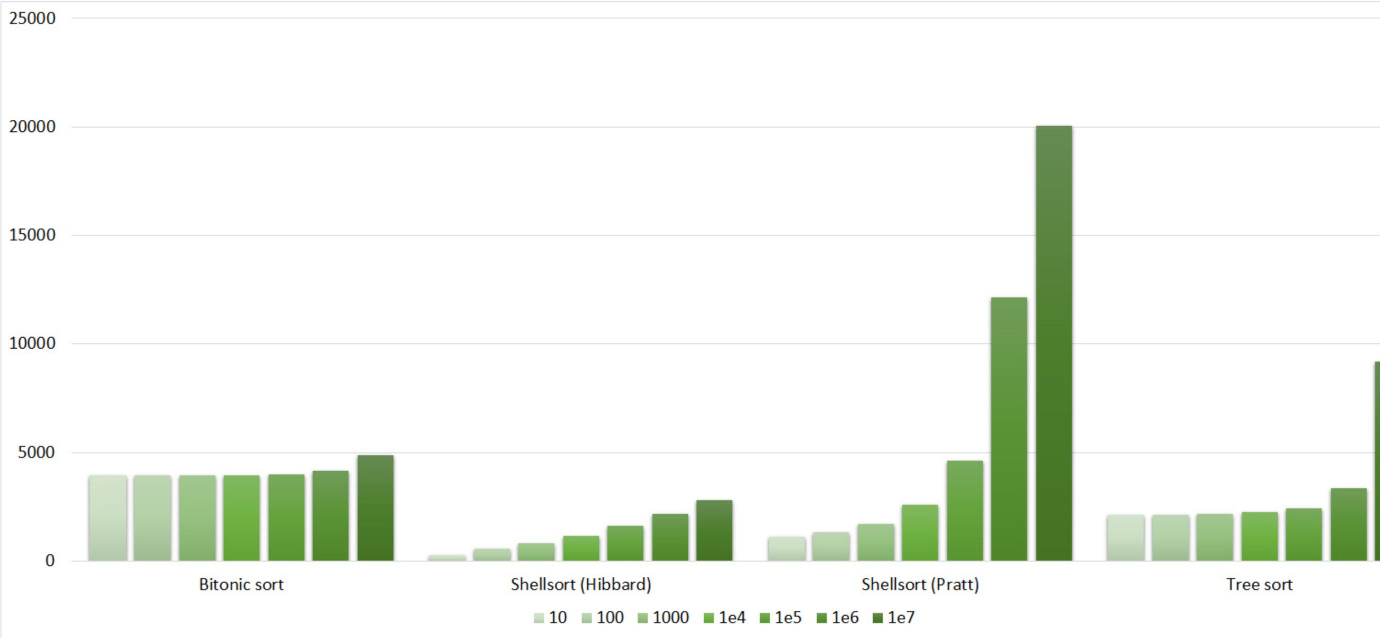

800
700
600
500
400
300
200
100
0
10
100
1000
1e4
1e5
1e6
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort

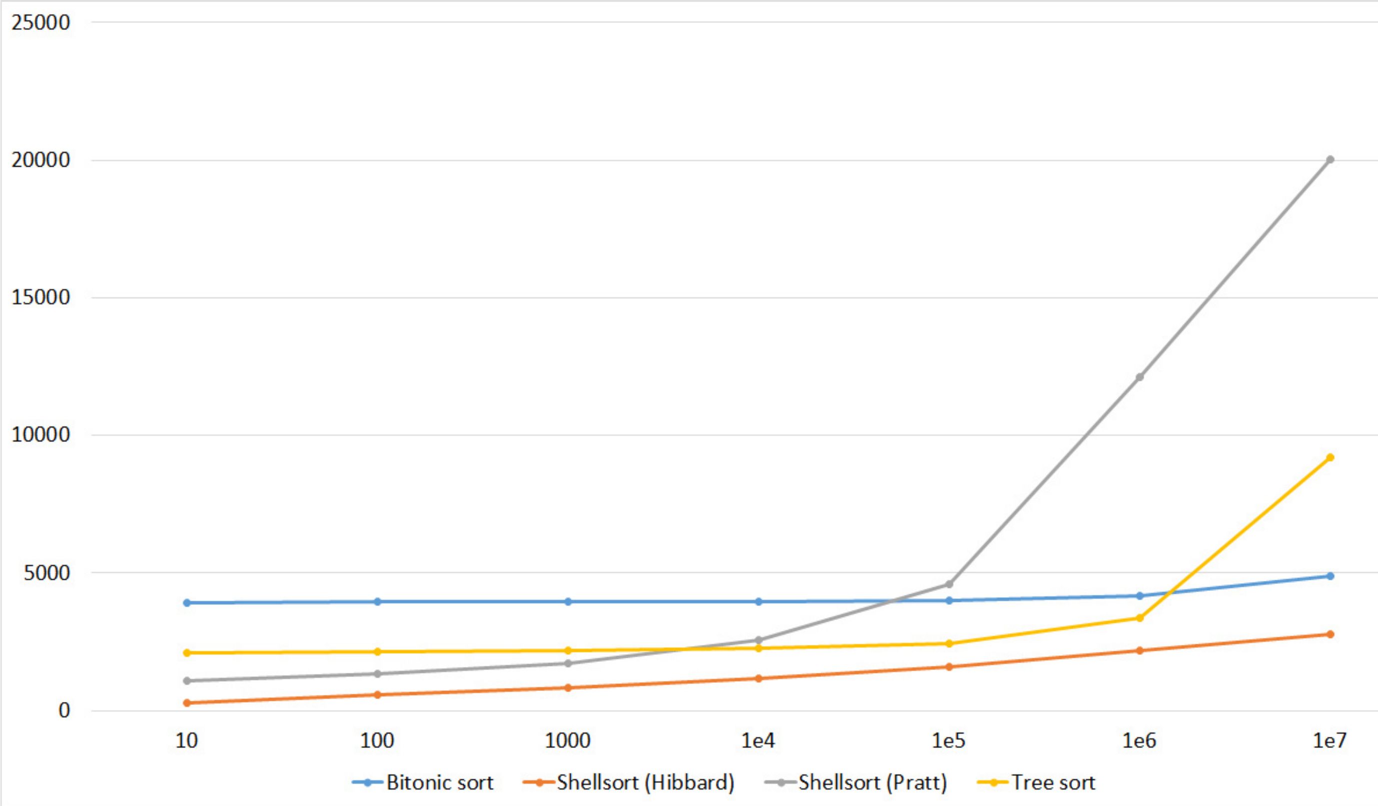


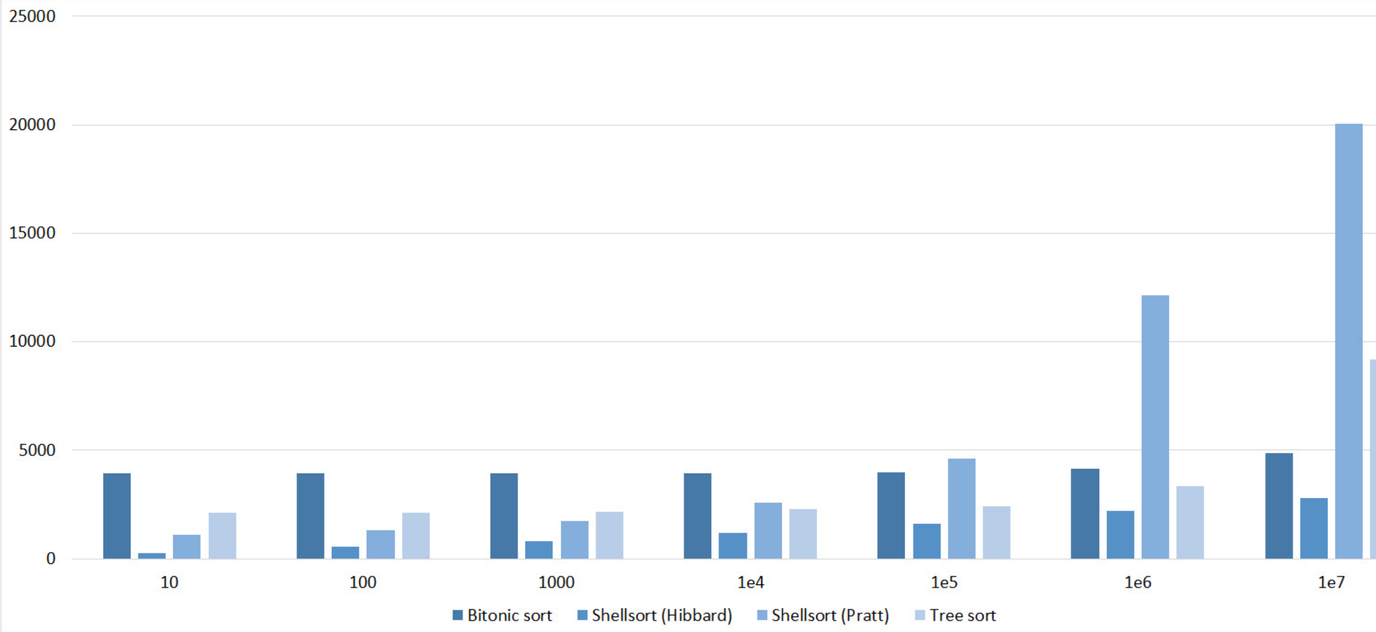
1e6	10	100	1000	1e4	1e5	1e6	Ито
Bitonic sort	188,15	188,375	190,15	191,45	207,8	267,325	205,5
Shellsort (Hibbard)	25,55	52,775	78,275	110,35	150,25	177,6	99,1
Shellsort (Pratt)	78,65	96,5	133,15	214,375	364,1	675,15	260,3
Tree sort	182,475	183,2	184,725	190,7	237,1	527,375	250,9

1e6	Итог
Shellsort (Hibbard)	99,13
Bitonic sort	205,54
Tree sort	250,93
Shellsort (Pratt)	260,32

Таблицы, 1e7 элементов


25000
20000
15000
10000
5000
0
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort
10
100
1000
1e4
1e5
1e6
1e7


25000
20000
15000
10000
5000
0
10
100
1000
1e4
1e5
1e6
1e7
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort



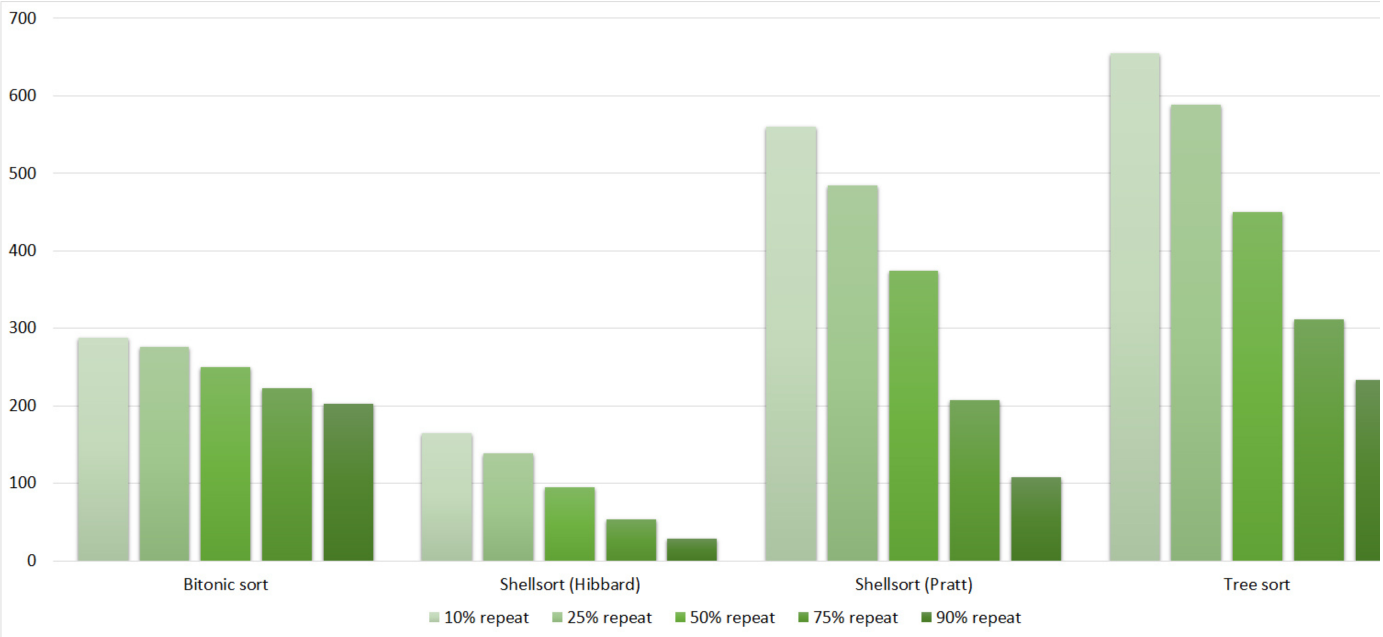
1e7	10	100	1000	1e4	1e5	1e6	1e7	Ито
Bitonic sort	3934,95	3947,15	3946,88	3955,33	3983,98	4161,30	4886,63	4116,
Shellsort (Hibbard)	262,88	554,90	826,98	1175,88	1601,88	2191,23	2792,70	1343,
Shellsort (Pratt)	1096,68	1324,85	1725,10	2576,08	4609,78	12137,80	20039,53	6215,
Tree sort	2112,40	2126,73	2178,73	2271,65	2428,20	3350,23	9198,38	3380,

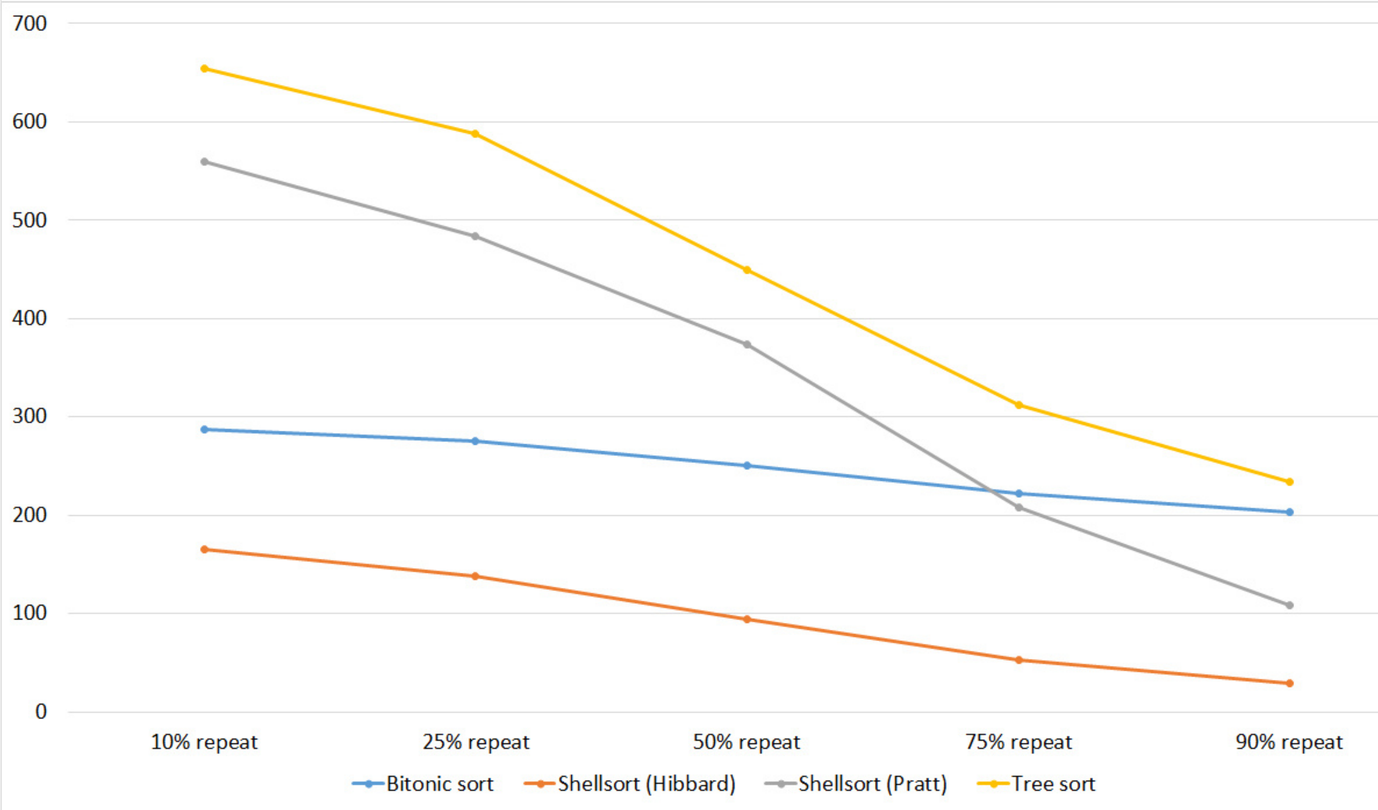
1e7	Итог
Shellsort (Hibbard)	1343,78
Tree sort	3380,90
Bitonic sort	4116,60
Shellsort (Pratt)	6215,69

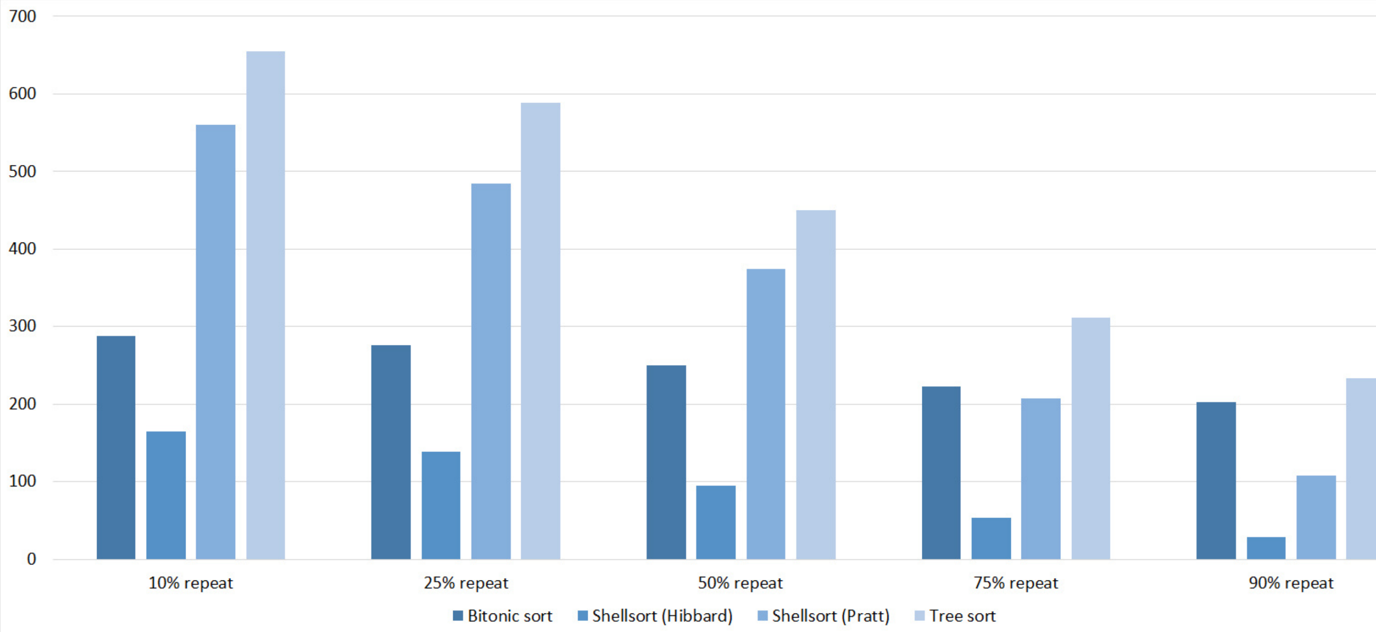
Здесь все похоже на предыдущую группу.

Повторы

Таблицы, 1e6 элементов


700
600
500
400
300
200
100
0
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat


700
600
500
400
300
200
100
0
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort

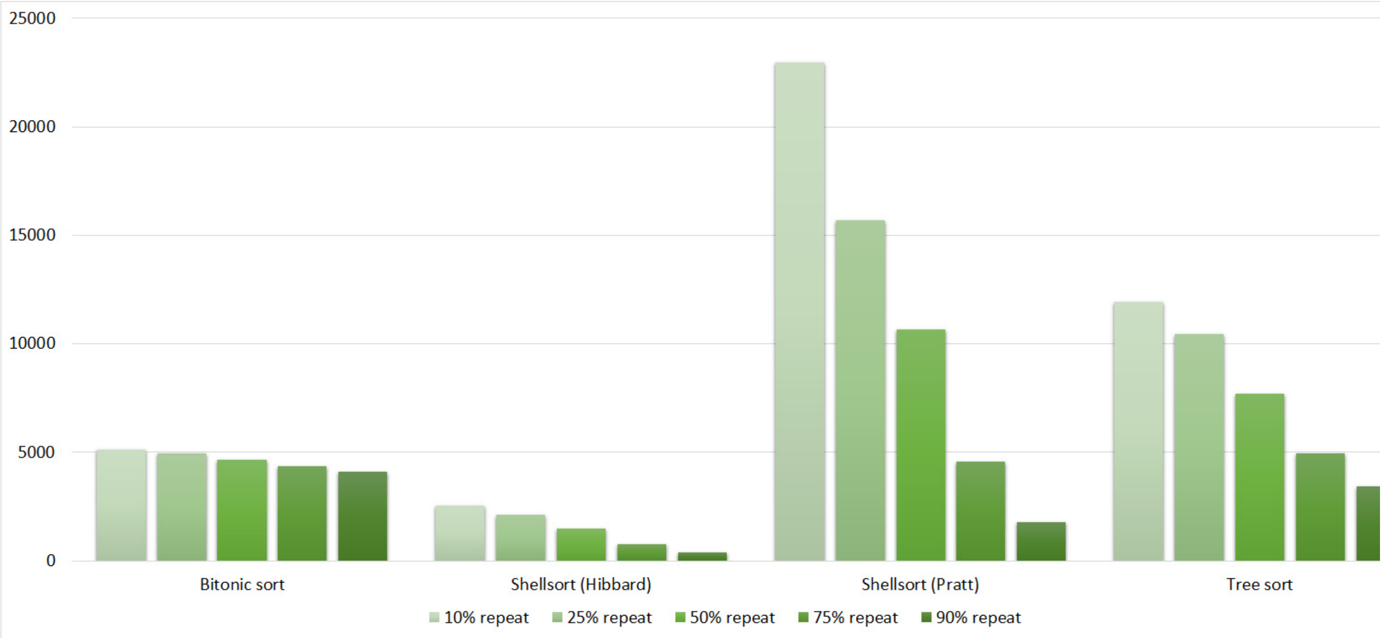


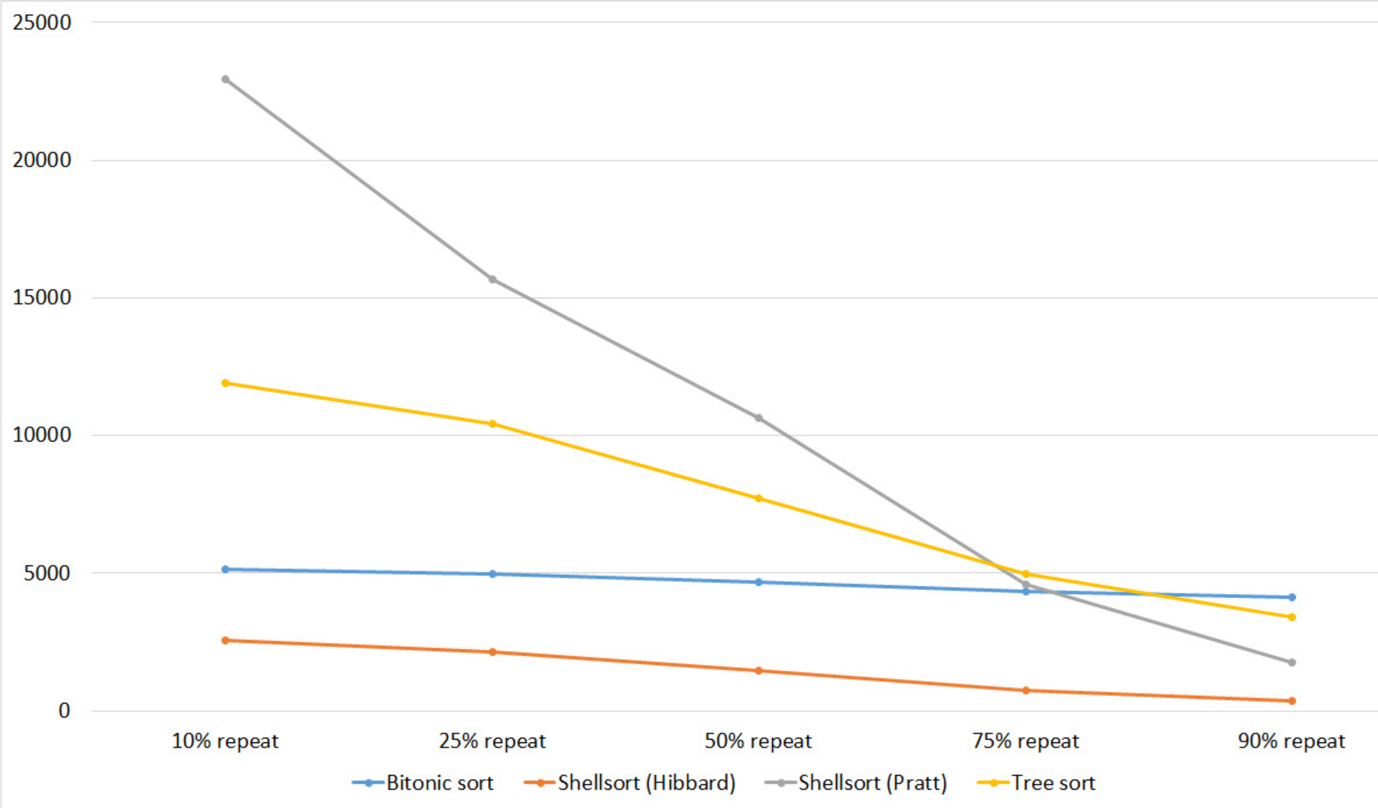
1e6	10% repeat	25% repeat	50% repeat	75% repeat	90% repeat	Итог
Bitonic sort	287,75	275,57	250,42	222,23	202,95	247,7
Shellsort (Hibbard)	165,07	138,03	94,35	53,22	29,03	95,94
Shellsort (Pratt)	559,67	484,00	373,97	207,48	107,93	346,6
Tree sort	654,40	588,23	449,68	311,48	233,35	447,4

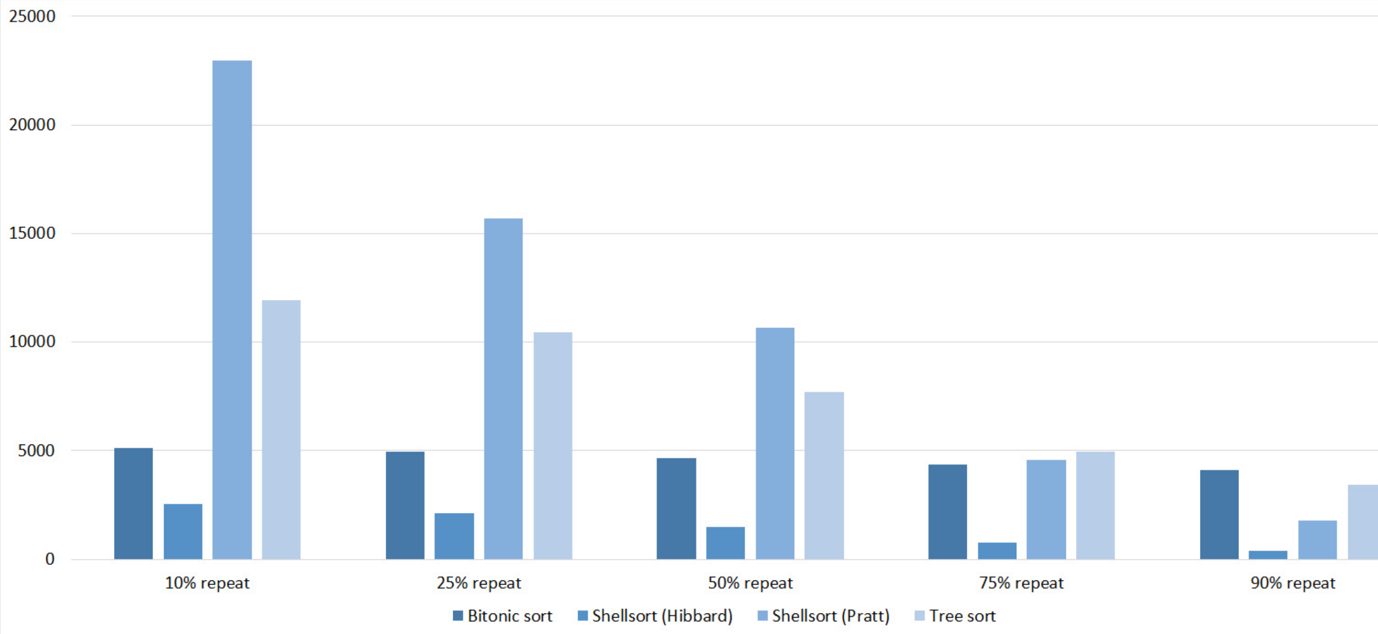
1e6	Итог
Shellsort (Hibbard)	95,94
Bitonic sort	247,78
Shellsort (Pratt)	346,61
Tree sort	447,43

Опять все сортировки продемонстрировали удивительную сбалансированность, даже битонная, которая, казалось бы, почти не зависит от массива.

Таблицы, 1e7 элементов


25000
20000
15000
10000
5000
0
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat


25000
20000
15000
10000
5000
0
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat
Bitonic sort
Shellsort (Hibbard)
Shellsort (Pratt)
Tree sort



1e7	10% repeat	25% repeat	50% repeat	75% repeat	90% repeat	Итог
Bitonic sort	5120,95	4966,55	4665,42	4337,53	4108,82	4639,8
Shellsort (Hibbard)	2542,32	2132,07	1475,77	757,82	379,47	1457,4
Shellsort (Pratt)	22943,85	15671,03	10653,87	4588,87	1757,48	11123,(
Tree sort	11917,65	10433,40	7719,57	4954,23	3410,65	7687,1

1e7	Итог
Shellsort (Hibbard)	1457,49
Bitonic sort	4639,85
Tree sort	7687,10
Shellsort (Pratt)	11123,02

Ничего интересного.

Итоговые результаты

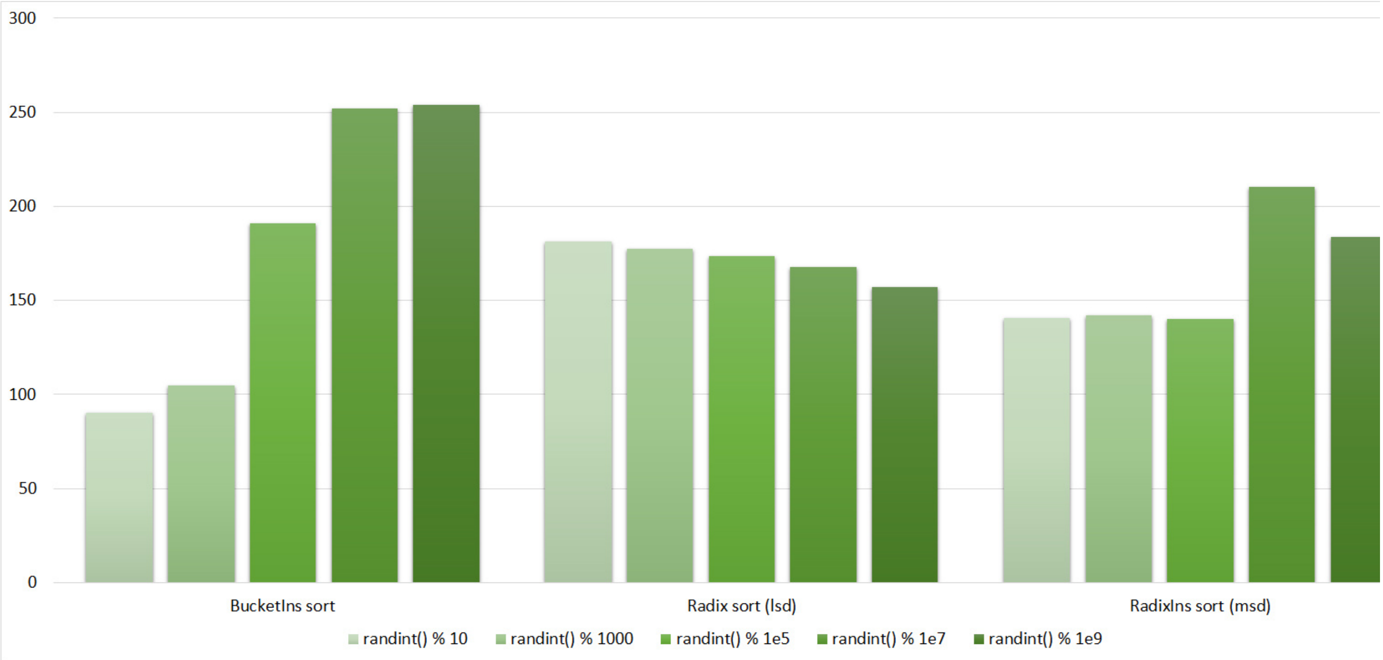
	1e6	Итог		1e7	Итог
1	Shellsort (Hibbard)	119,71	1	Shellsort (Hibbard)	2132,43
2	Bitonic sort	232,44	2	Bitonic sort	4415,81
3	Shellsort (Pratt)	343,50	3	Tree sort	5179,57
4	Tree sort	407,36	4	Shellsort (Pratt)	7741,29

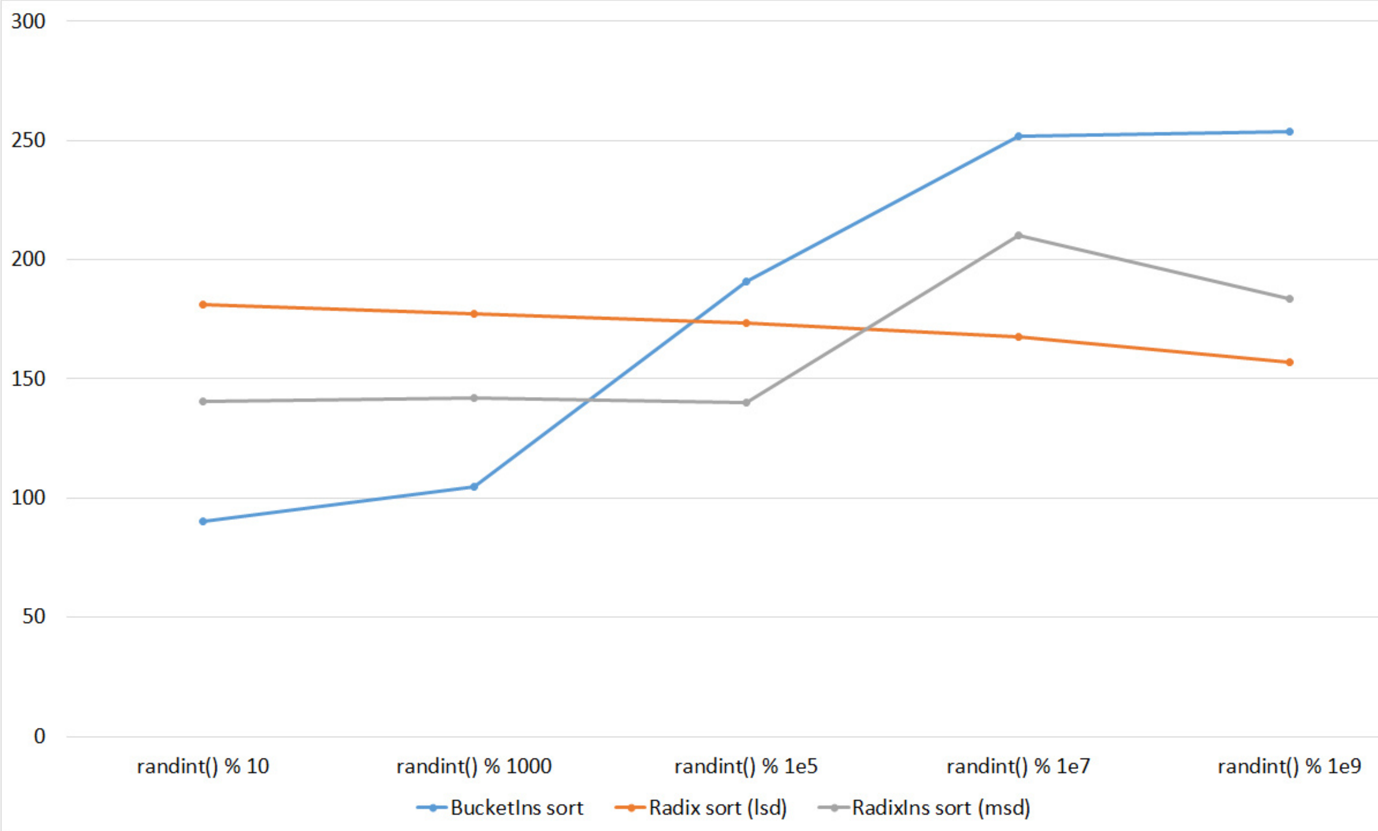
Убедительное первое место заняла сортировка Шелла по Хиббарду, не уступив ни в одной промежуточной группе. Возможно, стоило ее отправить в первую группу сортировок, но… она слишком слаба для этого, да и тогда почти никого не было бы в группе. Битонная сортиро довольно уверенно заняла второе место. Третье место при миллионе элементах заняла другая сортировка Шелла, а при десяти миллиона сортировка деревом (асимптотика сказалась). Стоит обратить внимание, что при десятикратном увеличении размера входных данных все алгоритмы, кроме древесной сортировки, замедлились почти в 20 раз, а последняя всего лишь в 13.

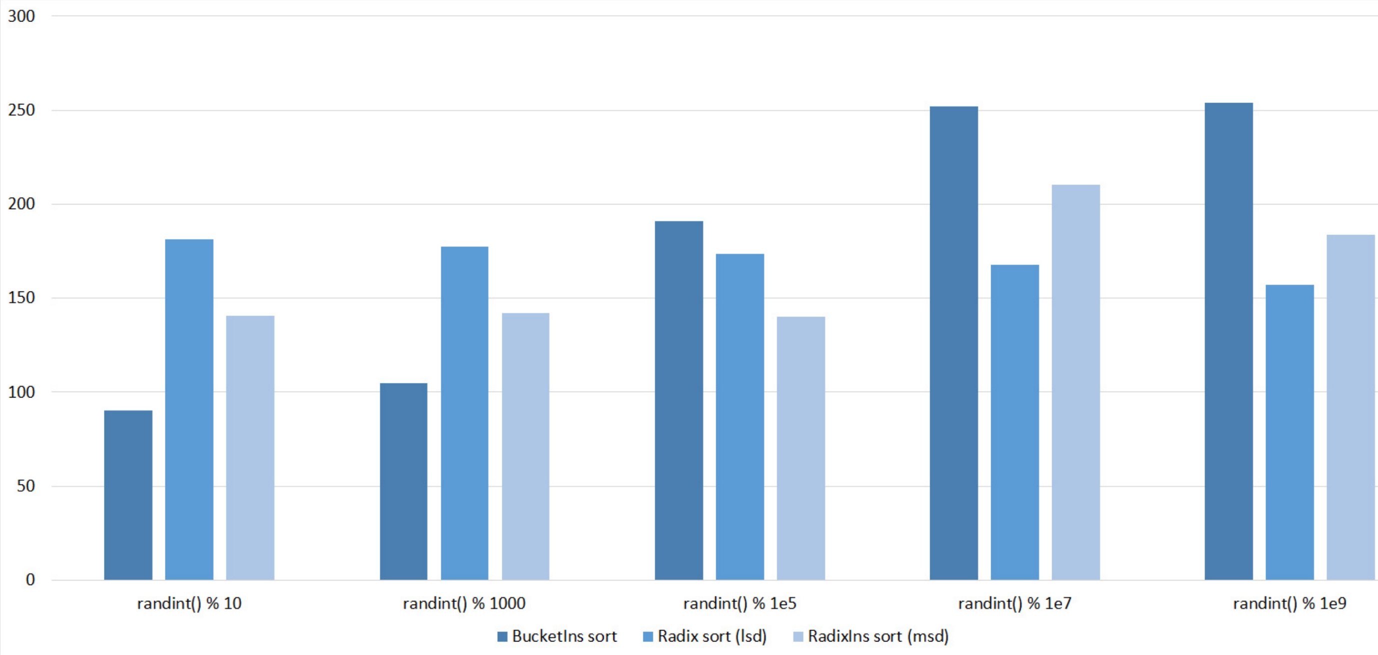
Третья группа сортировок

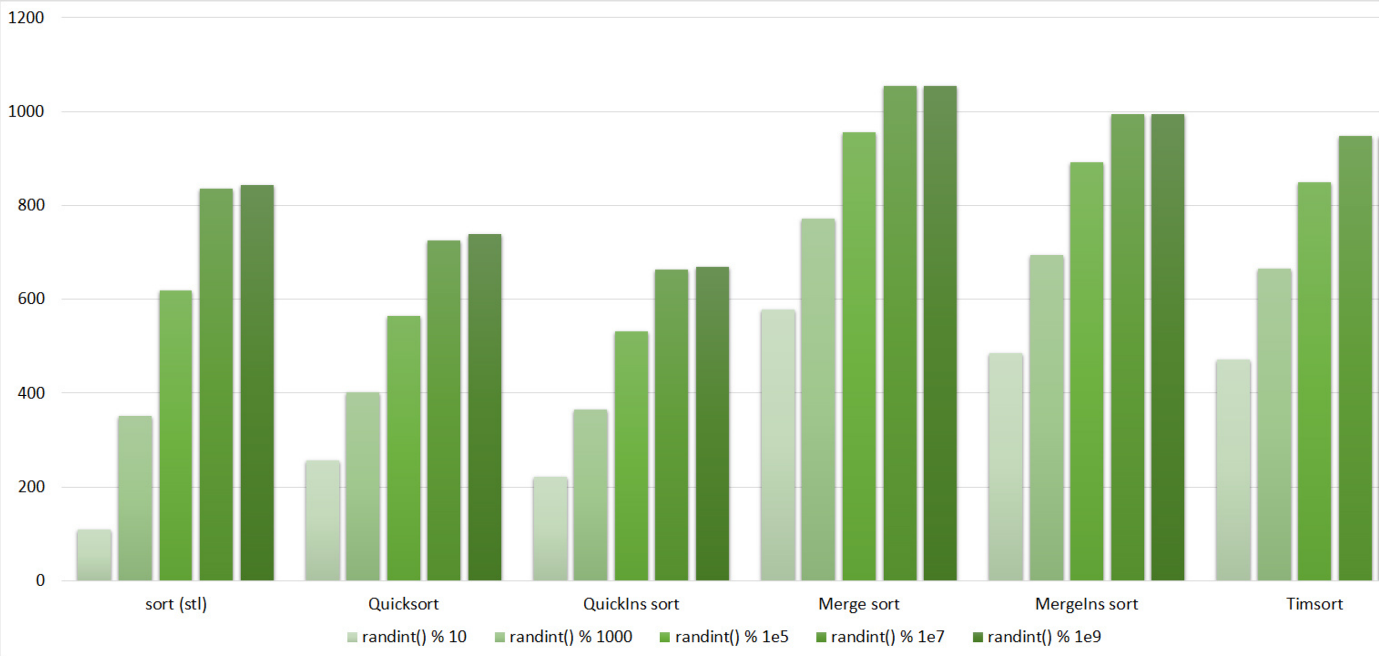
Массив случайных чисел

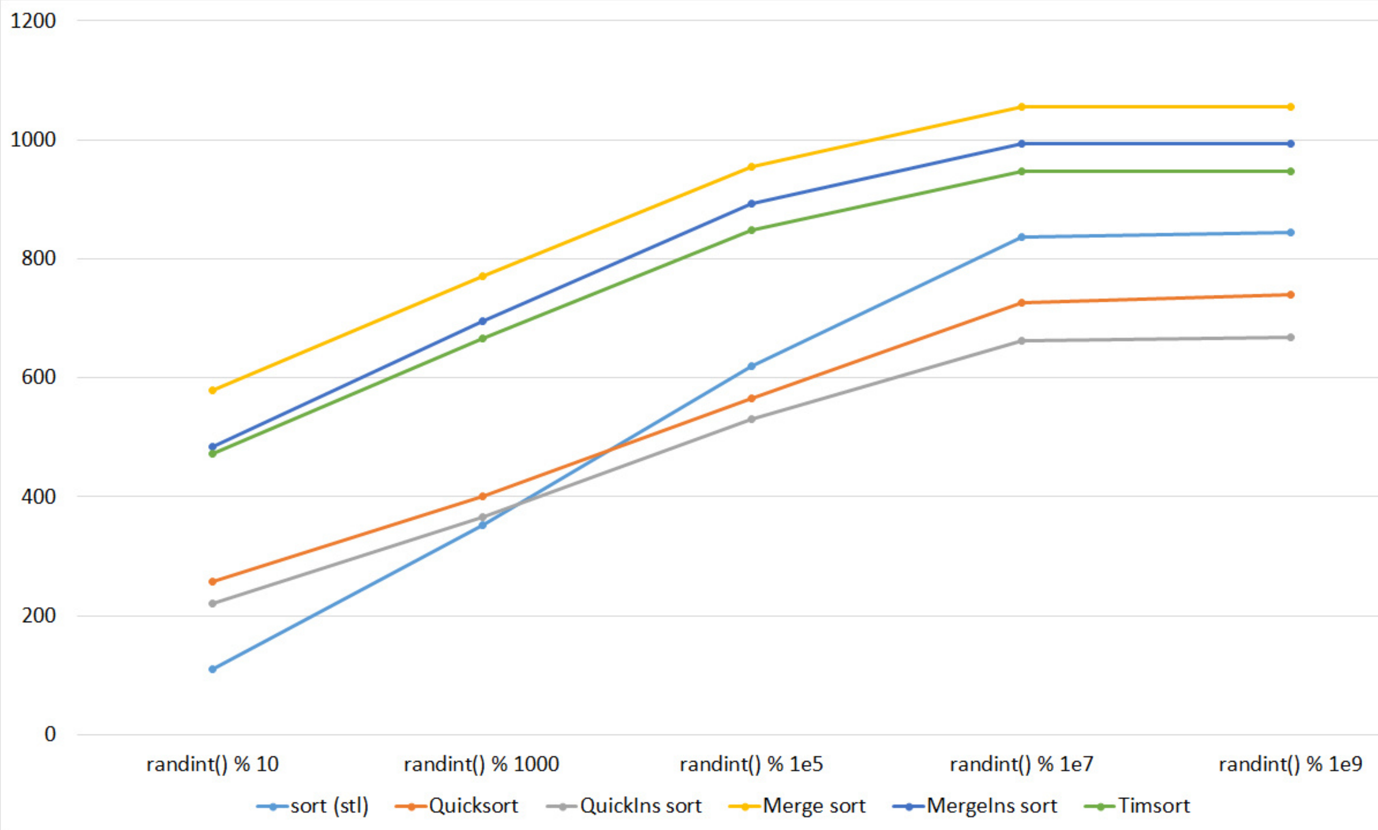
Таблицы, 1e7 элементов

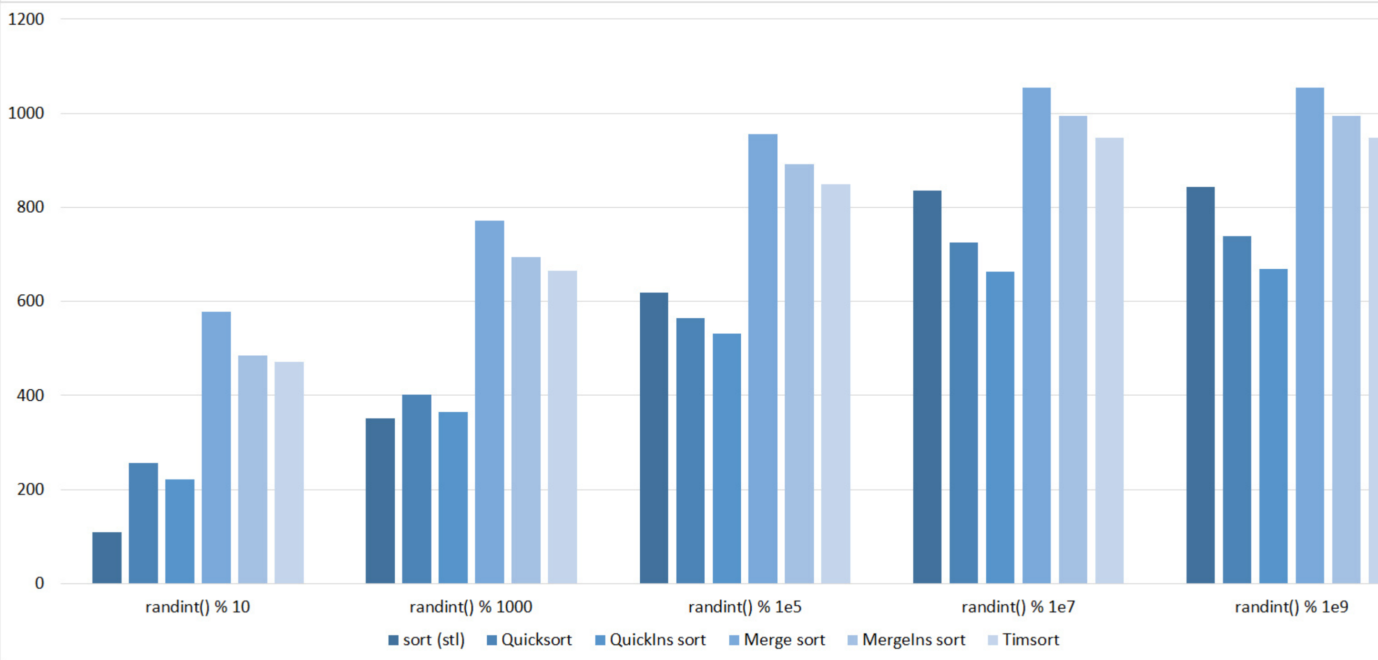

300
250
200
150
100
50
0
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9

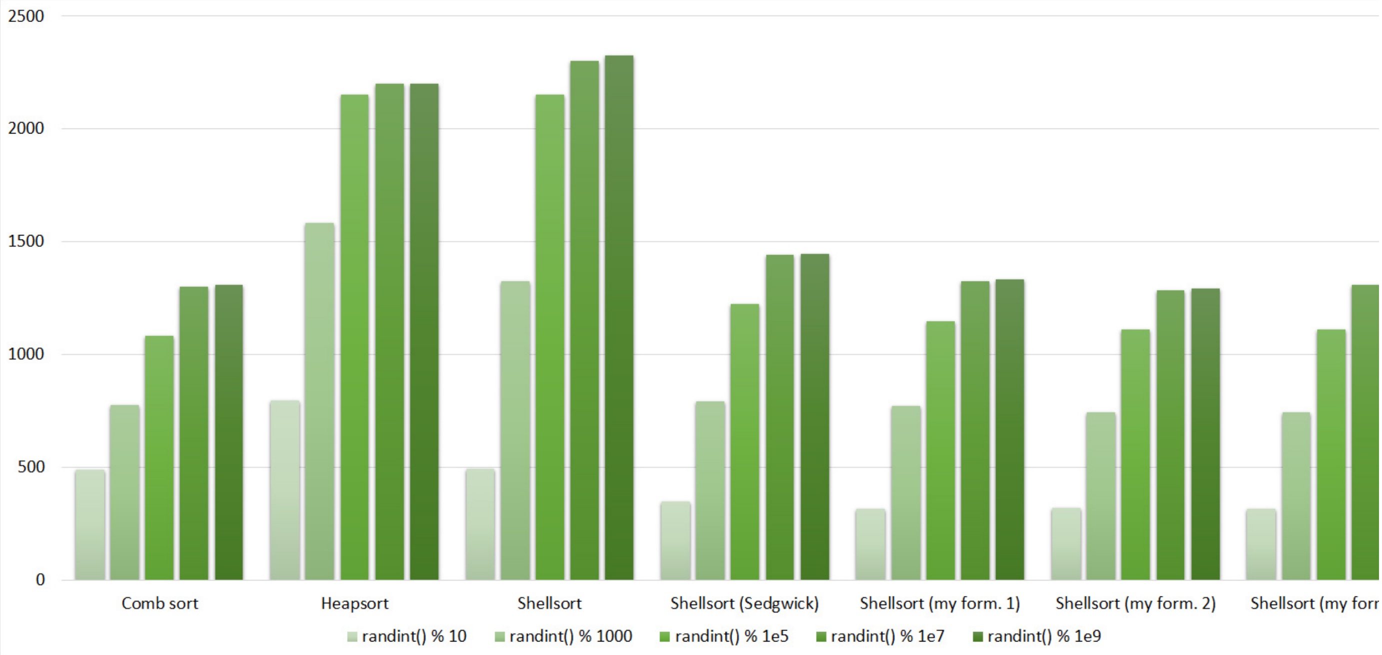

300
250
200
150
100
50
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

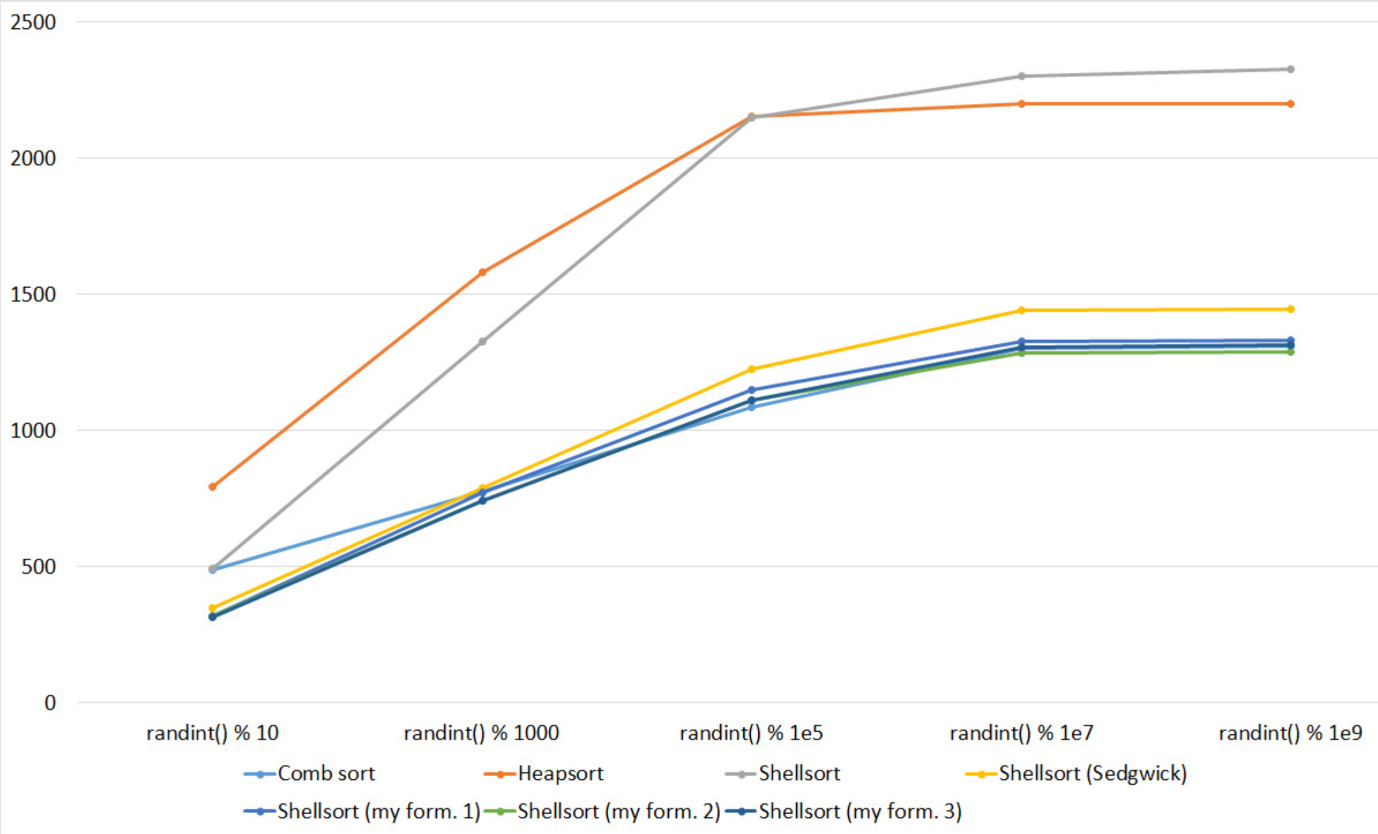

300
250
200
150
100
50
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

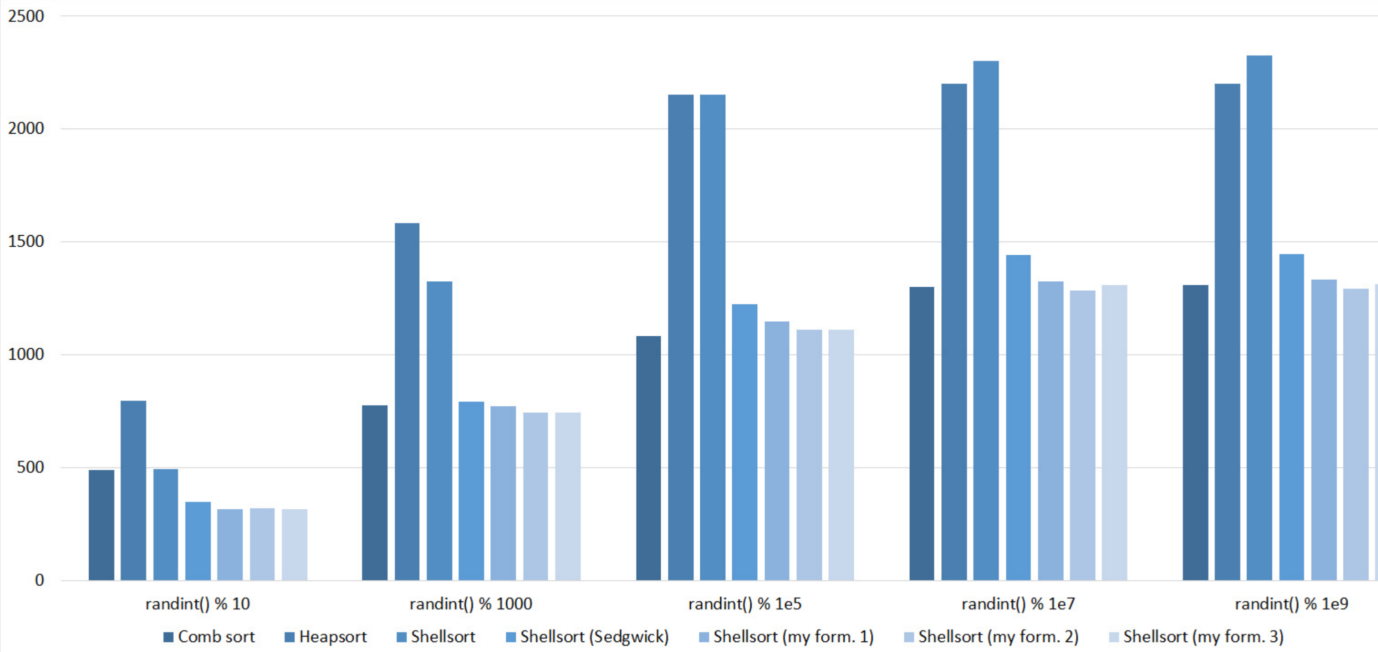

1200
1000
800
600
400
200
0
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9


1200
1000
800
600
400
200
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


1200
1000
800
600
400
200
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


2500
2000
1500
1000
500
0
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9

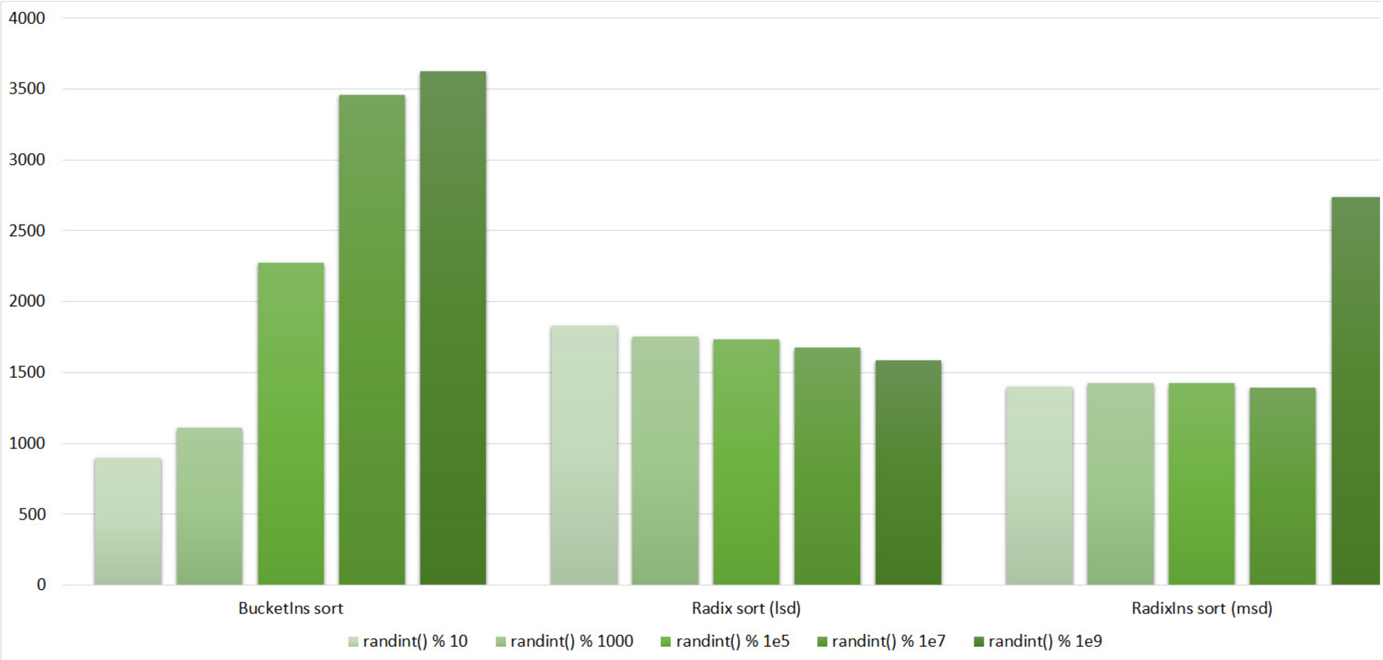

2500
2000
1500
1000
500
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my form. 3)

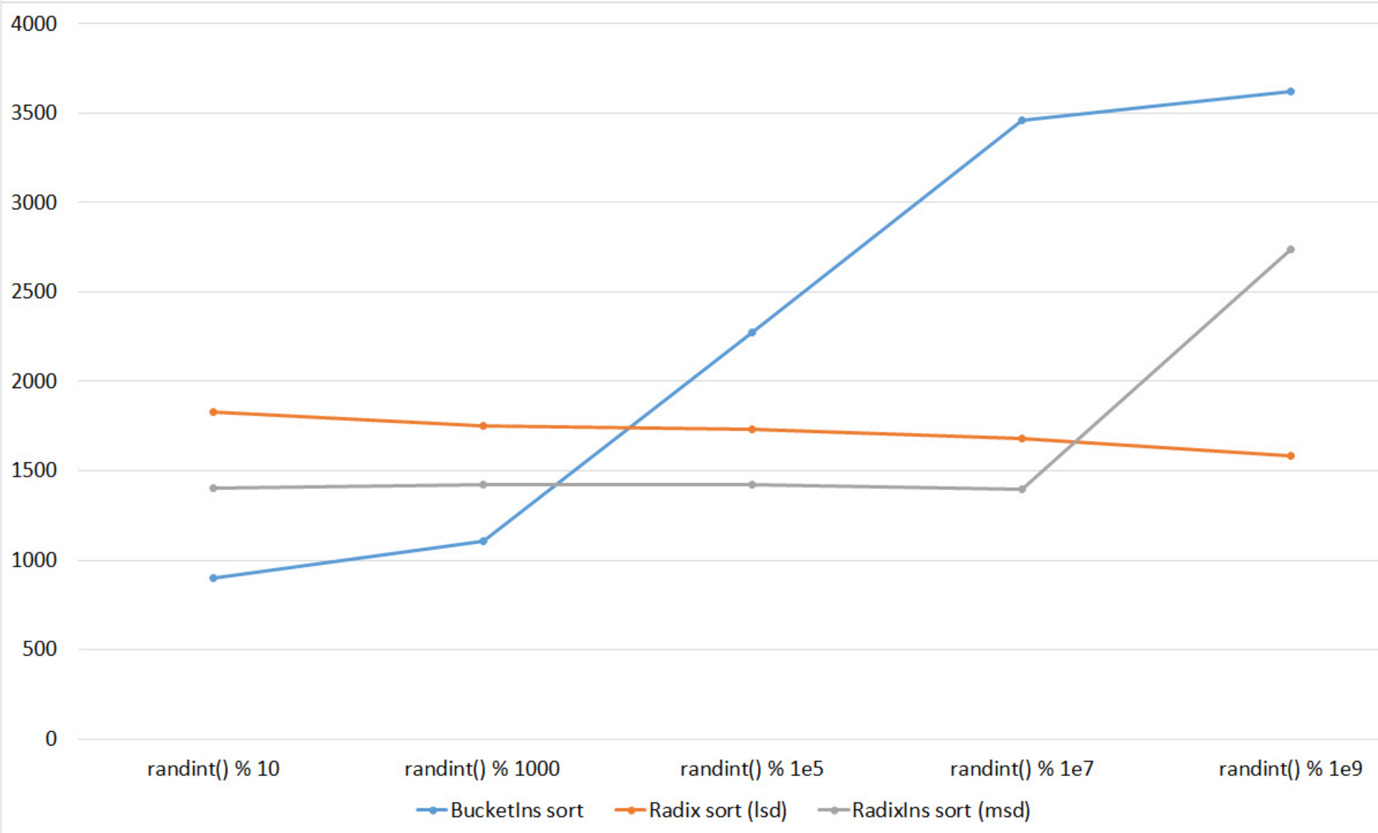


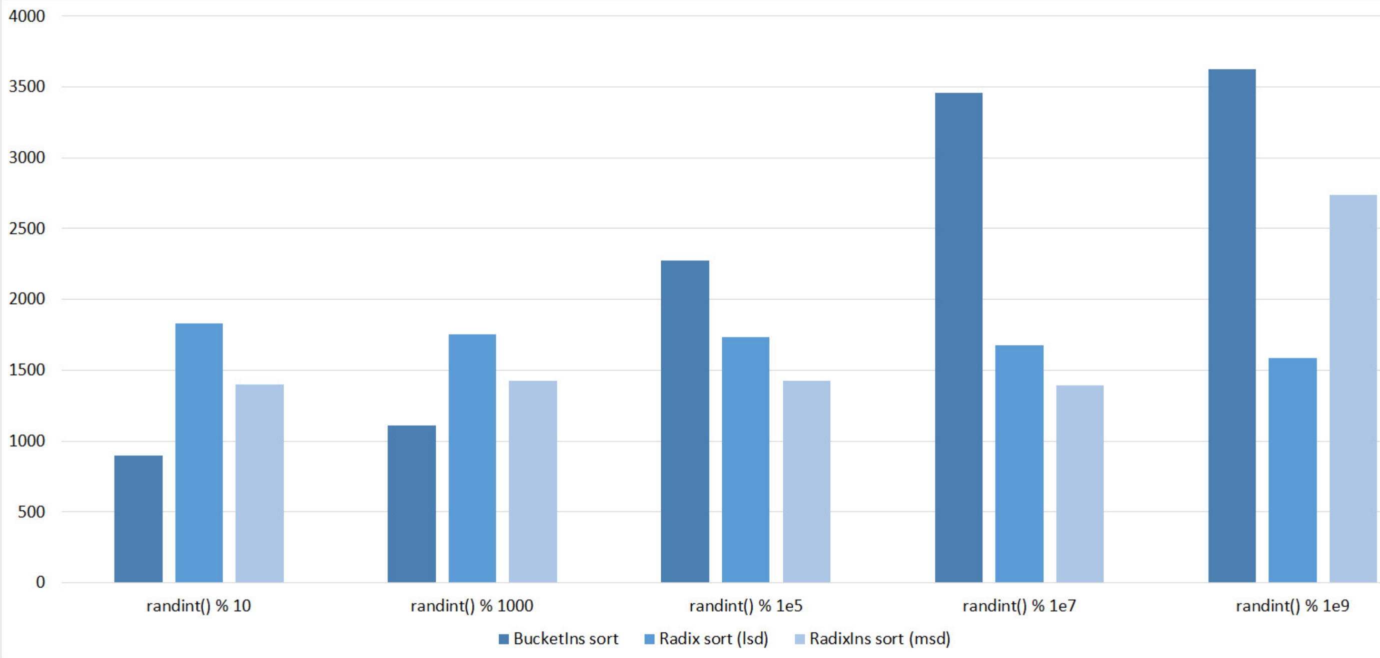
1e7	randint() % 10	randint() % 1000	randint() % 1e5	randint() % 1e7	randint() % 1e9	Итс
Comb sort	490,07	776,52	1084,92	1302,79	1309,92	992,
BucketIns sort	90,41	104,69	190,83	252,02	253,77	178,
Heapsort	795,00	1583,24	2153,63	2200,34	2201,13	1786
sort (stl)	109,95	351,97	619,36	835,72	844,22	552,
Quicksort	256,49	401,52	565,14	725,82	739,27	537,
QuickIns sort	221,17	365,27	530,97	662,79	668,09	489,
Merge sort	578,40	771,40	955,14	1054,61	1055,33	882,
MergeIns sort	484,19	695,08	891,85	994,18	994,18	811,
Shellsort	494,25	1327,03	2151,44	2303,41	2326,52	1720
Shellsort (Sedgwick)	348,94	790,38	1225,33	1442,77	1444,96	1050
Radix sort (lsd)	181,20	177,26	173,33	167,73	157,04	171,
RadixIns sort (msd)	140,44	142,01	139,89	210,19	183,72	163,
Timsort	472,02	665,68	848,17	947,70	947,92	776,
Shellsort (my form. 1)	317,33	773,39	1148,66	1326,39	1331,73	979,
Shellsort (my form. 2)	320,36	743,26	1111,25	1284,15	1290,80	949,
Shellsort (my form. 3)	316,82	742,40	1112,23	1308,68	1314,89	959,

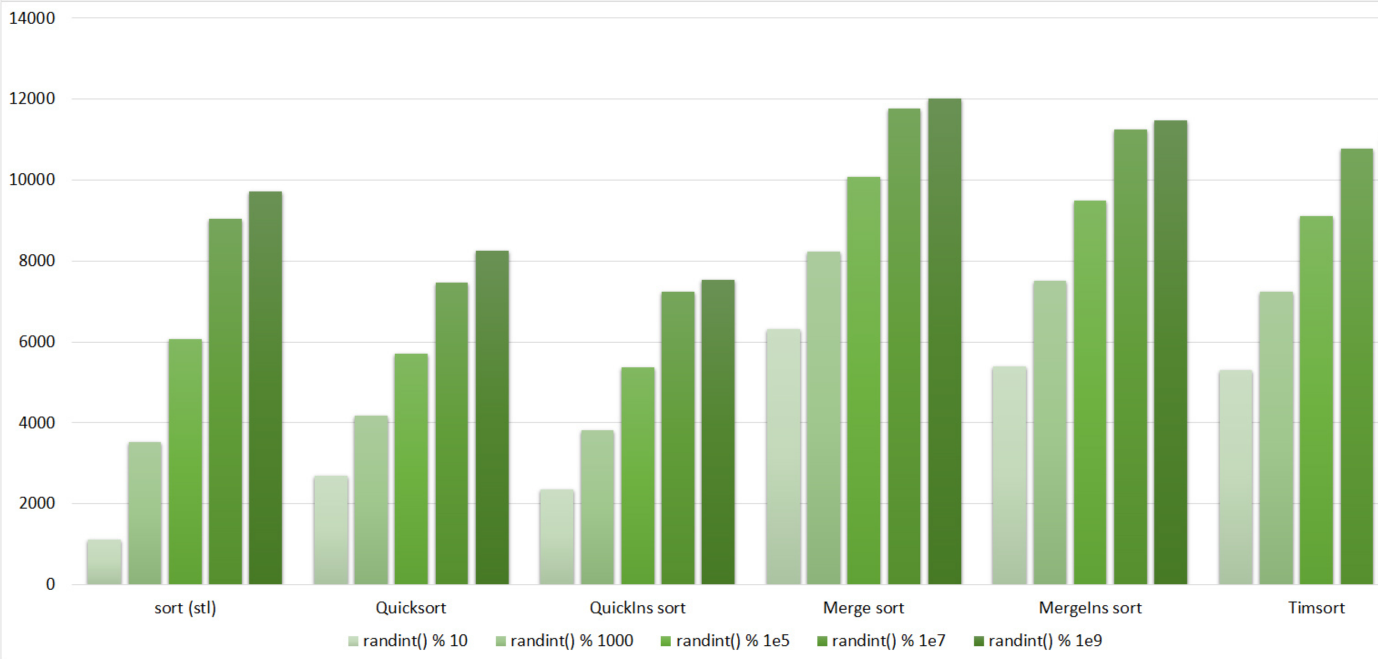
1e7	Итог
RadixIns sort (msd)	163,25
Radix sort (lsd)	171,31
BucketIns sort	178,34
QuickIns sort	489,66
Quicksort	537,65
sort (stl)	552,24
Timsort	776,30
MergeIns sort	811,90
Merge sort	882,98
Shellsort (my form. 2)	949,96
Shellsort (my form. 3)	959,00
Shellsort (my form. 1)	979,50
Comb sort	992,84
Shellsort (Sedgwick)	1050,48
Shellsort	1720,53
Heapsort	1786,67

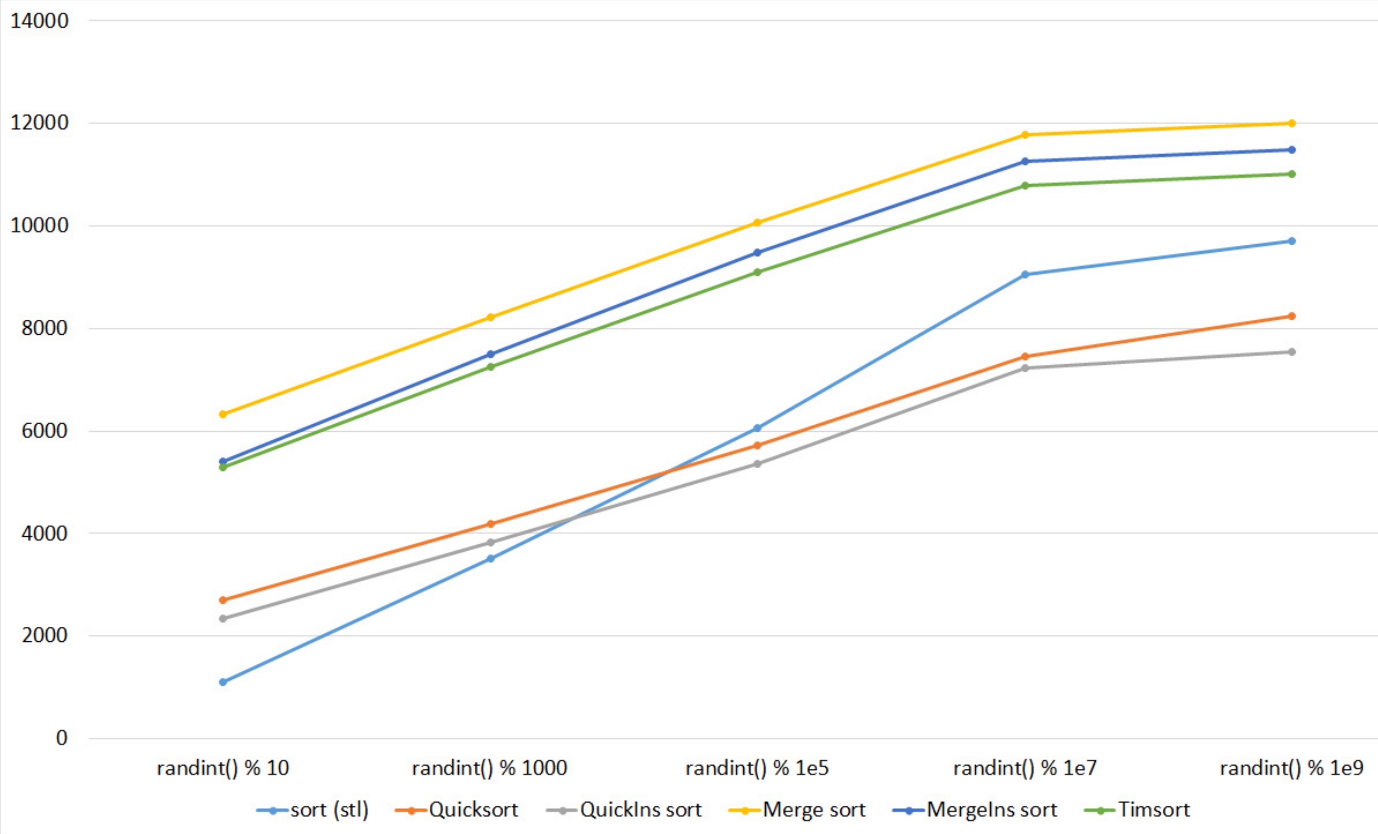
Таблицы, 1e8 элементов

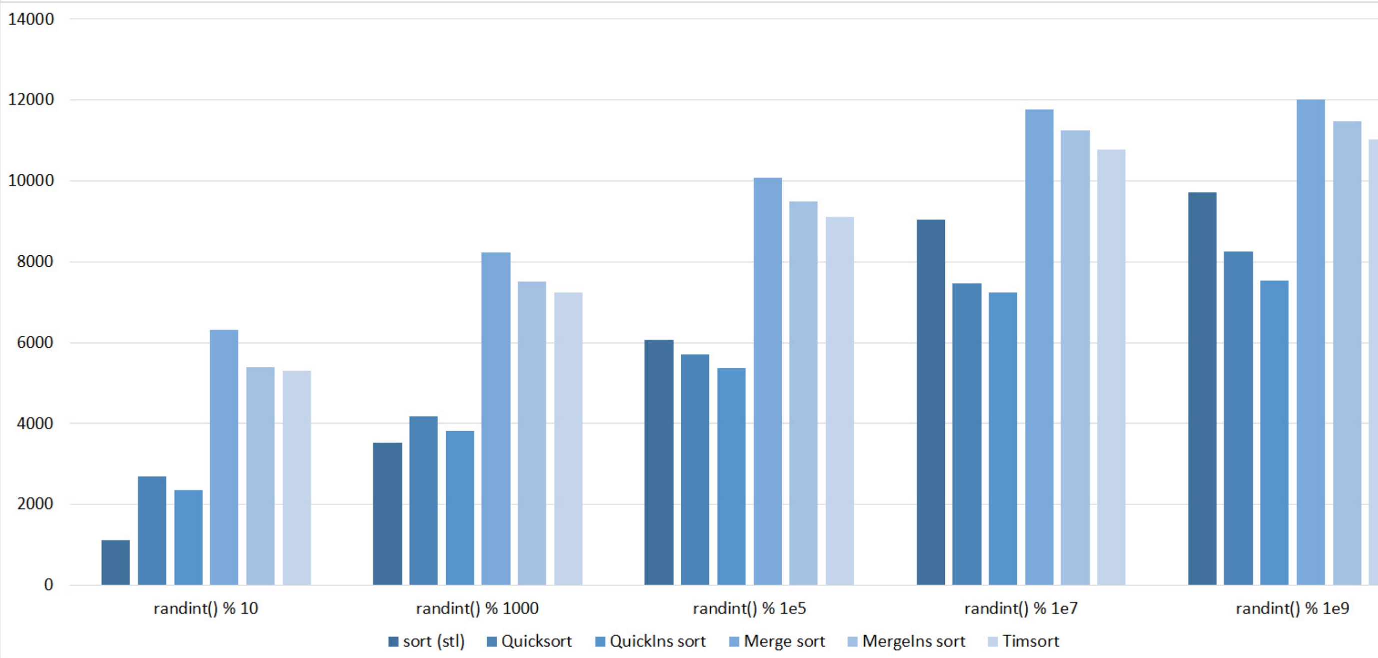

4000
3500
3000
2500
2000
1500
1000
500
0
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9

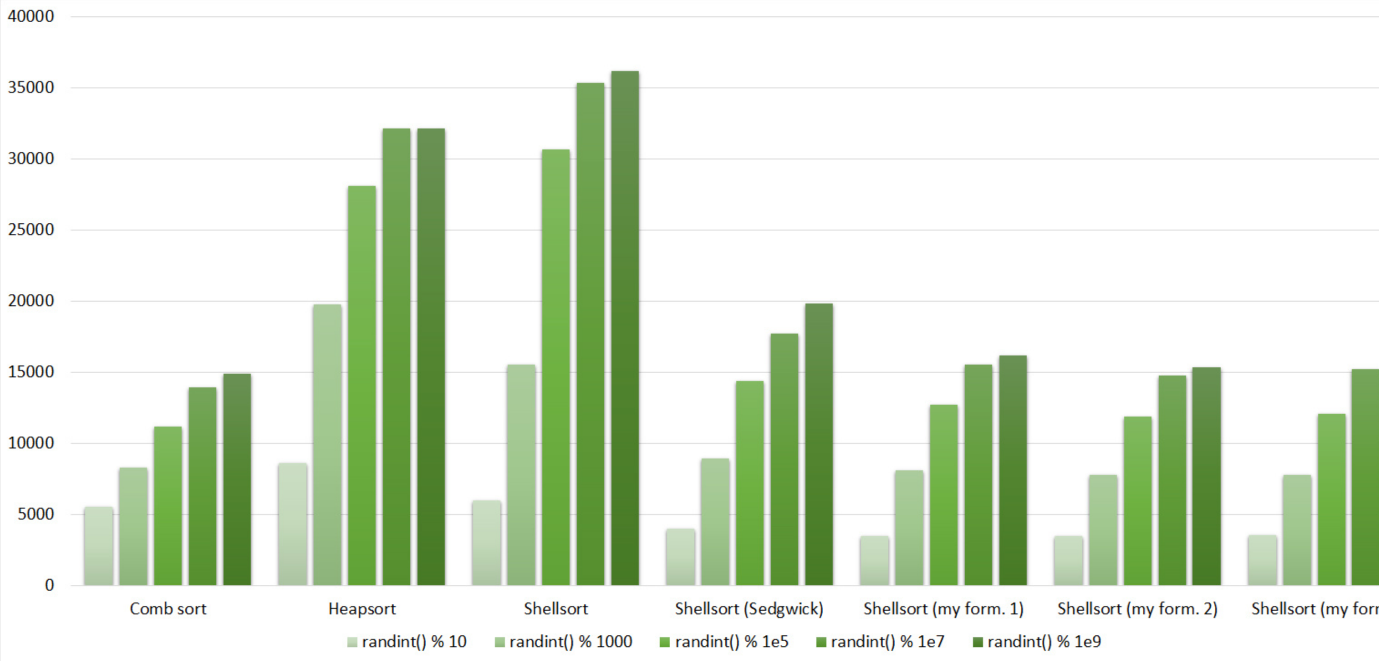

4000
3500
3000
2500
2000
1500
1000
500
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

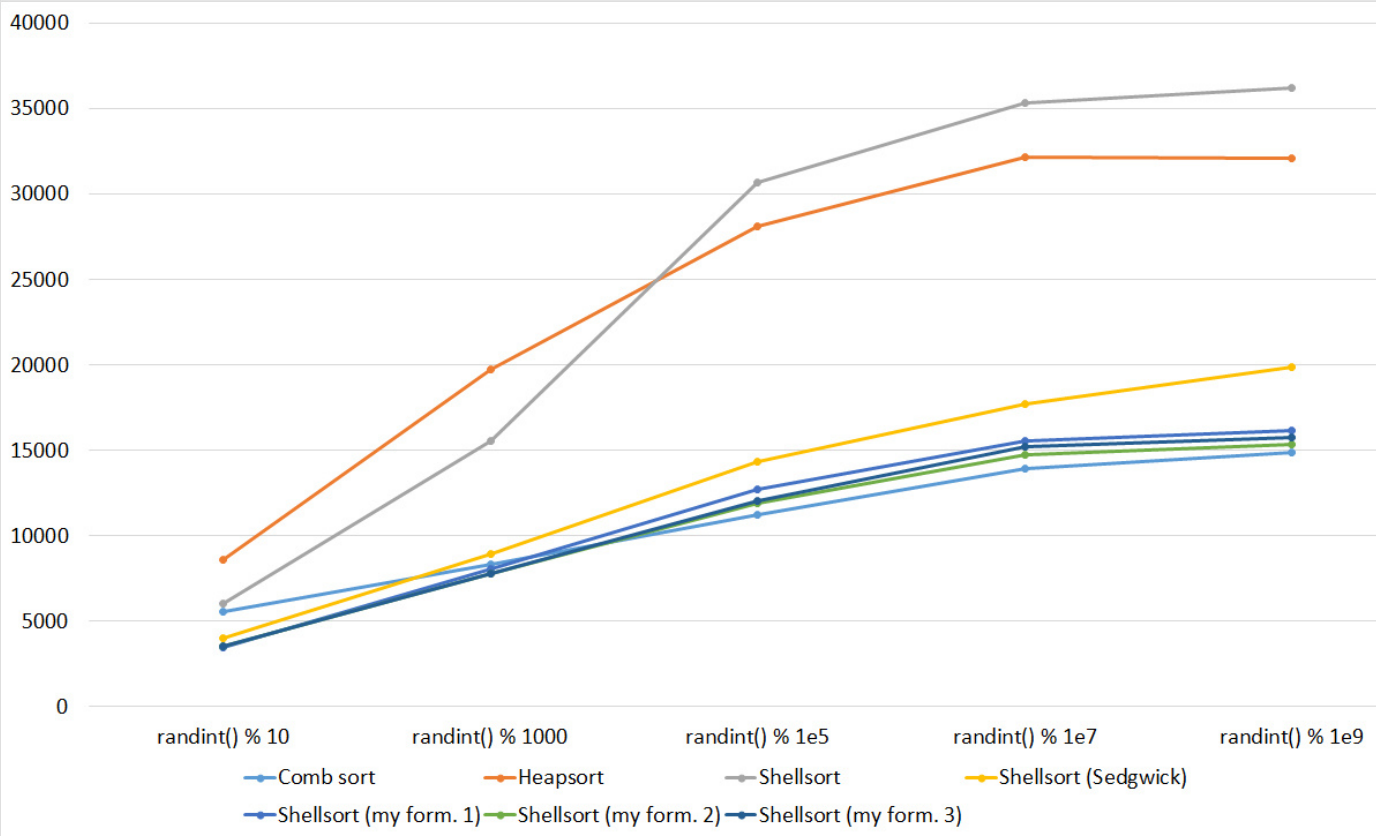

4000
3500
3000
2500
2000
1500
1000
500
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

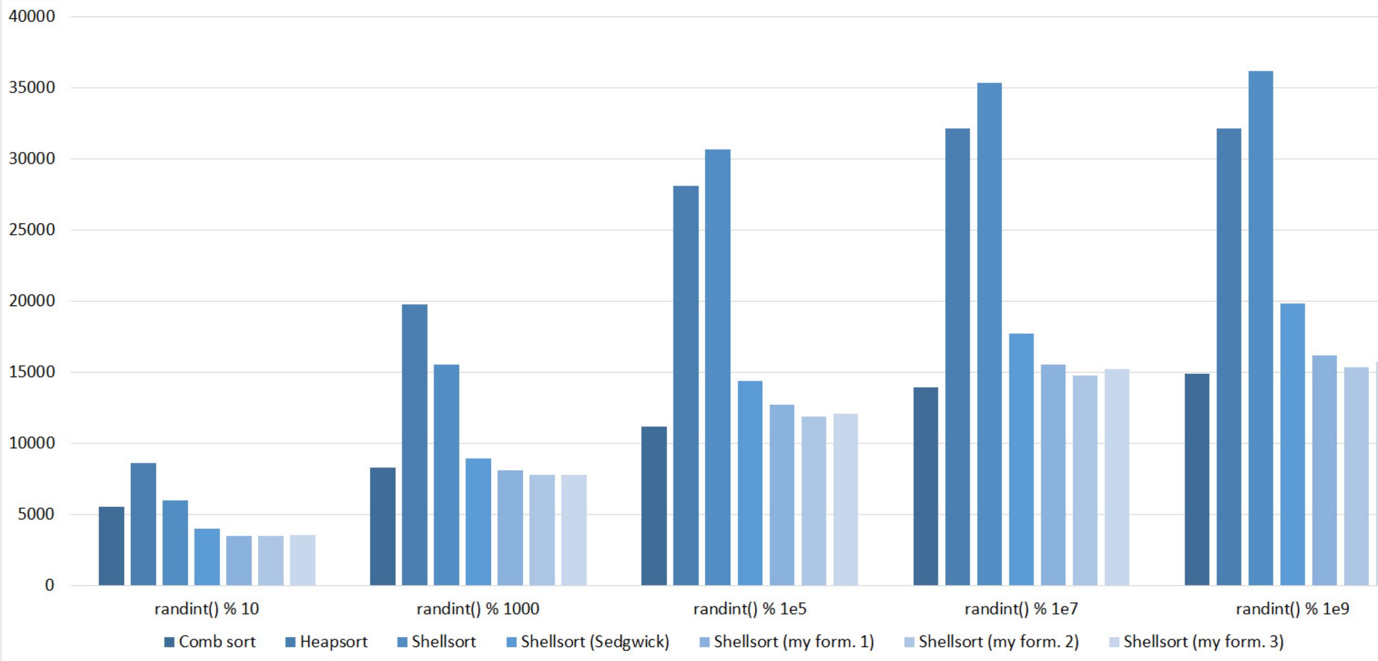

14000
12000
10000
8000
6000
4000
2000
0
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9


14000
12000
10000
8000
6000
4000
2000
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


14000
12000
10000
8000
6000
4000
2000
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


40000
35000
30000
25000
20000
15000
10000
5000
0
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9


40000
35000
30000
25000
20000
15000
10000
5000
0
randint() % 10
randint() % 1000
randint() % 1e5
randint() % 1e7
randint() % 1e9
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my form. 3)



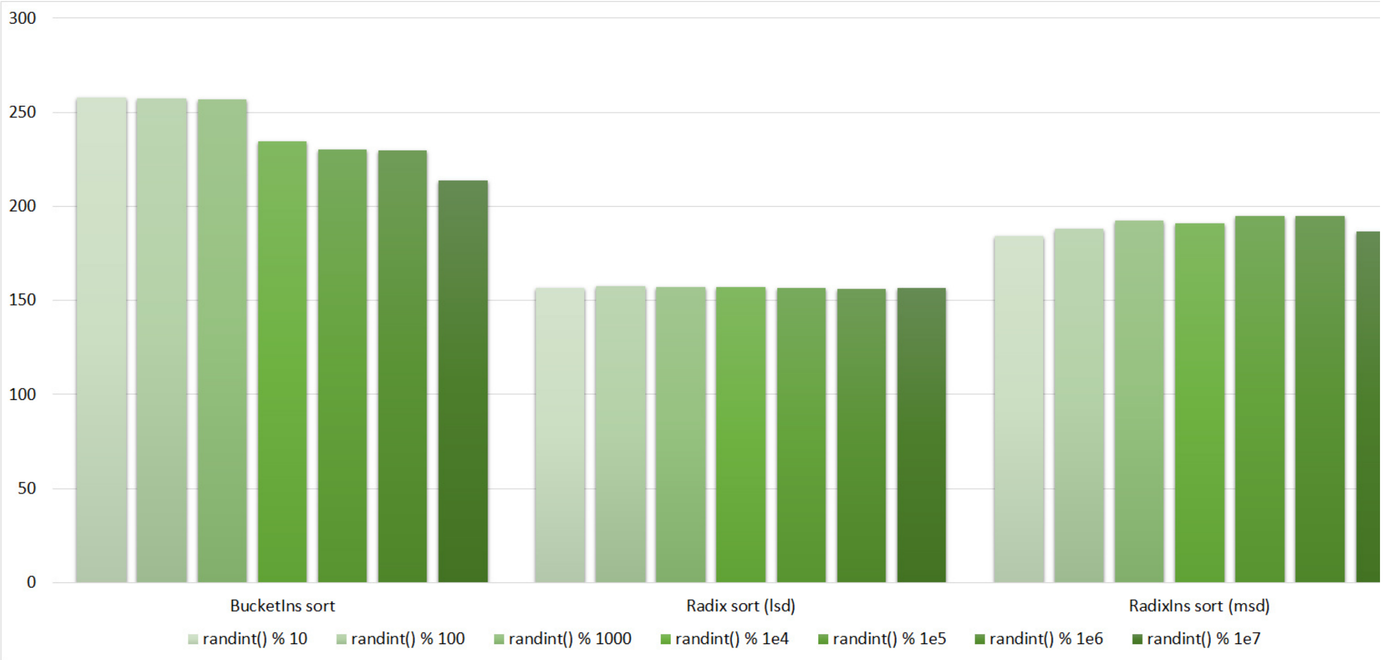
1e8	randint() % 10	randint() % 1000	randint() % 1e5	randint() % 1e7	randint() % 1e9	Ито
Comb sort	5539,46	8327,01	11214,76	13938,18	14889,92	10781
BucketIns sort	897,63	1107,86	2274,65	3461,23	3624,31	2273
Heapsort	8605,38	19781,78	28142,43	32154,49	32135,25	24163
sort (stl)	1111,53	3514,00	6066,36	9048,10	9708,09	5889
Quicksort	2696,49	4181,16	5717,60	7455,10	8253,56	5660
QuickIns sort	2349,34	3823,95	5361,33	7237,71	7542,84	5263
Merge sort	6319,84	8225,12	10078,11	11770,72	12006,49	9680
MergeIns sort	5397,77	7506,57	9485,87	11249,87	11480,90	9024
Shellsort	6021,47	15575,28	30676,05	35382,80	36224,07	24775
Shellsort (Sedgwick)	3998,27	8969,25	14372,50	17726,73	19877,87	12988
Radix sort (lsd)	1829,52	1753,09	1732,30	1677,63	1584,96	1715
RadixIns sort (msd)	1401,58	1422,18	1422,92	1395,71	2741,09	1676
Timsort	5297,20	7245,54	9099,84	10782,77	11016,88	8688
Shellsort (my form. 1)	3471,46	8097,97	12706,27	15570,39	16161,13	11201
Shellsort (my form. 2)	3509,83	7781,40	11882,80	14771,89	15348,12	10658
Shellsort (my form. 3)	3539,88	7796,23	12065,66	15243,48	15739,60	10876

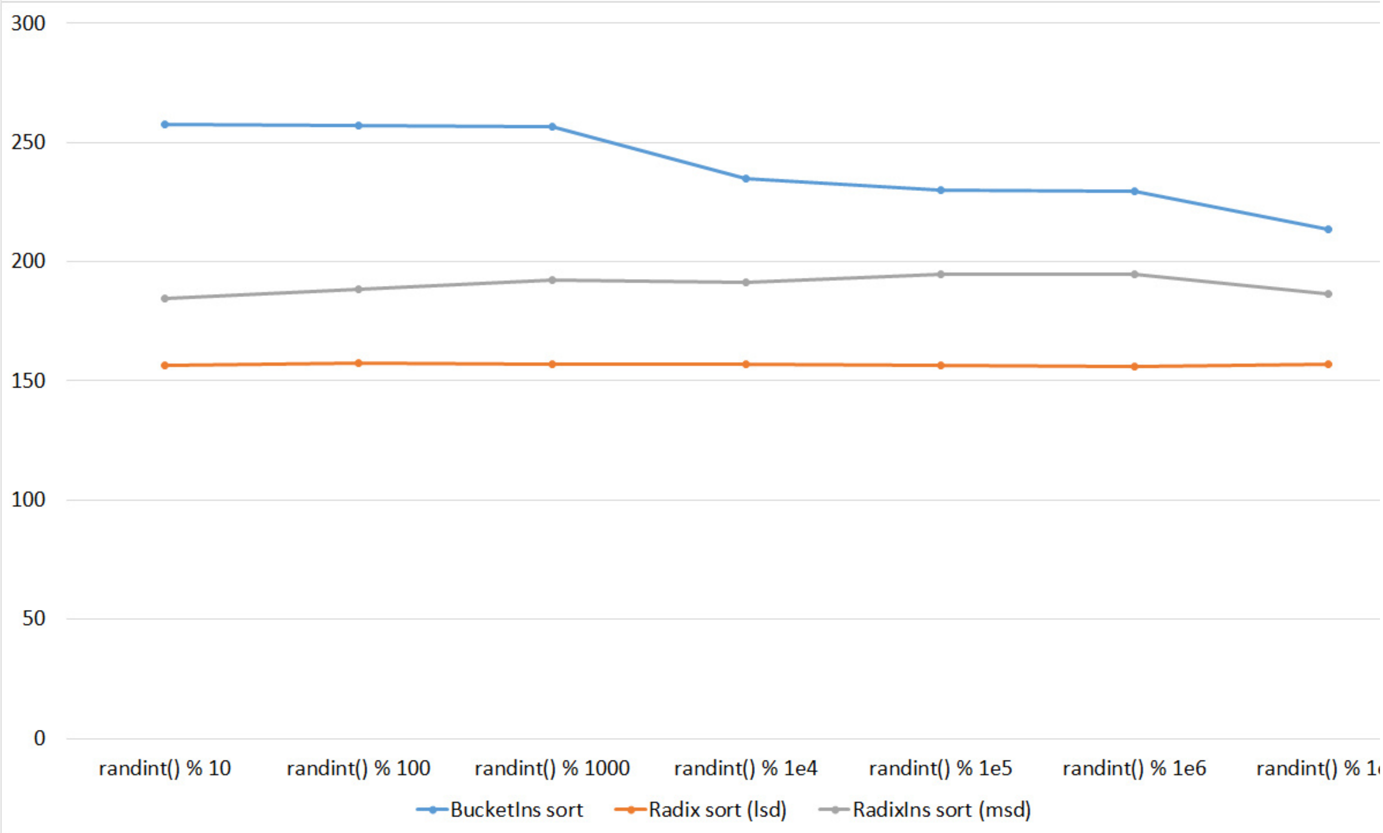
1e8	Итог
RadixIns sort (msd)	1676,70
Radix sort (lsd)	1715,50
BucketIns sort	2273,14
QuickIns sort	5263,03
Quicksort	5660,78
sort (stl)	5889,62
Timsort	8688,45
MergeIns sort	9024,20
Merge sort	9680,06
Shellsort (my form. 2)	10658,81
Comb sort	10781,87
Shellsort (my form. 3)	10876,97
Shellsort (my form. 1)	11201,44
Shellsort (Sedgwick)	12988,92
Heapsort	24163,87
Shellsort	24775,93

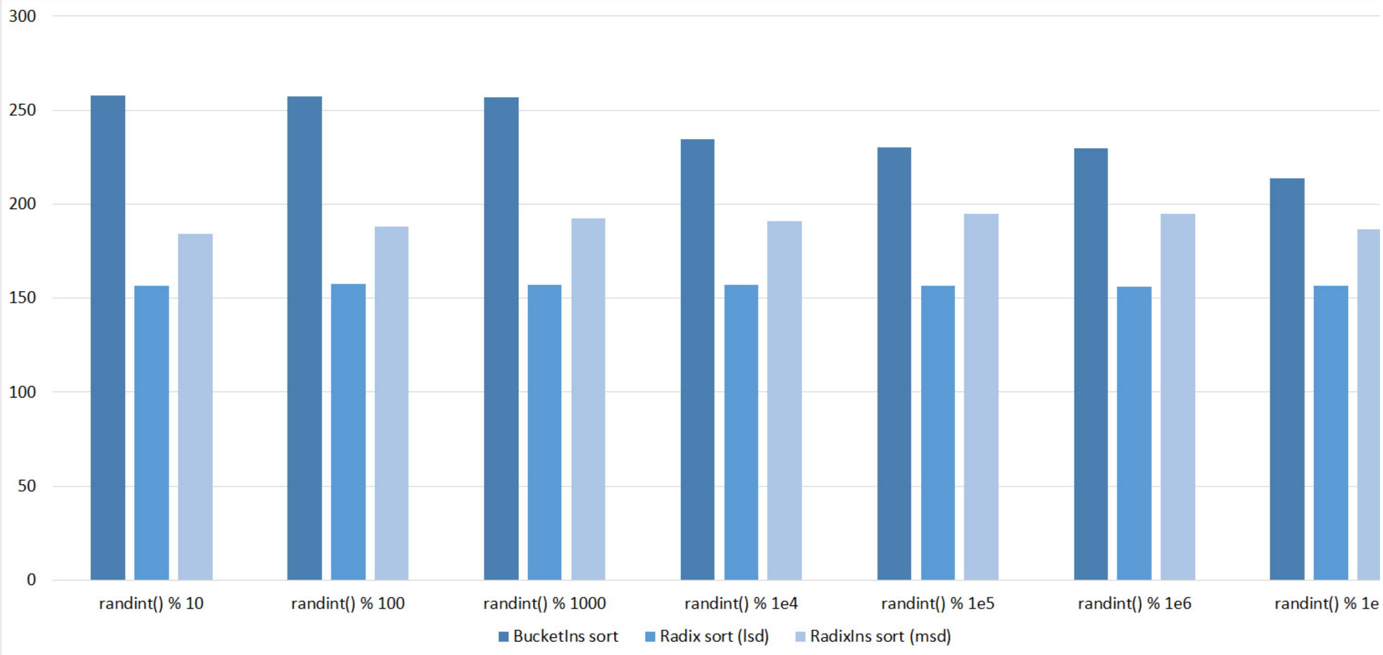
Почти все сортировки этой группы имеют почти одинаковую динамику. Почему же почти все сортировки ускоряются, когда массив частично отсортирован? Обменные сортировки работают быстрее потому, что надо делать меньше обменов, в сортировке Шелла выполняется сорті вставками, которая сильно ускоряется на таких массивах, в пирамидальной сортировке при вставке элементов сразу завершается просеив сортировке слиянием выполняется в лучшем случае вдвое меньше сравнений. Блочная сортировка работает тем лучше, чем меньше разн между минимальным и максимальным элементом. Принципиально отличается только поразрядная сортировка, которой все это безразличн LSD-версия работает тем лучше, чем больший модуль. Динамика MSD-версия мне не ясна, то, что она сработала быстрее чем LSD удивил

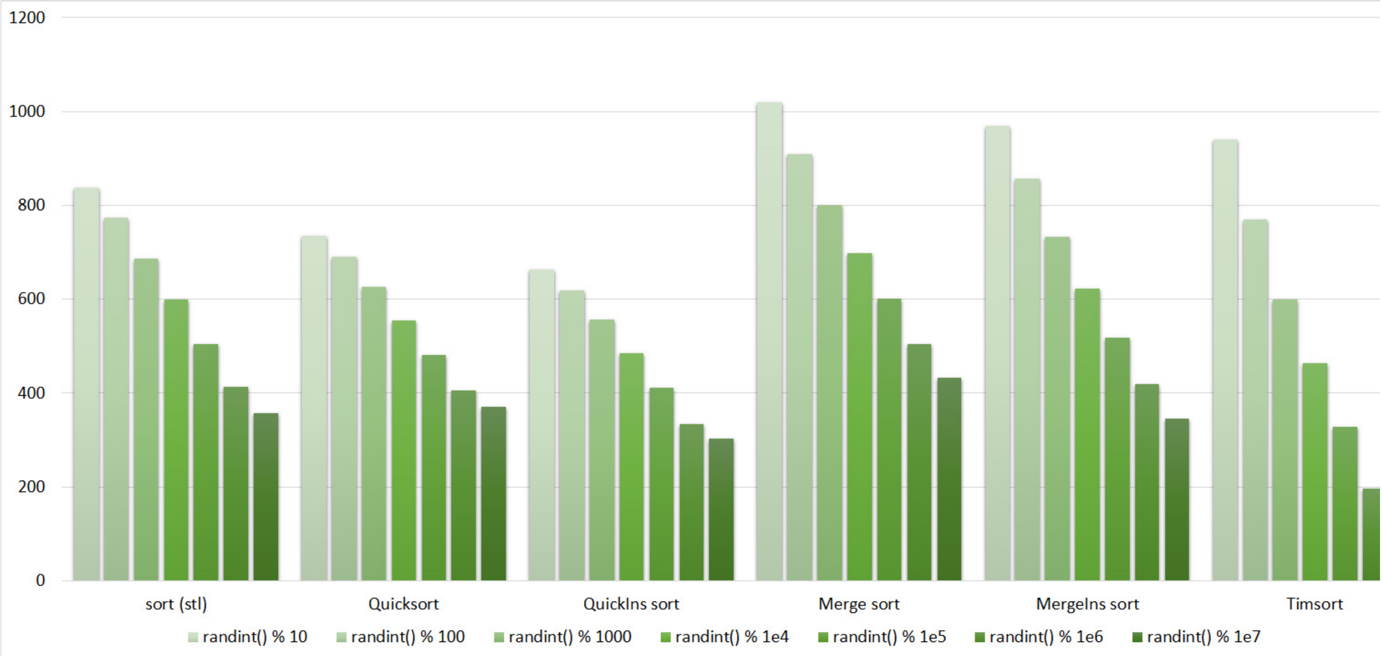
Частично отсортированный массив

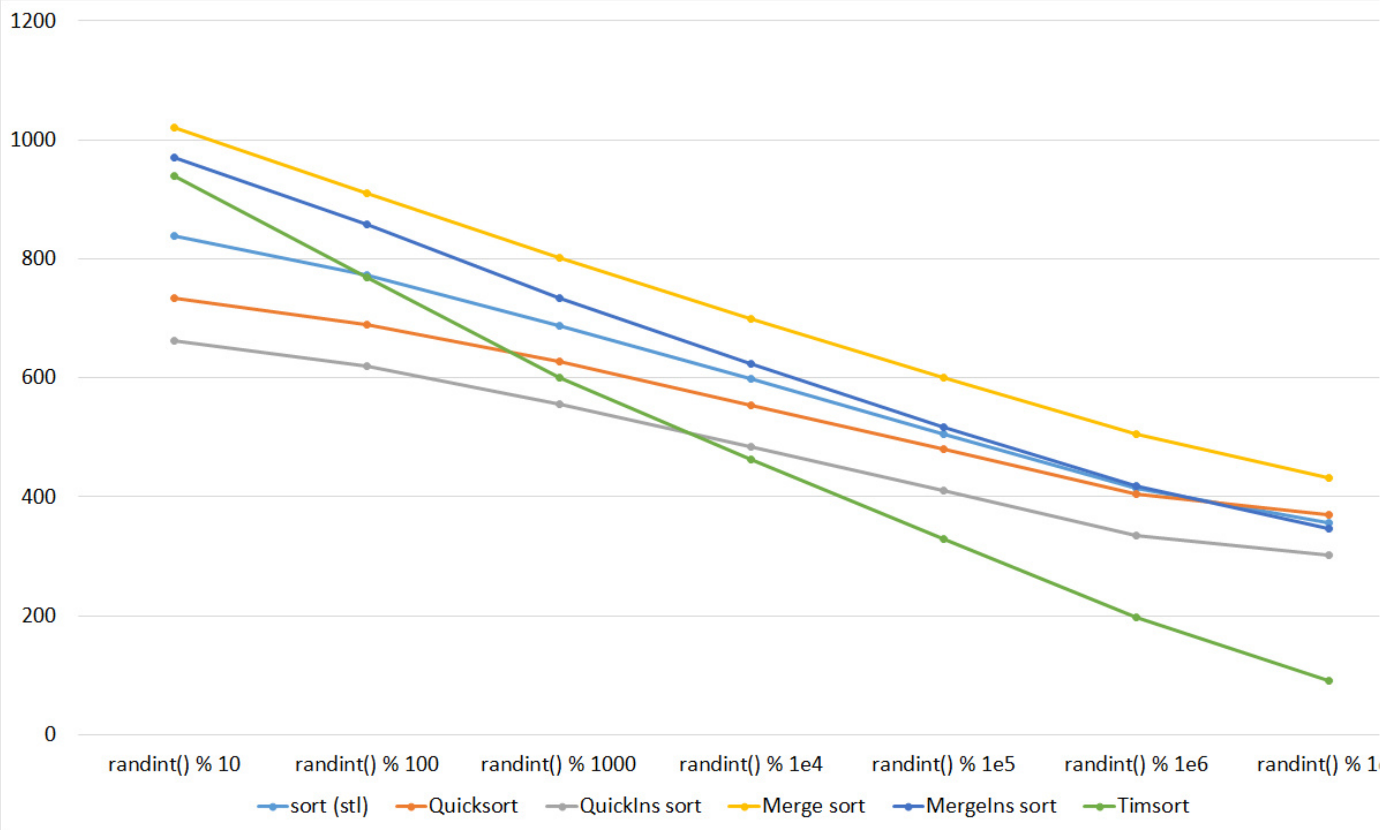
Таблицы, 1e7 элементов


300
250
200
150
100
50
0
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e7

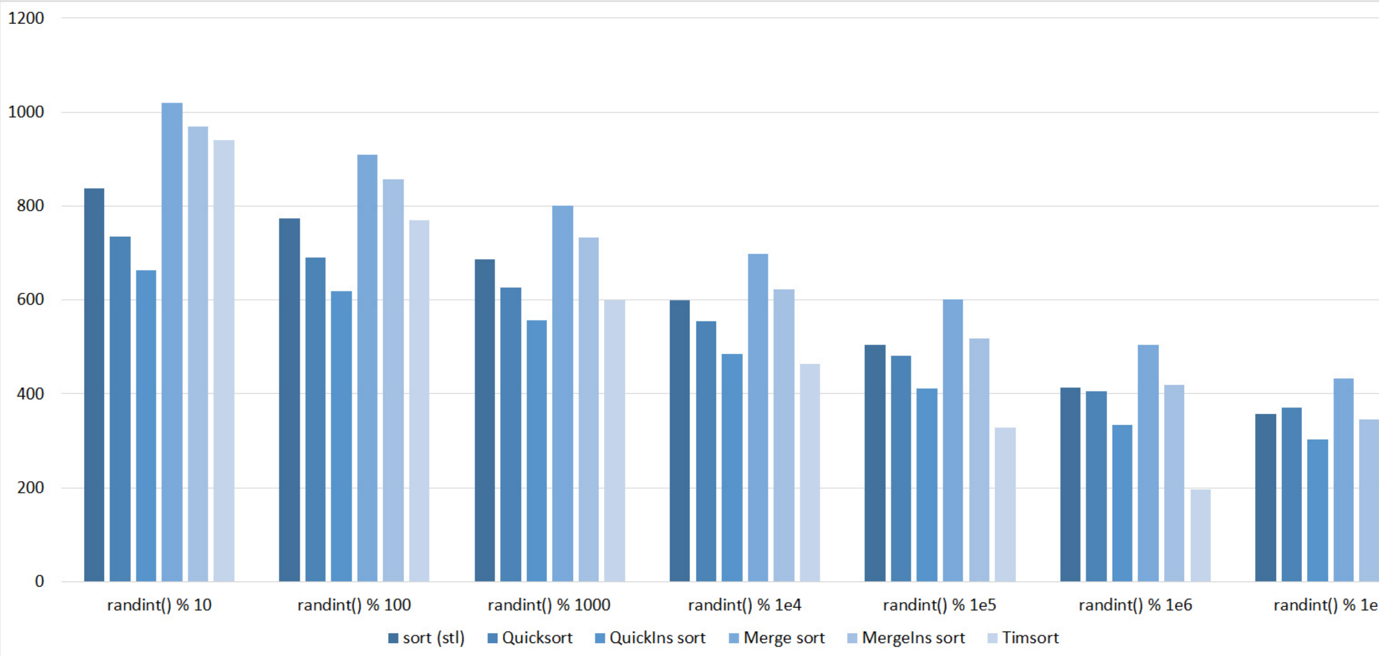

300
250
200
150
100
50
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)


300
250
200
150
100
50
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

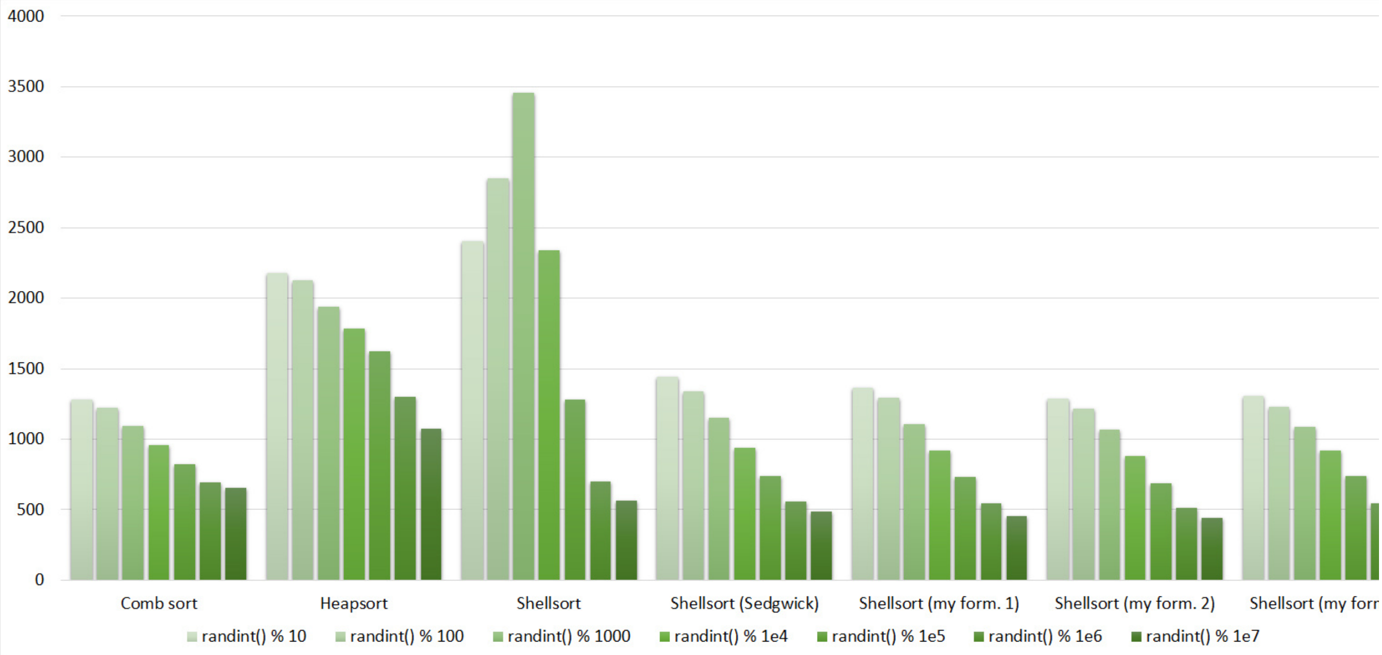

1200
1000
800
600
400
200
0
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e7

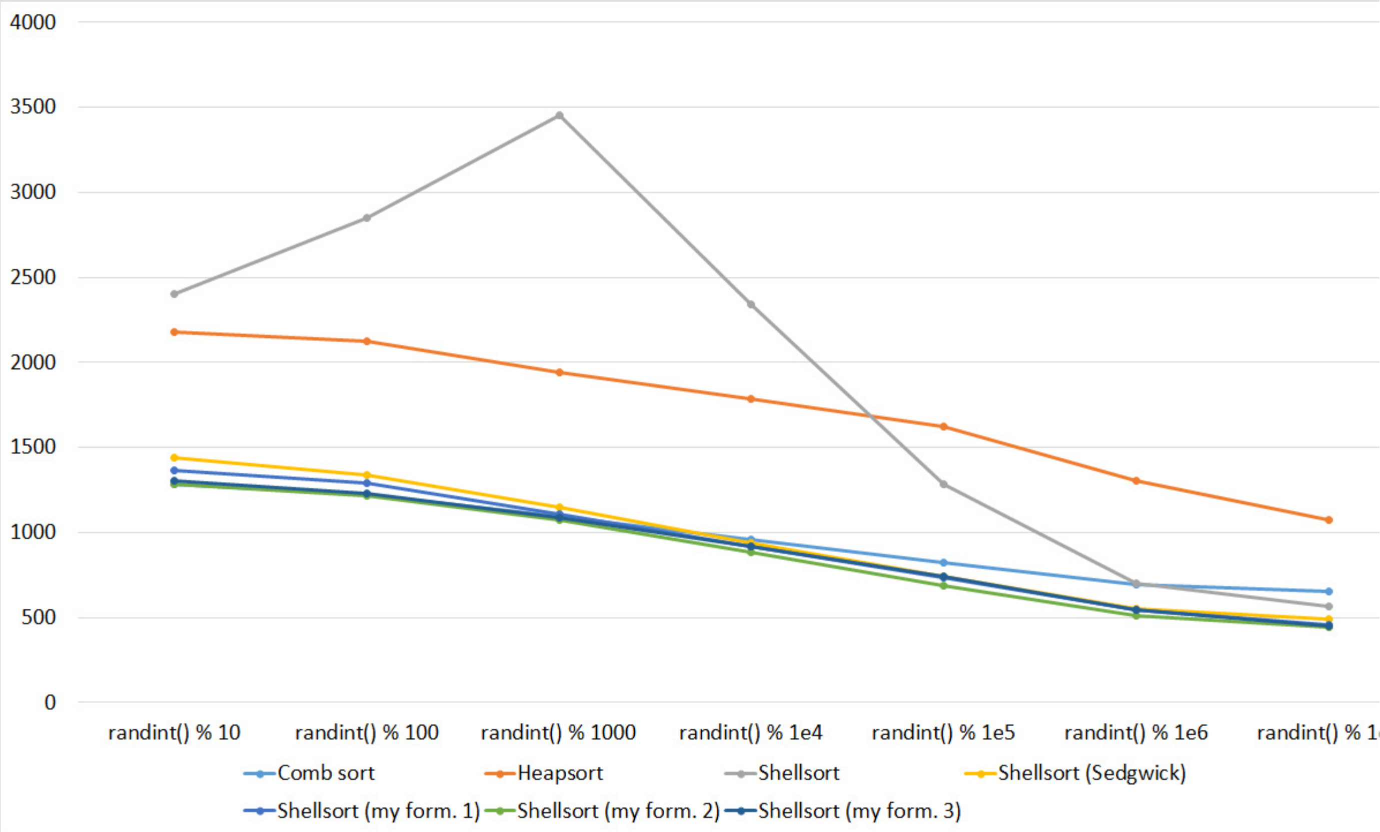


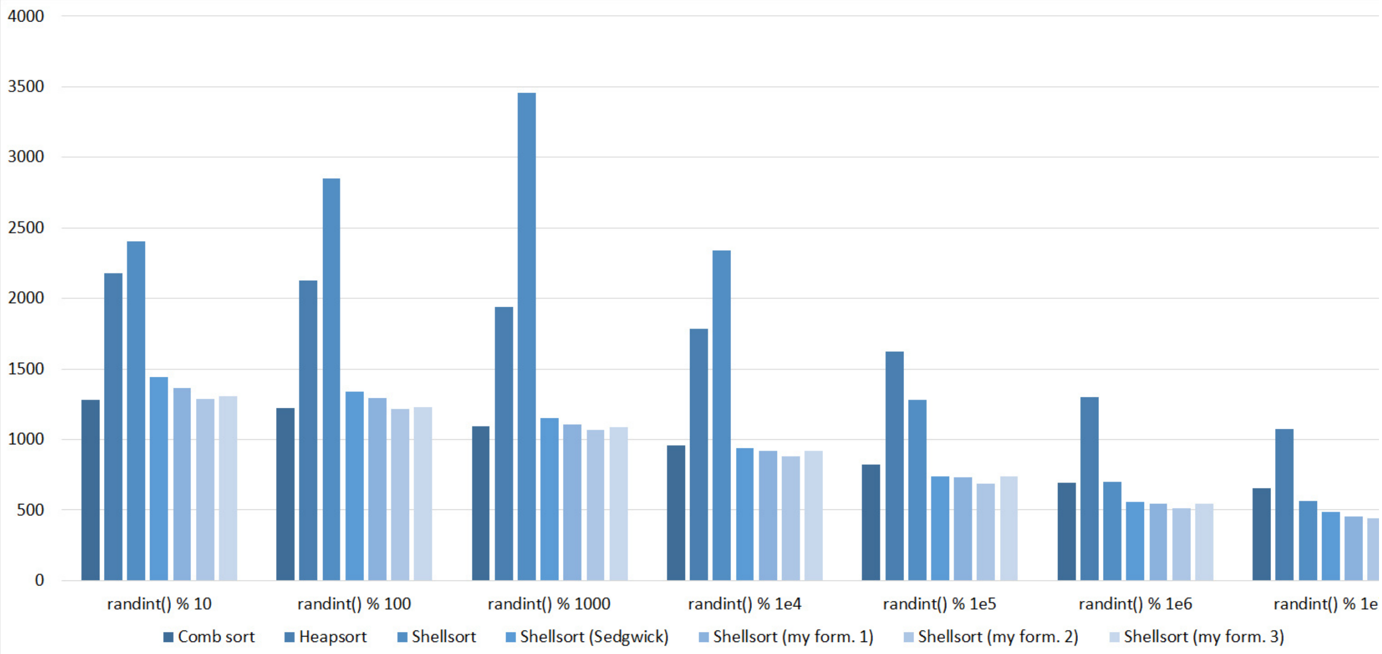
1200
1000
800
600
400
200
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort



1200
1000
800
600
400
200
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


4000
3500
3000
2500
2000
1500
1000
500
0
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e7

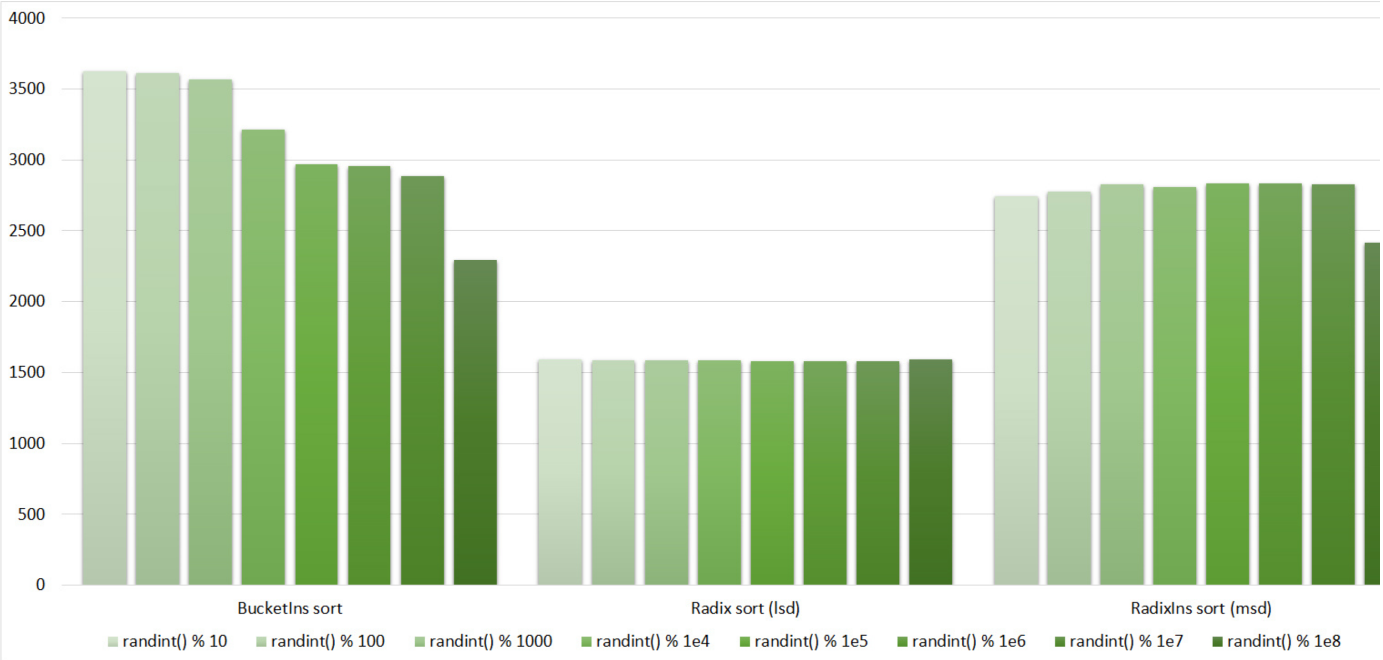

4000
3500
3000
2500
2000
1500
1000
500
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my form. 3)

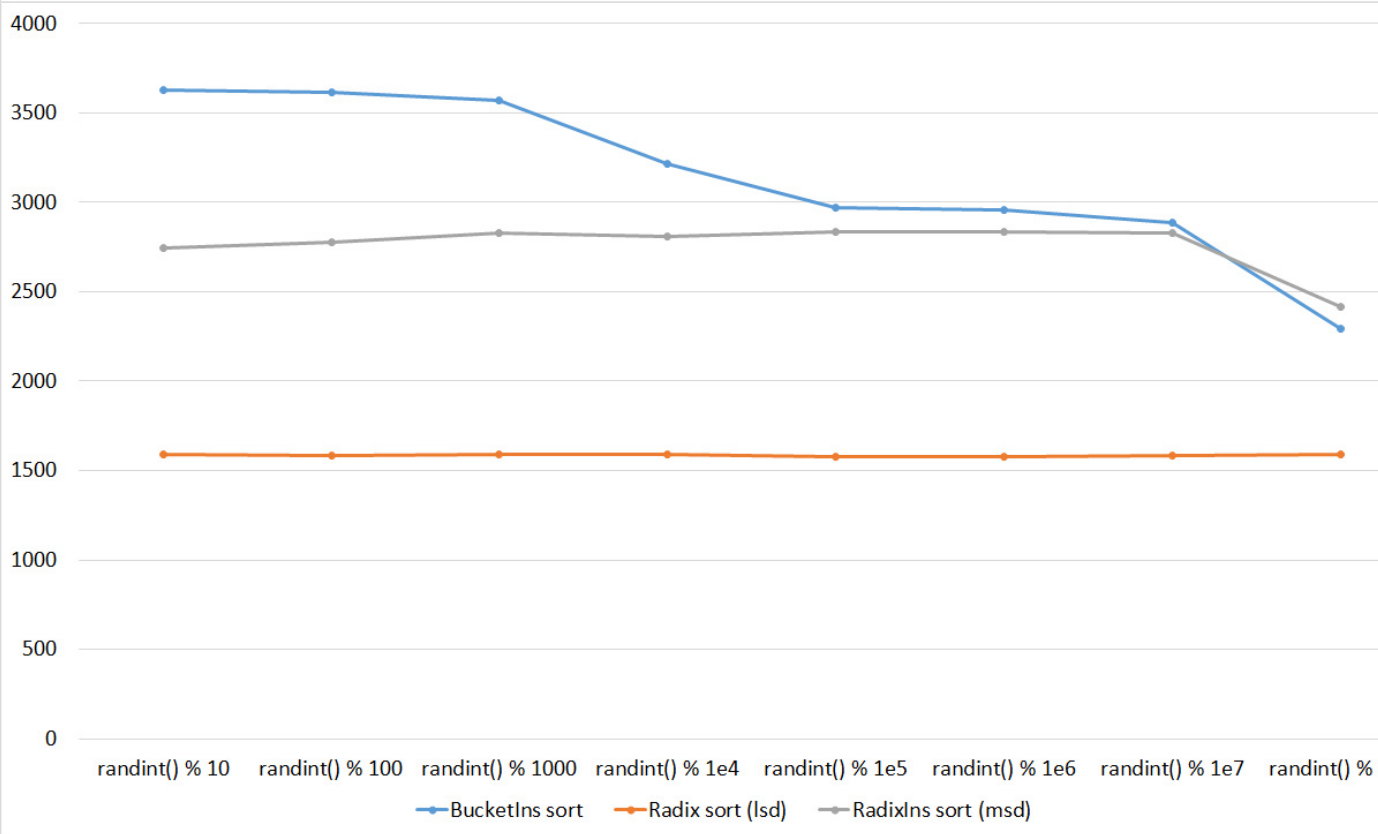


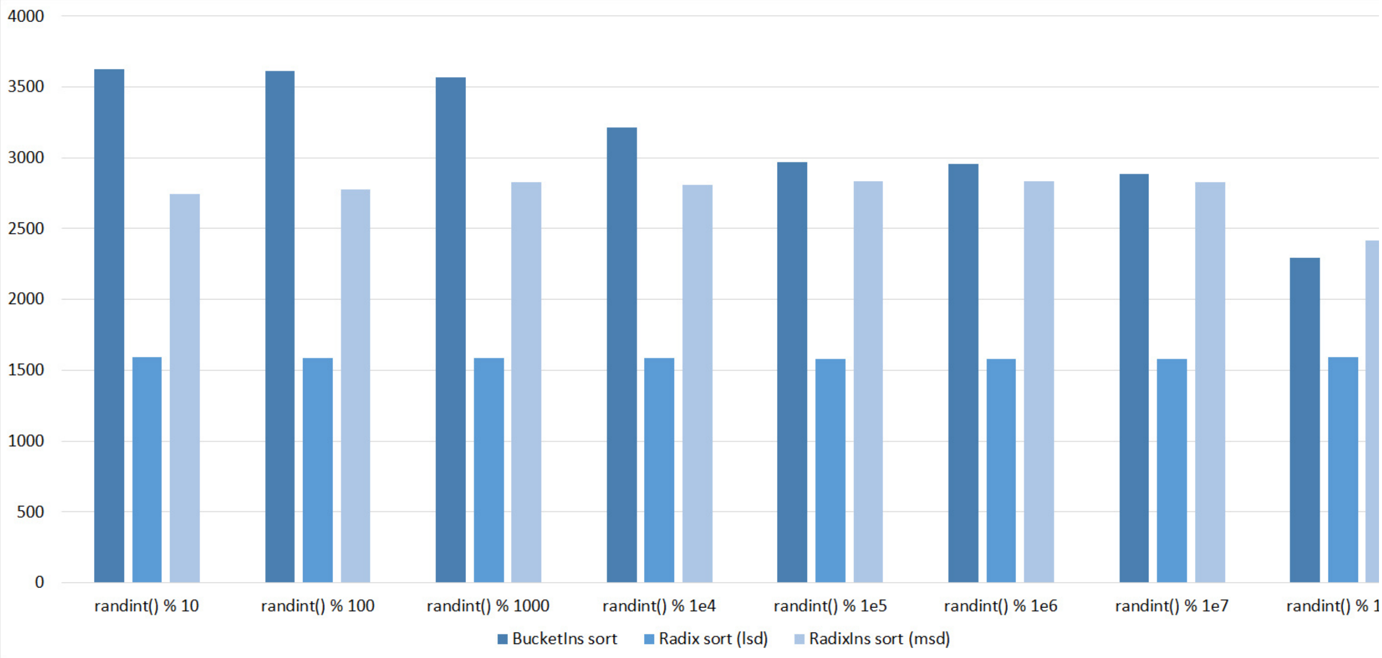
1e7	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	Ито
Comb sort	1283,22	1220,17	1094,9	960,58	822,06	695,34	653,19	961,3
BucketIns sort	257,79	257,18	256,77	234,7	230,25	229,7	213,64	240,0
Heapsort	2176,01	2127,09	1941,94	1784,09	1620,96	1302,36	1071,85	1717,
sort (stl)	837,91	772,87	686,78	598,46	505,03	413,98	356,96	596,0
Quicksort	733,99	690,18	626,81	554,51	480,24	405,41	370,02	551,5
QuickIns sort	662,34	619,15	556,33	484,94	411,19	334,7	302,81	481,6
Merge sort	1020,41	909,4	800,87	698,4	600,76	505,23	432,59	709,6
MergeIns sort	969,91	857,77	732,94	622,51	517,22	418,48	346,24	637,8
Shellsort	2405,7	2846,7	3454,34	2339,92	1281,62	699,37	565,15	1941,
Shellsort (Sedgwick)	1440,43	1339,72	1150,49	939,82	738,22	554,46	487,79	950,1
Radix sort (lsd)	156,44	157,65	156,86	156,96	156,68	156,1	156,78	156,7
RadixIns sort (msd)	184,4	188,28	192,47	191,11	194,91	194,91	186,59	190,3
Timsort	939,27	769,61	599,95	462,68	328,72	196,86	91,64	484,1
Shellsort (my form. 1)	1364,44	1291,51	1108,21	920,21	733,07	541,67	456,47	916,5
Shellsort (my form. 2)	1285,24	1217,32	1070,78	882,82	685,67	511,92	440,24	870,5
Shellsort (my form. 3)	1307,63	1229	1085,53	917,84	740,26	546,78	450,24	896,7

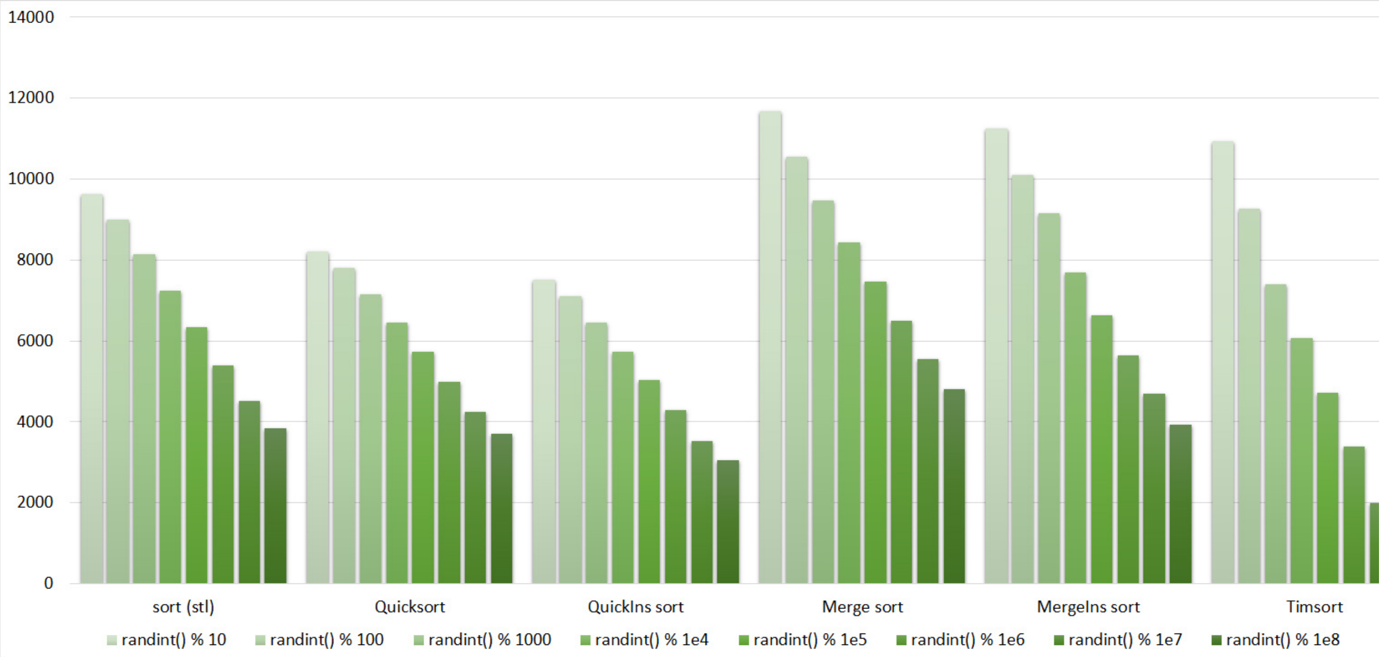
1e7	Итог
Radix sort (lsd)	156,7814286
RadixIns sort (msd)	190,3814286
BucketIns sort	240,0042857
QuickIns sort	481,6371429
Timsort	484,1042857
Quicksort	551,5942857
sort (stl)	595,9985714
MergeIns sort	637,8671429
Merge sort	709,6657143
Shellsort (my form. 2)	870,57
Shellsort (my form. 3)	896,7542857
Shellsort (my form. 1)	916,5114286
Shellsort (Sedgwick)	950,1328571
Comb sort	961,3514286
Heapsort	1717,757143
Shellsort	1941,828571

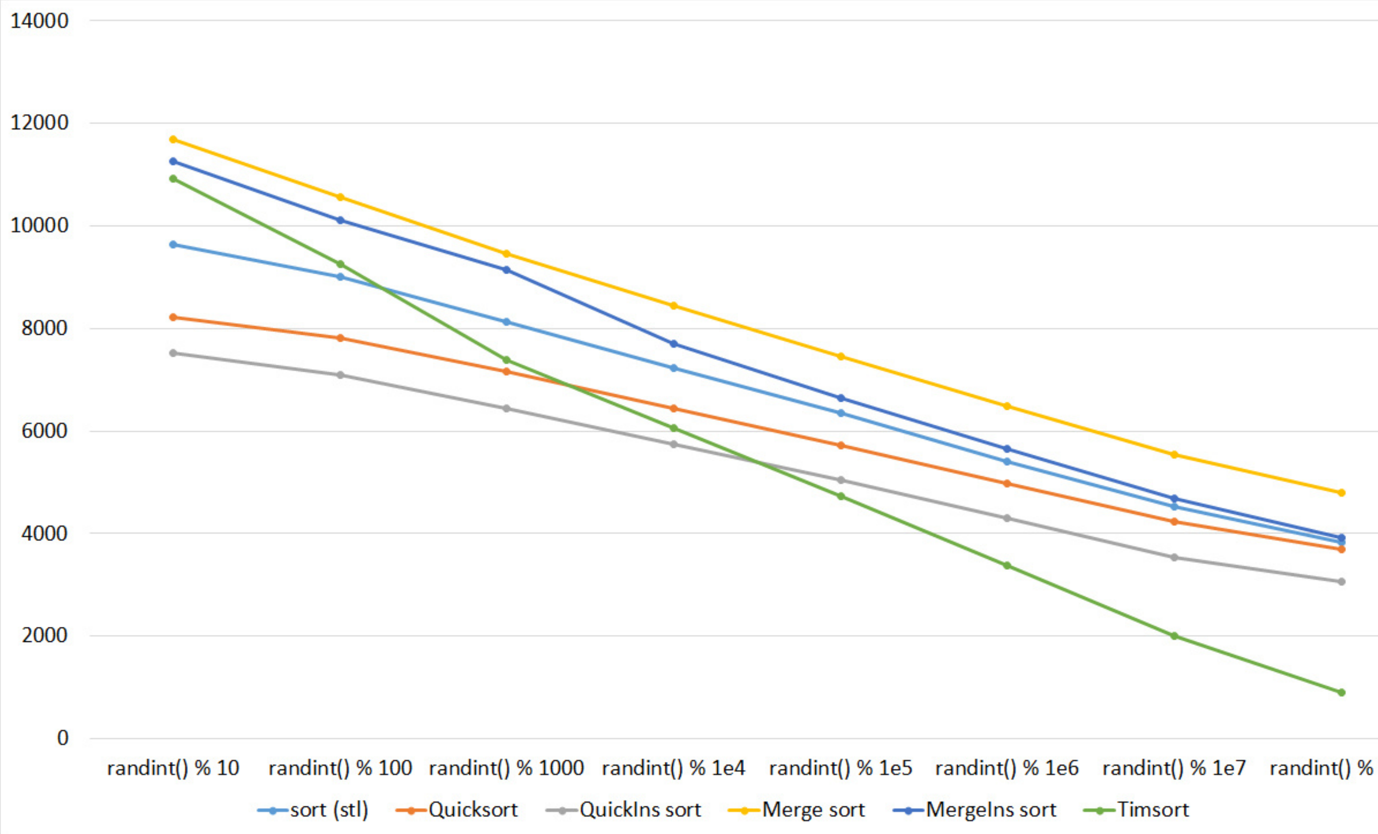
Таблицы, 1e8 элементов

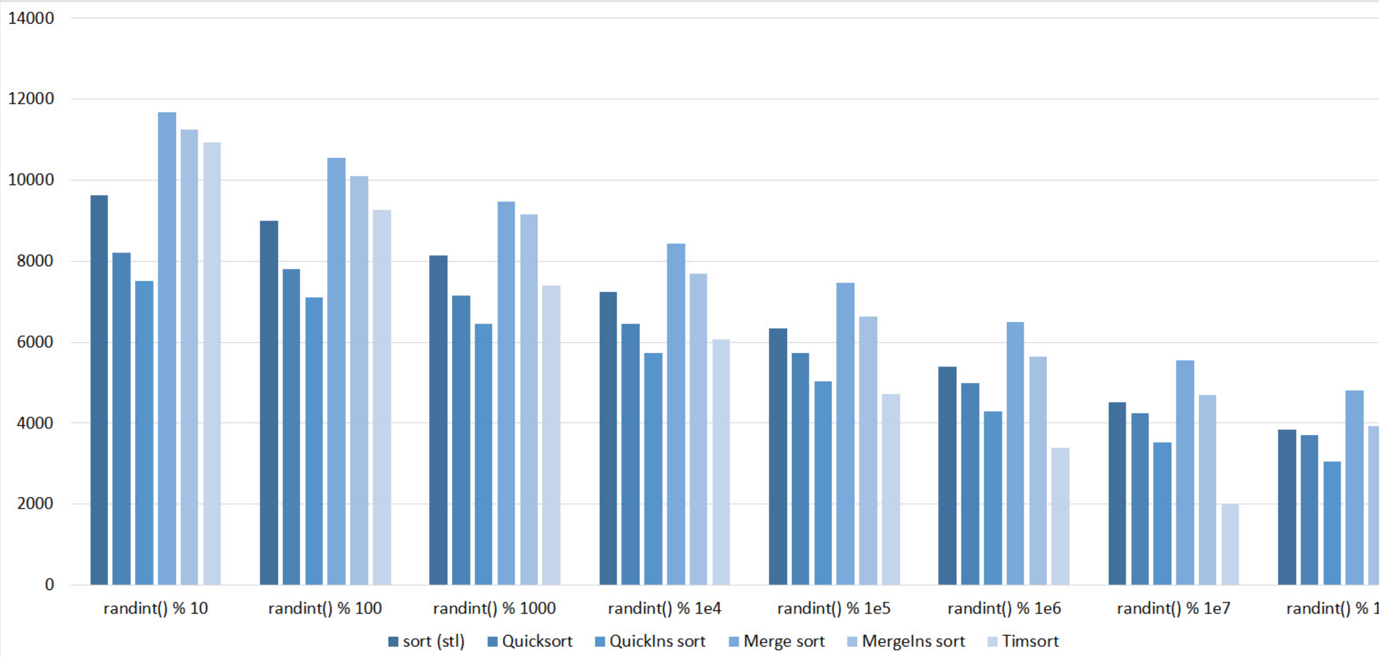

4000
3500
3000
2500
2000
1500
1000
500
0
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e7
randint() % 1e8

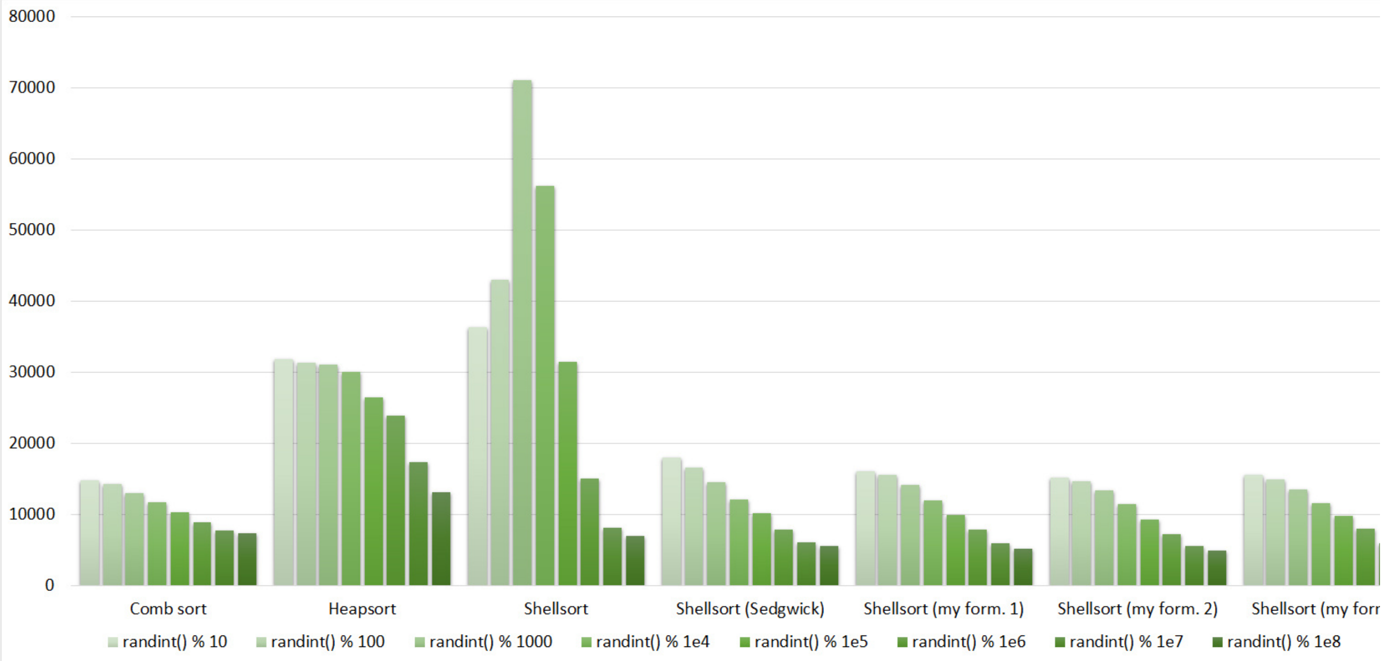

4000
3500
3000
2500
2000
1500
1000
500
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e7
randint() %
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

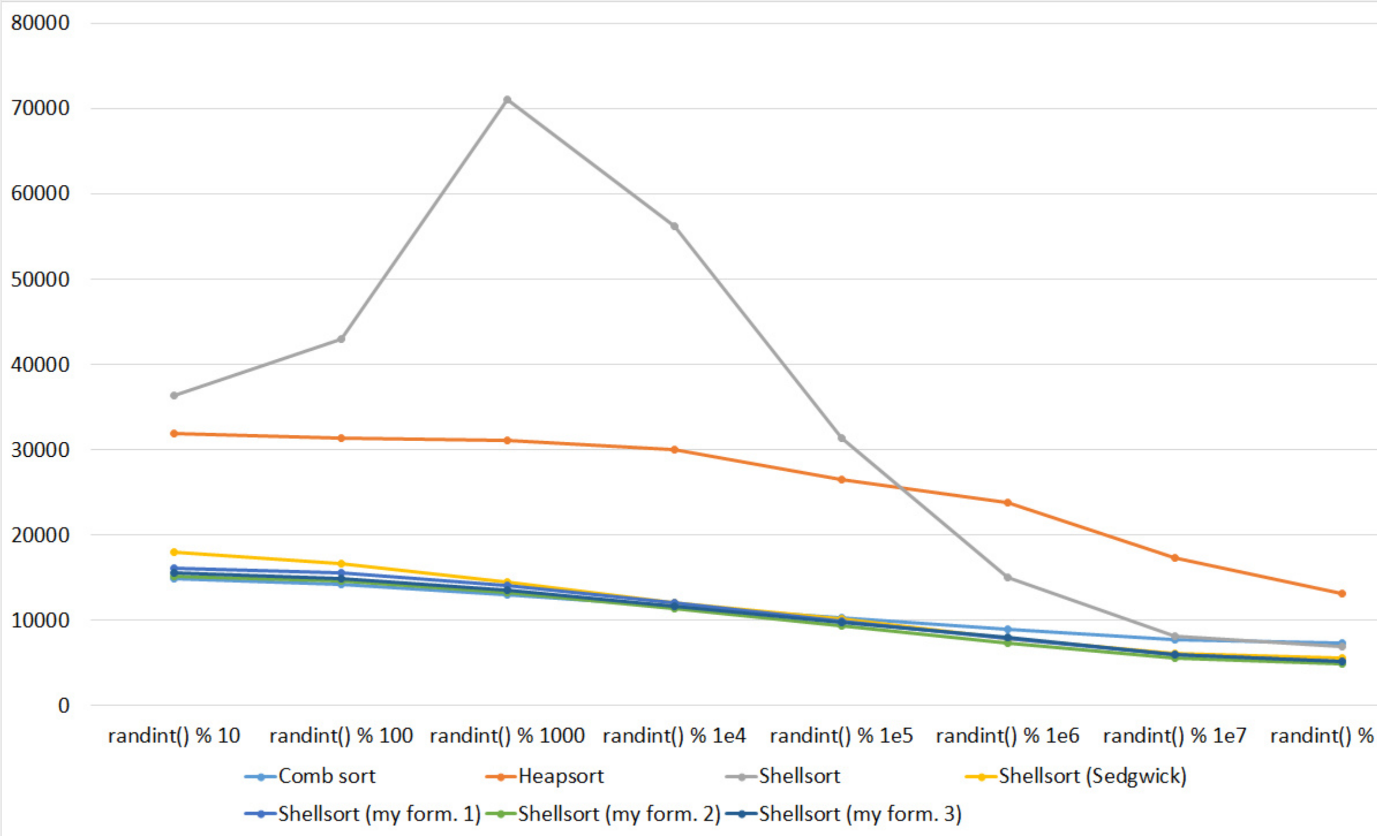

4000
3500
3000
2500
2000
1500
1000
500
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e7
randint() % 1
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

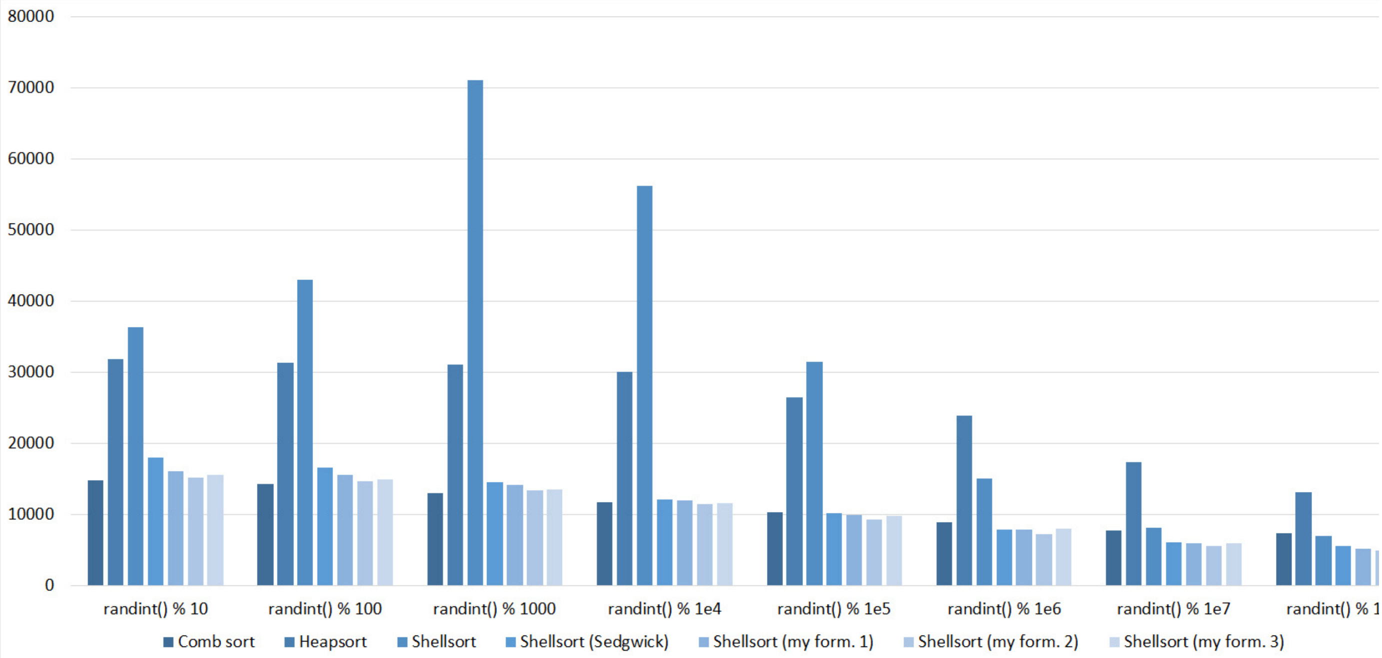

14000
12000
10000
8000
6000
4000
2000
0
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e7
randint() % 1e8


14000
12000
10000
8000
6000
4000
2000
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e7
randint() %
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


14000
12000
10000
8000
6000
4000
2000
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e7
randint() % 1
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


80000
70000
60000
50000
40000
30000
20000
10000
0
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e7
randint() % 1e8


80000
70000
60000
50000
40000
30000
20000
10000
0
randint() % 10
randint() % 100
randint() % 1000
randint() % 1e4
randint() % 1e5
randint() % 1e6
randint() % 1e7
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my form. 3)



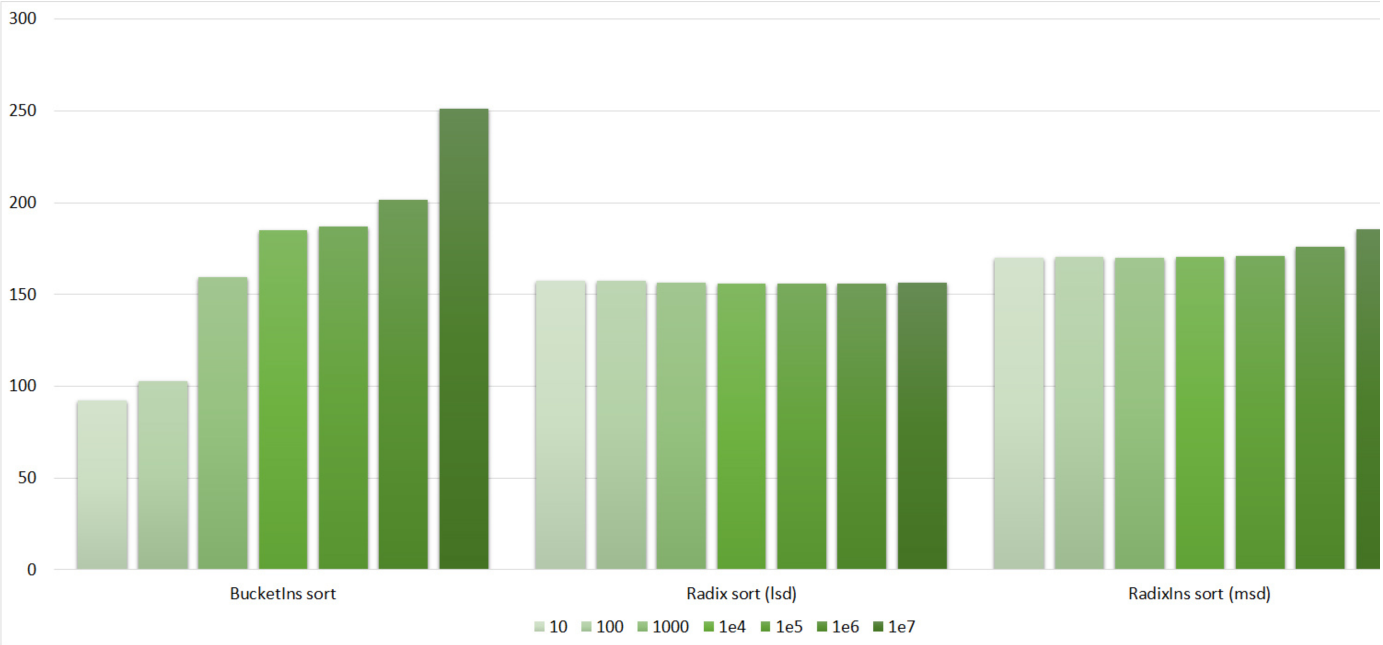
1e8	randint() % 10	randint() % 100	randint() % 1000	randint() % 1e4	randint() % 1e5	randint() % 1e6	randint() % 1e7	randint() % 1e8	Ит
Comb sort	14871,94	14258,90	13005,51	11671,18	10301,96	8910,10	7719,03	7390,51	1101
BucketIns sort	3625,75	3614,65	3568,46	3212,97	2967,55	2956,98	2884,69	2293,16	314
Heapsort	31876,16	31390,71	31116,75	30068,75	26478,36	23856,83	17323,63	13183,21	2566
sort (stl)	9634,56	8997,37	8139,04	7229,49	6344,47	5404,23	4517,81	3832,72	676
Quicksort	8218,65	7805,60	7150,87	6446,33	5730,49	4980,30	4234,95	3697,34	603
QuickIns sort	7516,39	7101,18	6448,16	5738,35	5034,20	4293,00	3527,67	3052,63	533
Merge sort	11684,14	10552,64	9467,94	8438,64	7456,49	6491,41	5541,72	4803,30	805
MergeIns sort	11257,35	10106,04	9148,66	7693,59	6642,82	5643,55	4692,07	3927,76	738
Shellsort	36385,72	43029,63	71122,36	56245,04	31428,09	15039,60	8113,65	6966,76	3354
Shellsort (Sedgwick)	17976,67	16658,26	14504,07	12103,96	10140,11	7912,57	6095,09	5535,37	1136
Radix sort (lsd)	1591,20	1582,91	1587,03	1588,98	1579,62	1579,96	1580,74	1591,14	158
RadixIns sort (msd)	2744,31	2779,41	2825,62	2808,98	2832,41	2833,08	2829,75	2417,27	275
Timsort	10933,18	9259,29	7397,37	6061,19	4721,67	3380,21	1999,61	901,16	558
Shellsort (my form. 1)	16092,15	15592,52	14112,86	12026,71	9945,46	7905,32	5936,45	5196,07	1085
Shellsort (my form. 2)	15242,49	14687,78	13337,28	11427,31	9356,83	7300,34	5600,08	4958,75	1023
Shellsort (my form. 3)	15617,02	14917,72	13505,82	11653,72	9759,24	7955,91	6001,66	5124,81	1056

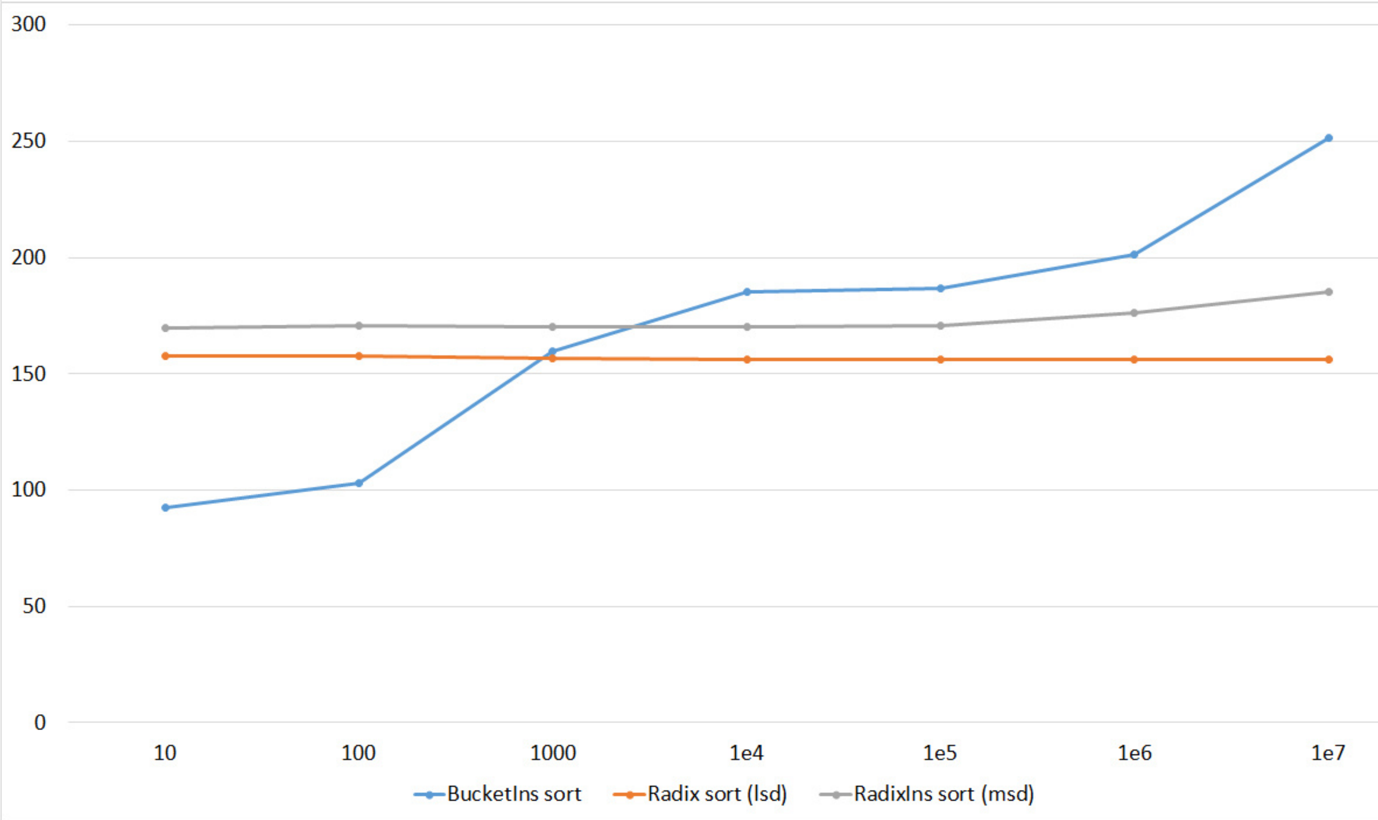
1e8	Итог
Radix sort (lsd)	1585,20
RadixIns sort (msd)	2758,85
BucketIns sort	3140,53
QuickIns sort	5338,95
Timsort	5581,71
Quicksort	6033,07
sort (stl)	6762,46
MergeIns sort	7388,98
Merge sort	8054,54
Shellsort (my form. 2)	10238,86
Shellsort (my form. 3)	10566,99
Shellsort (my form. 1)	10850,94
Comb sort	11016,14
Shellsort (Sedgwick)	11365,76
Heapsort	25661,80
Shellsort	33541,36

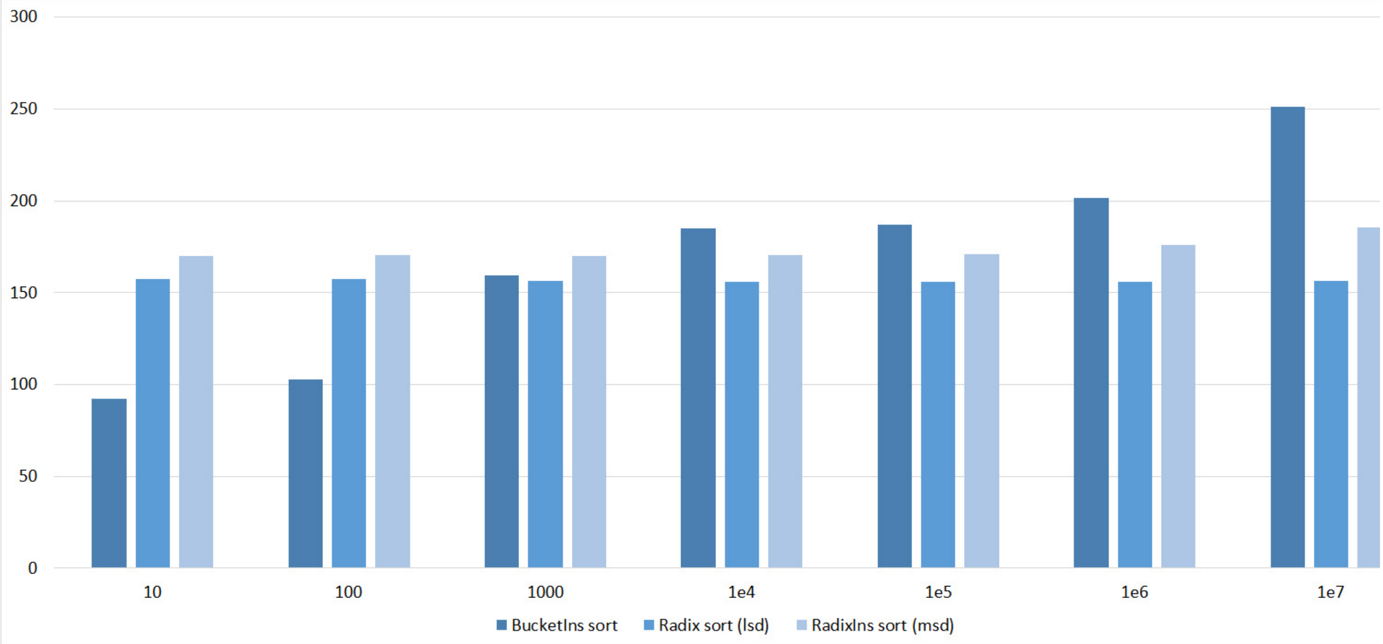
Здесь все тоже довольно понятно. Стало заметен алгоритм Timsort, на него отсортированность действует сильнее, чем на остальные. Это позволило этому алгоритму почти сравняться с оптимизированной версией быстрой сортировки. Блочная сортировка, несмотря на улучше времени работы при частичной отсортированности, не смогла обогнать поразрядную сортировку.

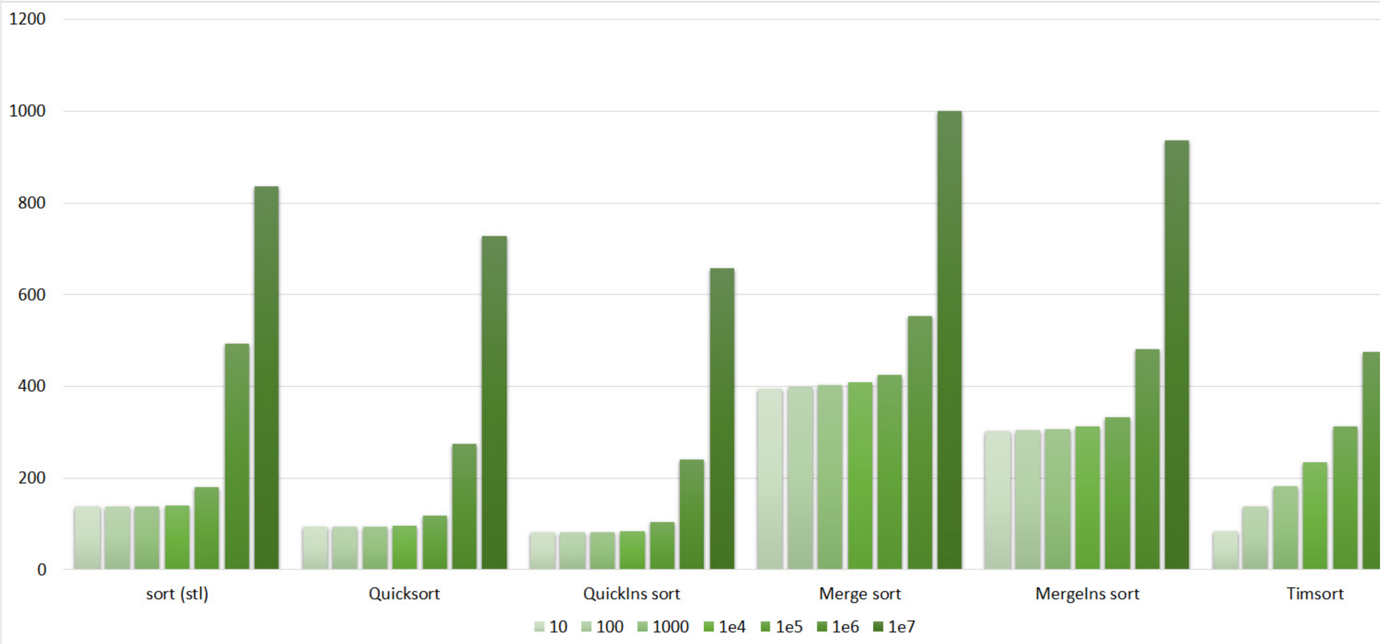
Свопы

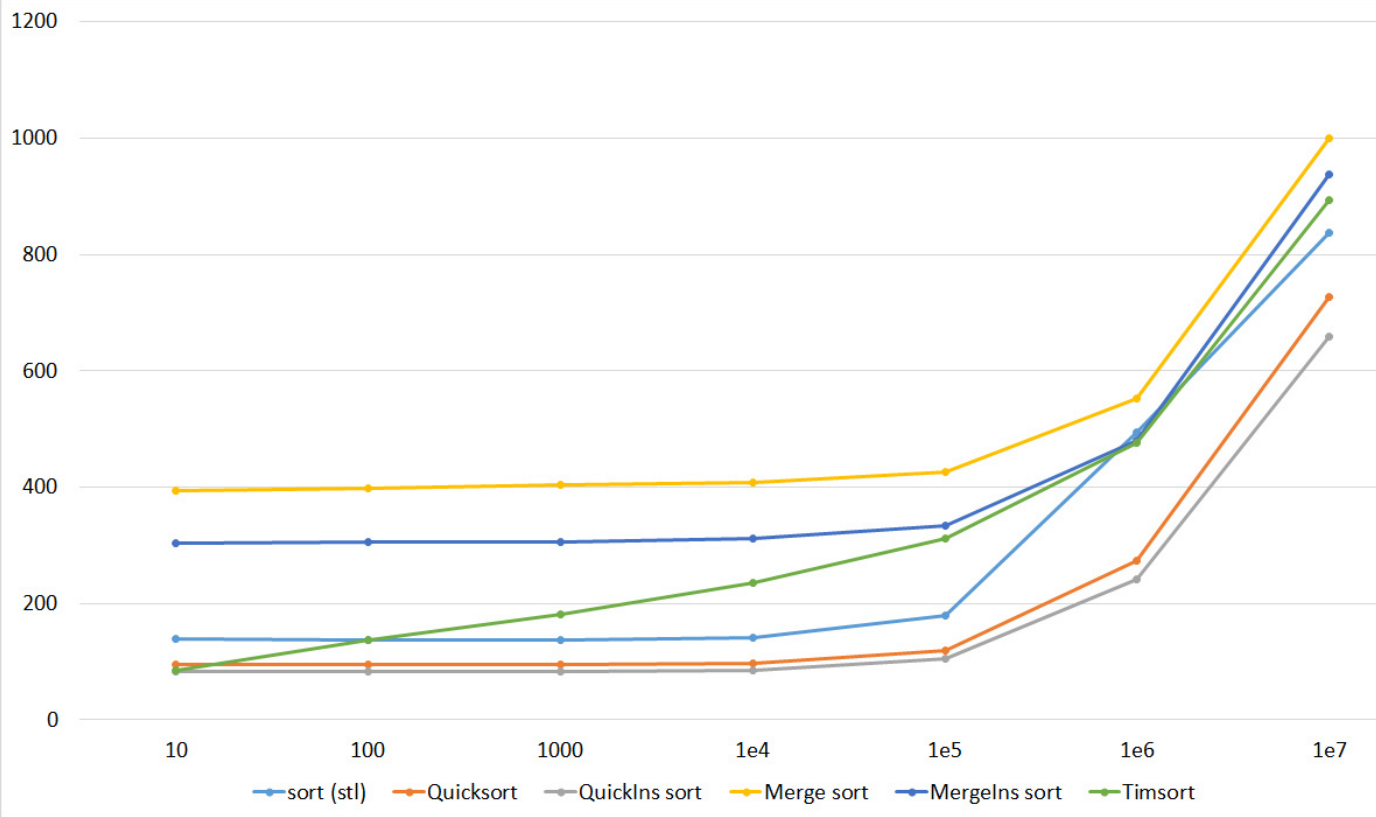
Таблицы, 1e7 элементов

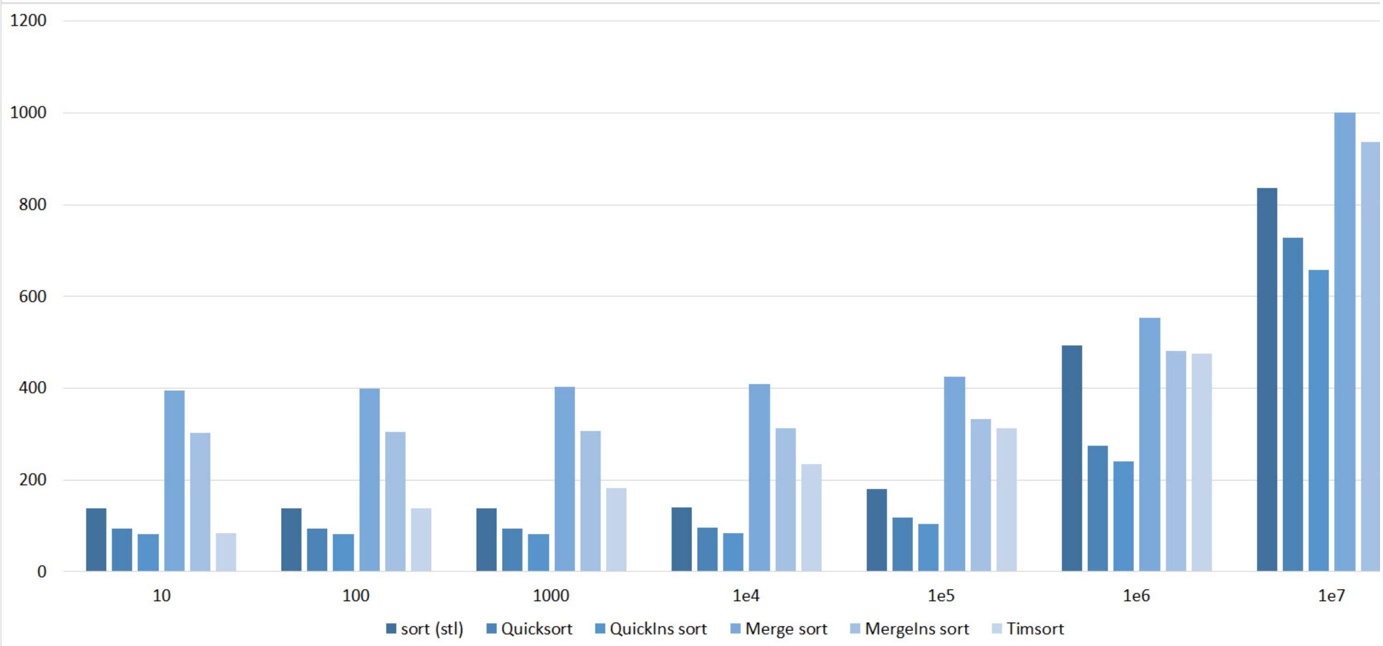


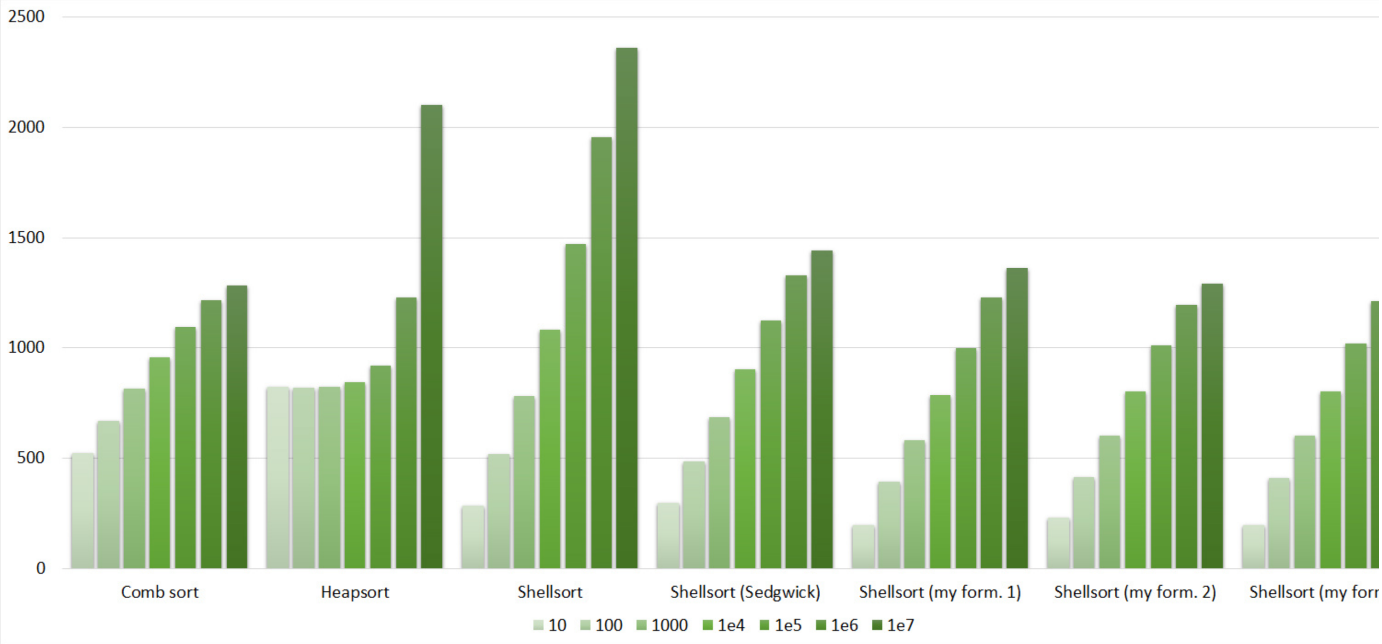


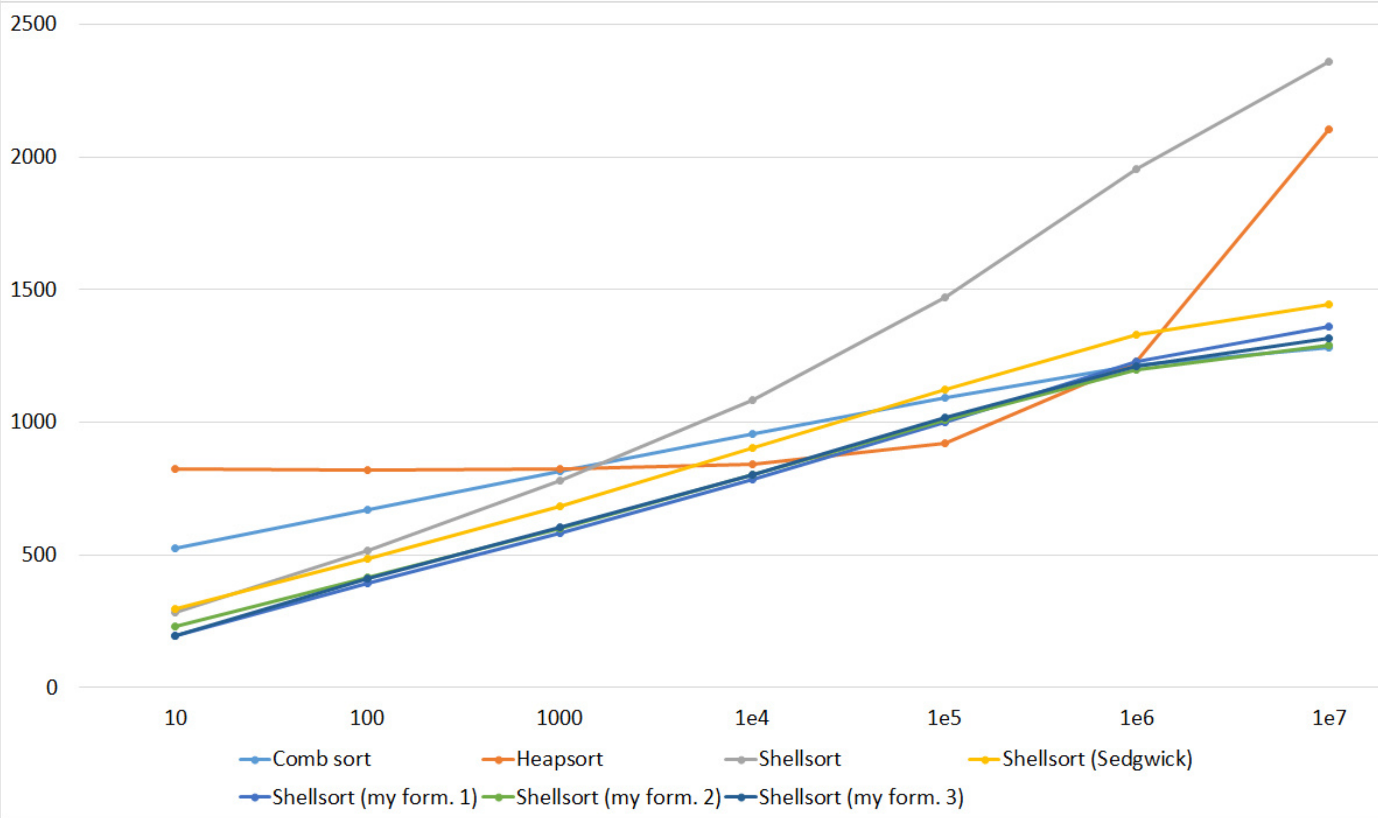

300
250
200
150
100
50
0
10
100
1000
1e4
1e5
1e6
1e7
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

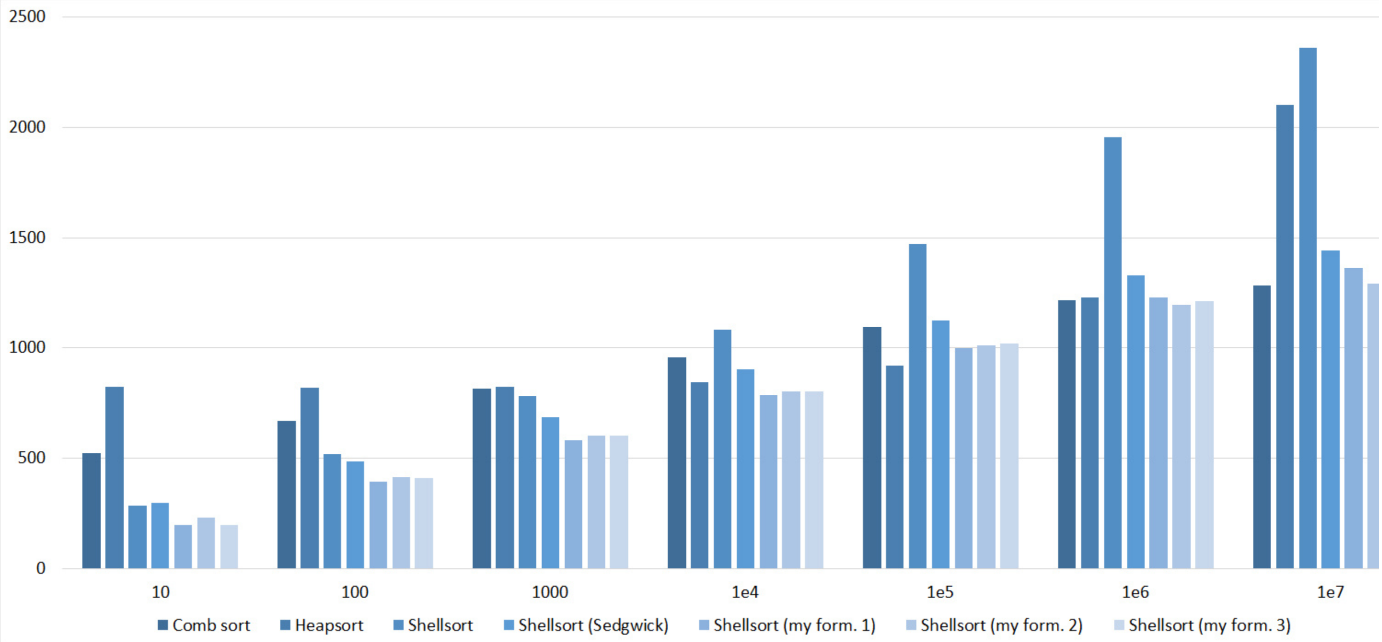

1200
1000
800
600
400
200
0
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort
10
100
1000
1e4
1e5
1e6
1e7


1200
1000
800
600
400
200
0
10
100
1000
1e4
1e5
1e6
1e7
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


1200
1000
800
600
400
200
0
10
100
1000
1e4
1e5
1e6
1e7
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


2500
2000
1500
1000
500
0
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my forr
10
100
1000
1e4
1e5
1e6
1e7

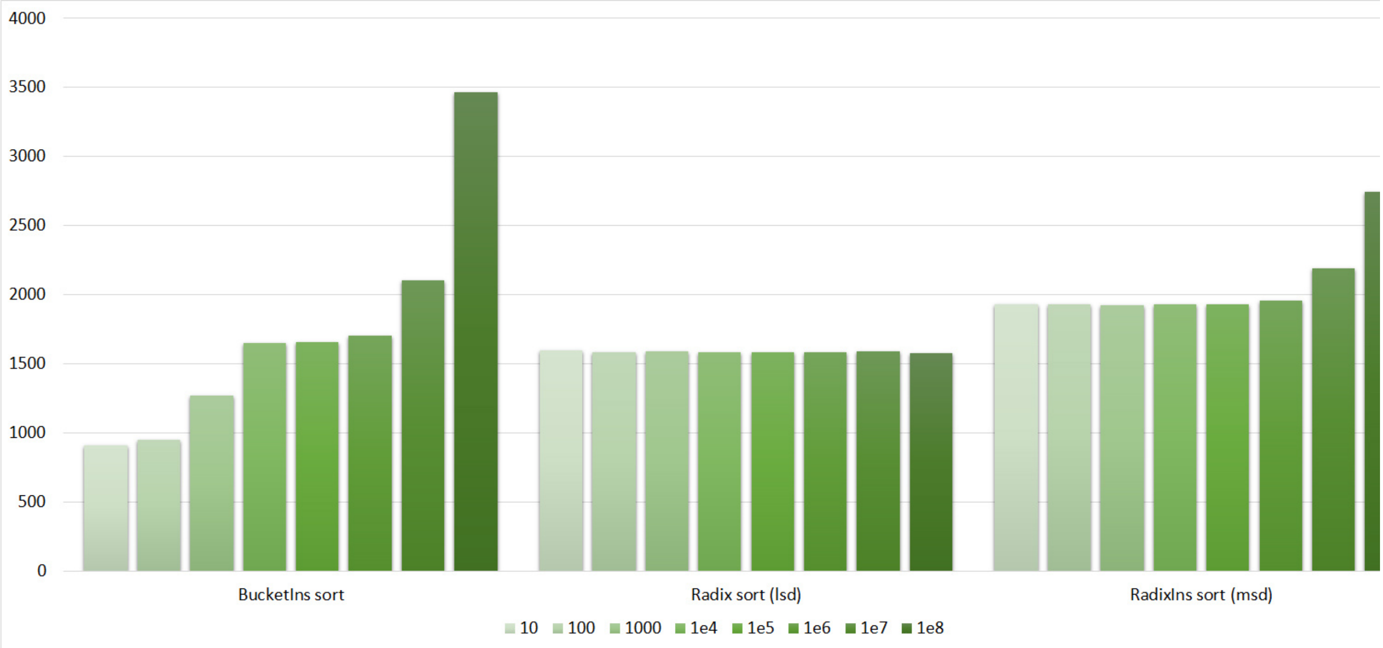

2500
2000
1500
1000
500
0
10
100
1000
1e4
1e5
1e6
1e7
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my form. 3)

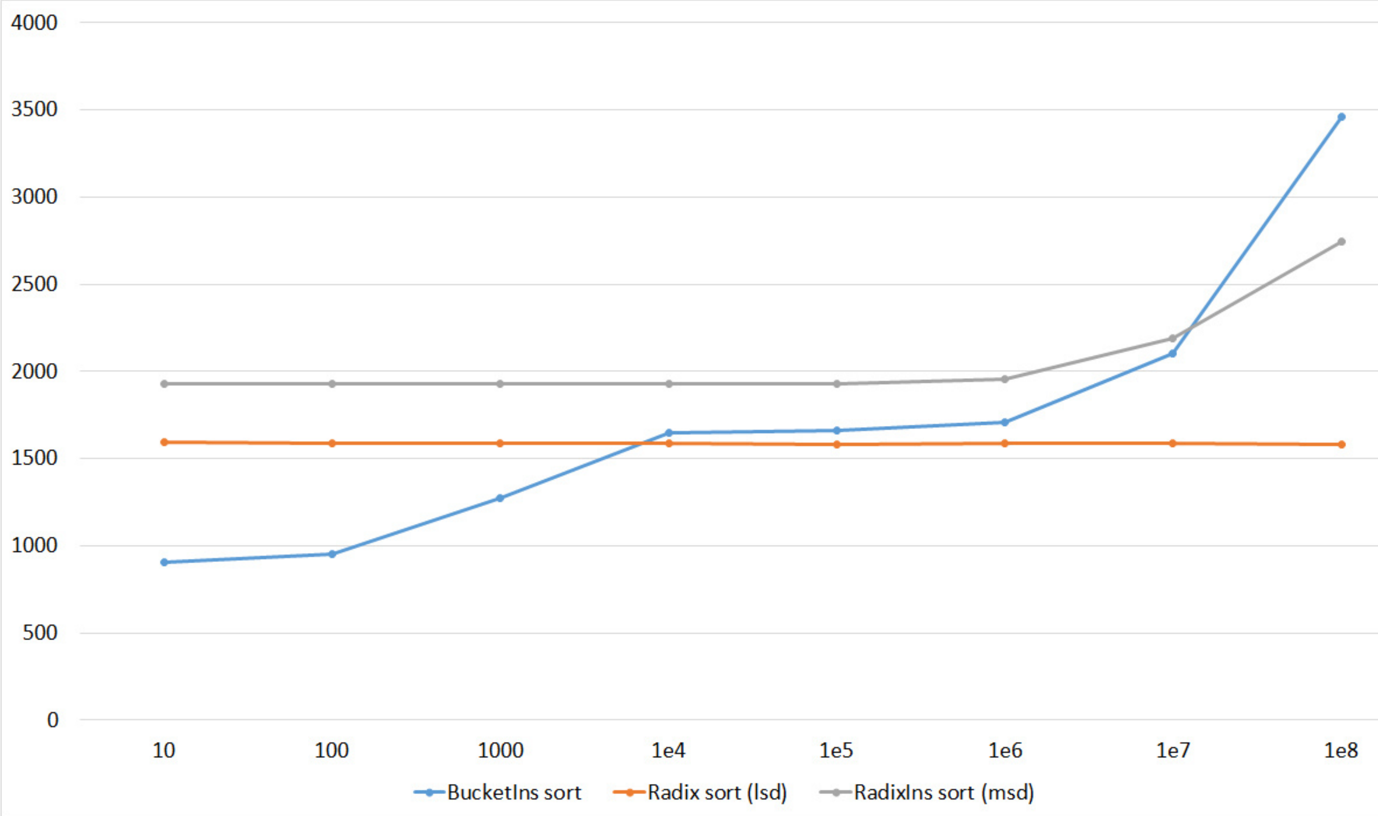


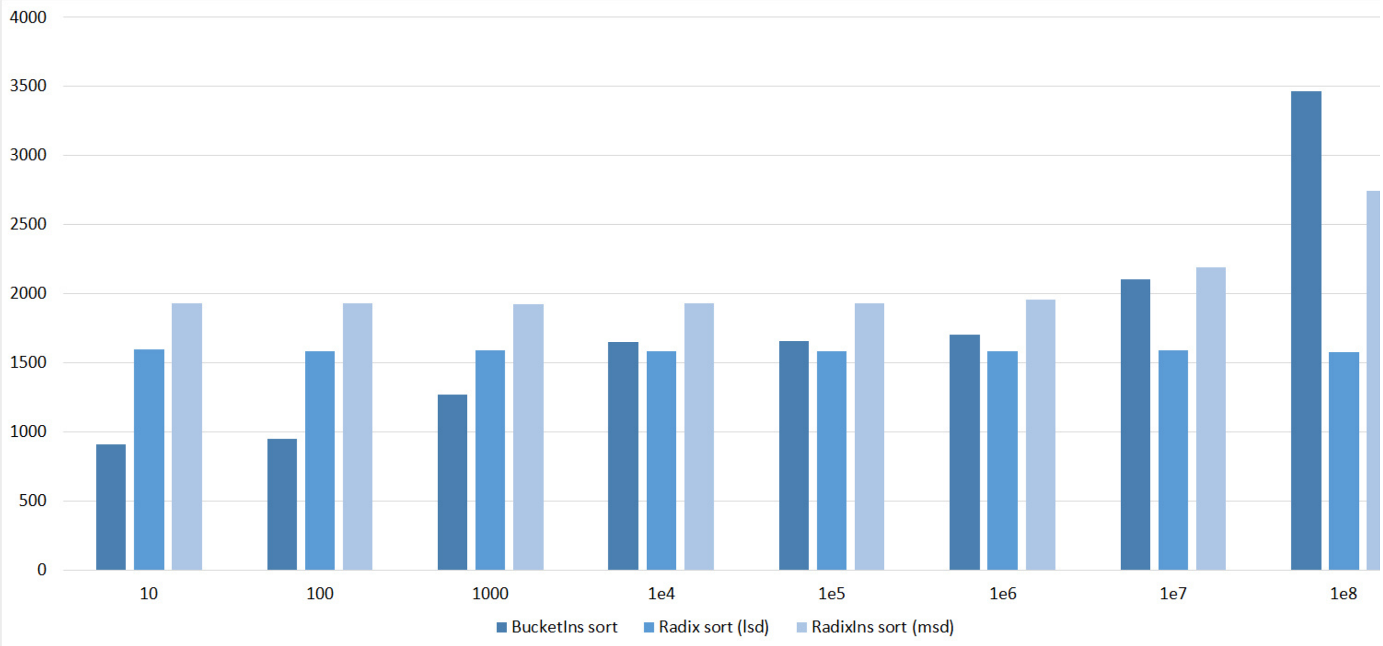
1e7	10	100	1000	1e4	1e5	1e6	1e7	Итог
Comb sort	523,06	669,5	814,43	955,95	1094,28	1217,91	1281,45	936,65
BucketIns sort	92,38	102,88	159,57	185,09	186,84	201,51	251,22	168,50
Heapsort	822,43	819,63	822,26	843,34	919,35	1228,43	2103,36	1079,83
sort (stl)	138,48	138,13	138,11	140,8	179,5	494,38	836,58	295,14
Quicksort	94,35	94,4	94,51	96,53	118,89	274,28	727,47	214,35
QuickIns sort	82,57	82,61	82,62	84,63	105,05	241,19	658,82	191,07
Merge sort	394,72	398,44	403,22	408,33	425,61	553,2	1000,63	512,02
MergeIns sort	303,84	304,9	306,27	312,38	332,99	480,84	937,02	425,46
Shellsort	283,46	517,93	781,05	1084,11	1471,54	1954,92	2361,94	1207,85
Shellsort (Sedgwick)	296,89	486,73	684,4	902,2	1124,42	1329,82	1443,73	895,46
Radix sort (lsd)	157,61	157,48	156,52	156,1	156,08	156,03	156,29	156,59
RadixIns sort (msd)	169,94	170,58	170,02	170,4	170,9	176,2	185,38	173,35
Timsort	84,75	138,18	182,28	234,82	312,52	475,4	893,19	331,59
Shellsort (my form. 1)	196,97	392,1	582,25	786,14	999,92	1229,44	1360,91	792,53
Shellsort (my form. 2)	229,1	413,38	601,97	803,84	1010,03	1196,97	1290,97	792,32
Shellsort (my form. 3)	196,31	409,66	603,49	804,15	1018,26	1211,95	1316,41	794,32

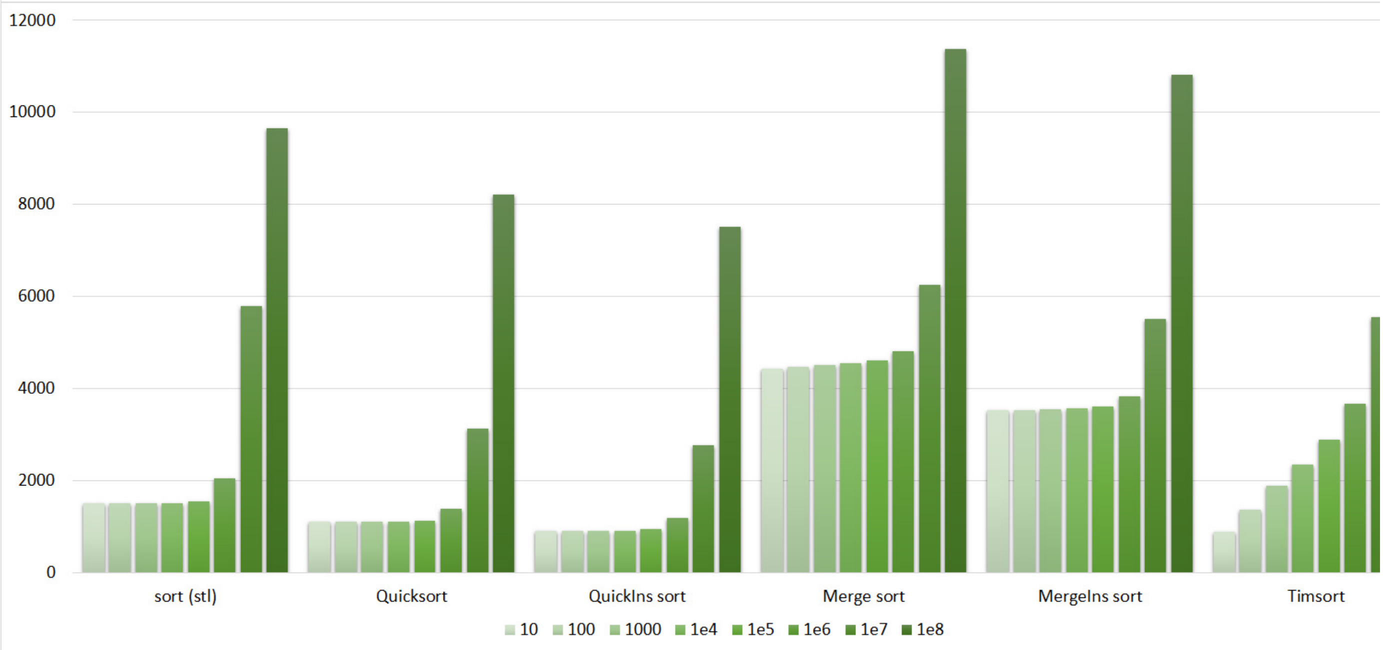
1e7	Итог
Radix sort (lsd)	156,59
BucketIns sort	168,50
RadixIns sort (msd)	173,35
QuickIns sort	191,07
Quicksort	214,35
sort (stl)	295,14
Timsort	331,59
MergeIns sort	425,46
Merge sort	512,02
Shellsort (my form. 2)	792,32
Shellsort (my form. 1)	792,53
Shellsort (my form. 3)	794,32
Shellsort (Sedgwick)	895,46
Comb sort	936,65
Heapsort	1079,83
Shellsort	1207,85

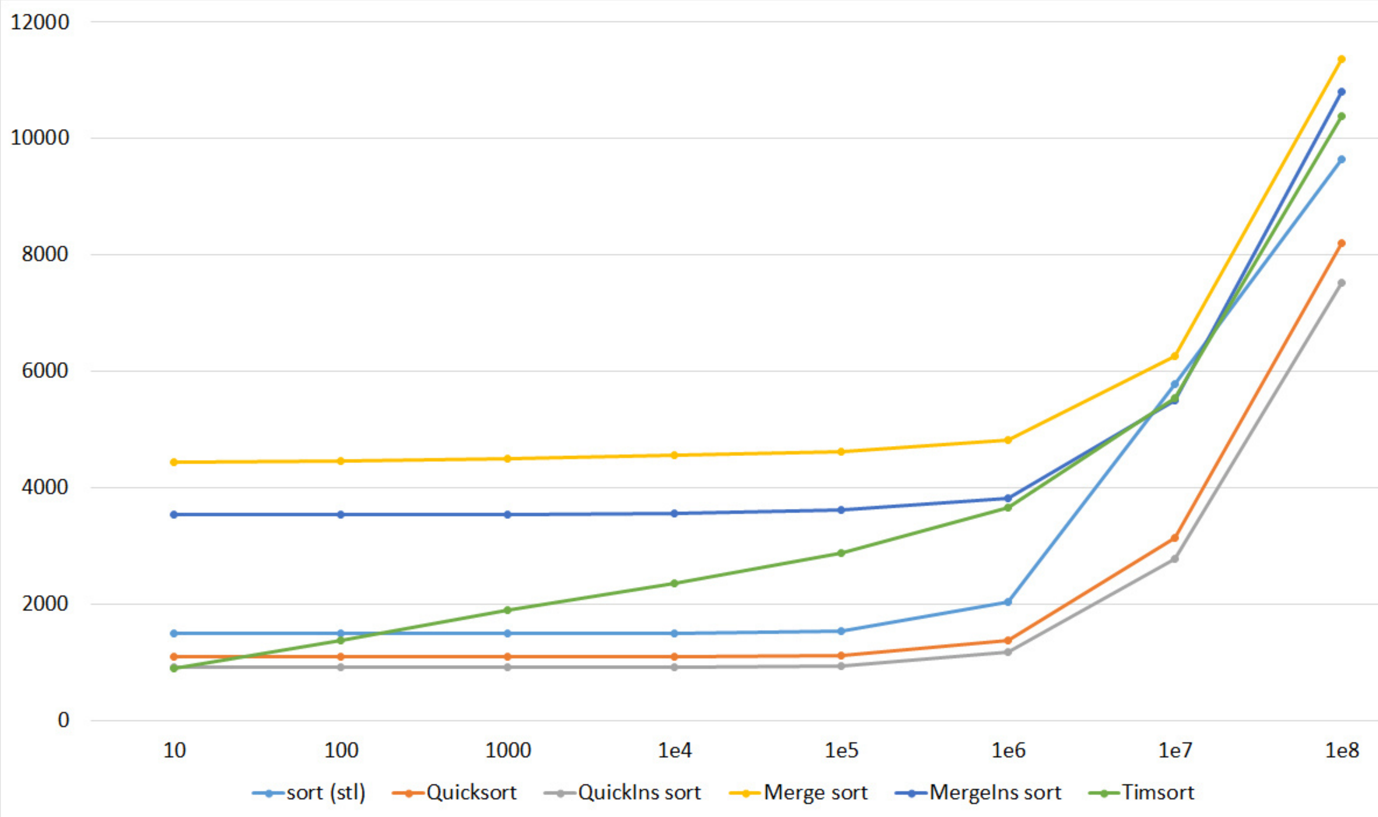
Таблицы, 1e8 элементов

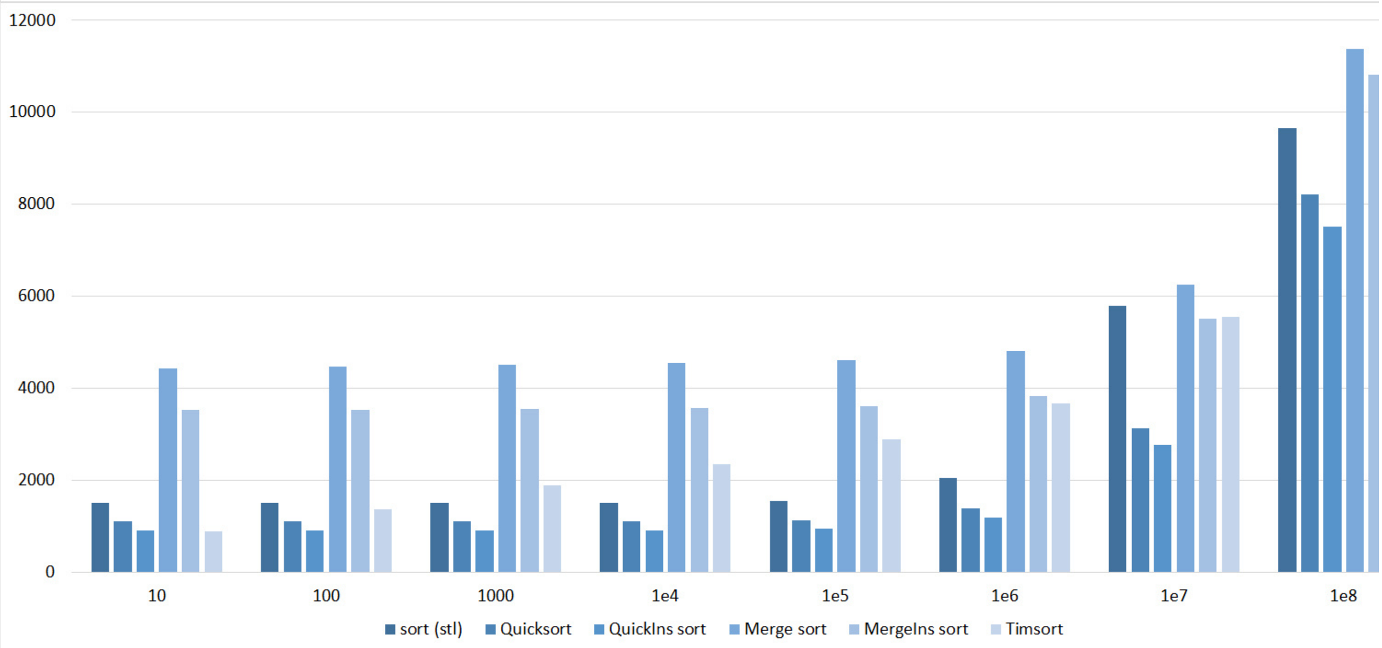

4000
3500
3000
2500
2000
1500
1000
500
0
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)
10 100 1000 1e4 1e5 1e6 1e7 1e8

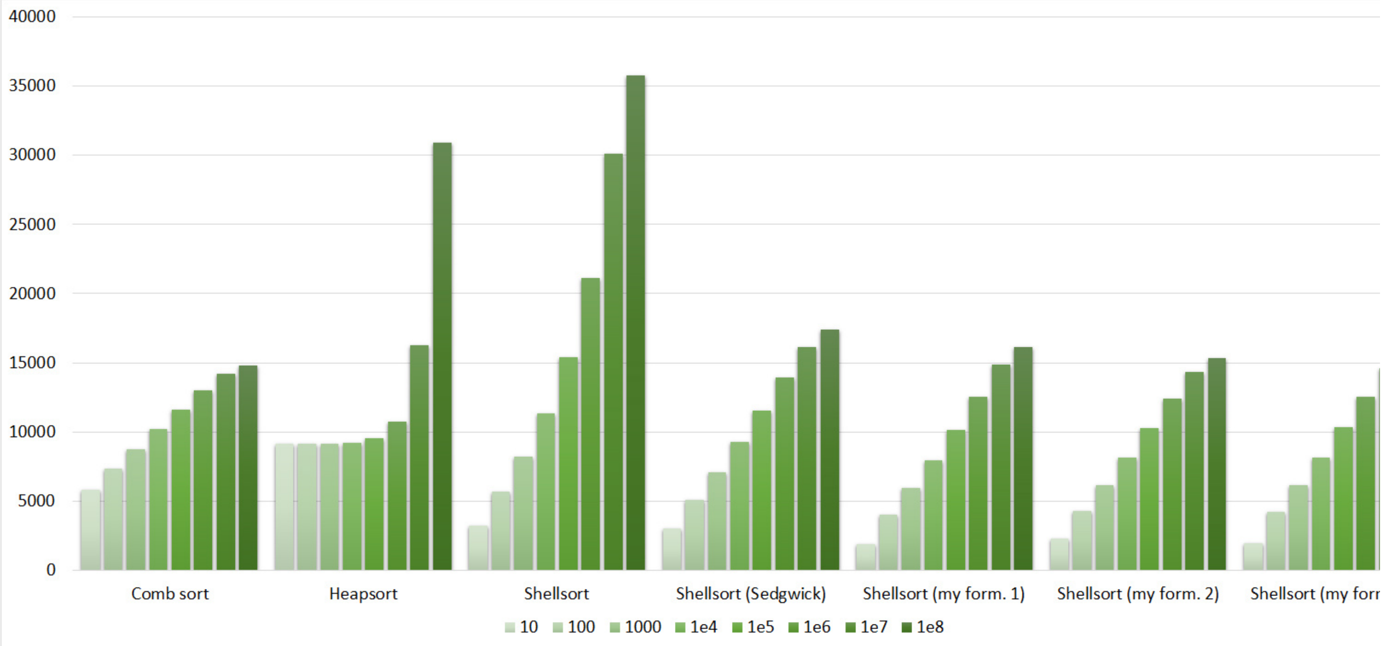

4000
3500
3000
2500
2000
1500
1000
500
0
10 100 1000 1e4 1e5 1e6 1e7 1e8
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

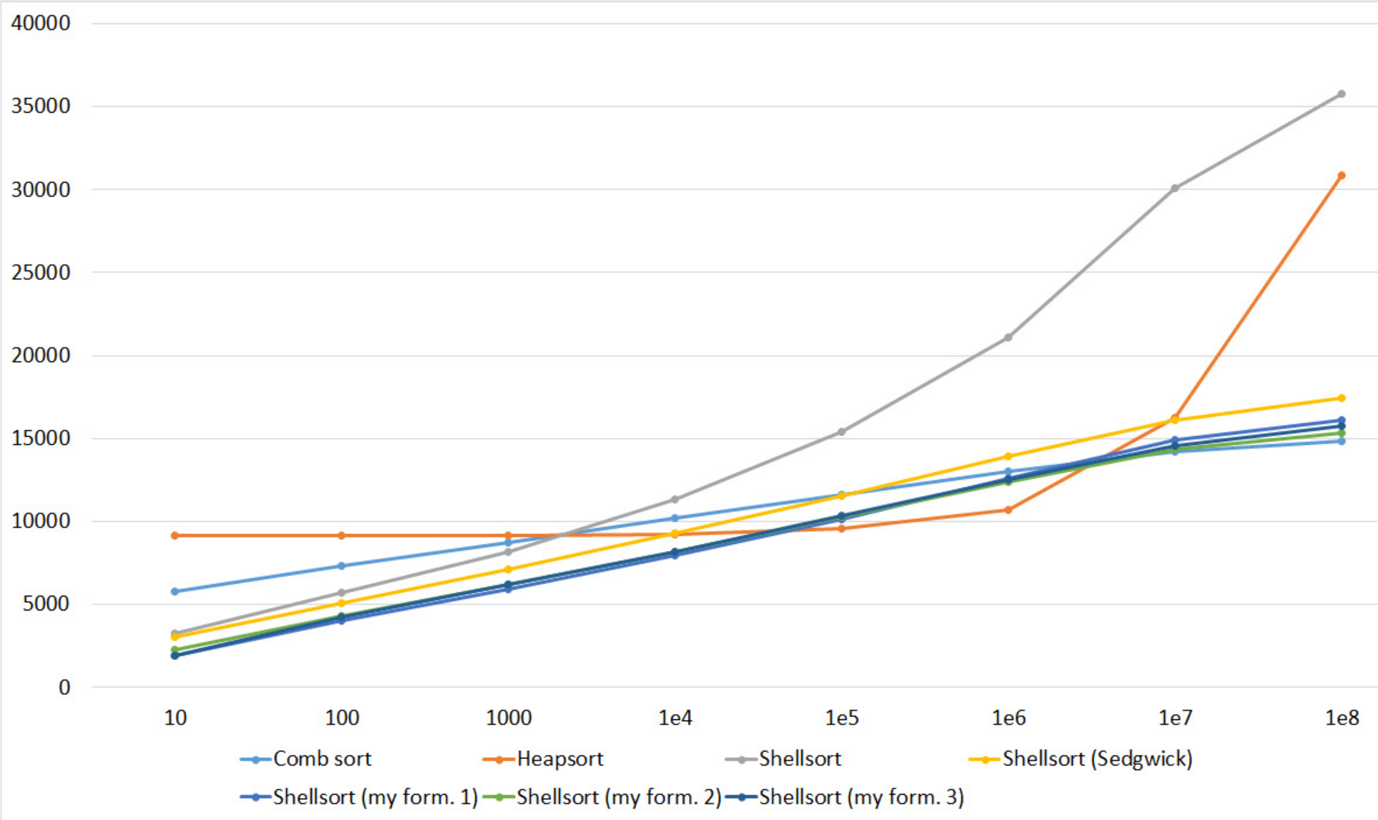

4000
3500
3000
2500
2000
1500
1000
500
0
10
100
1000
1e4
1e5
1e6
1e7
1e8
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

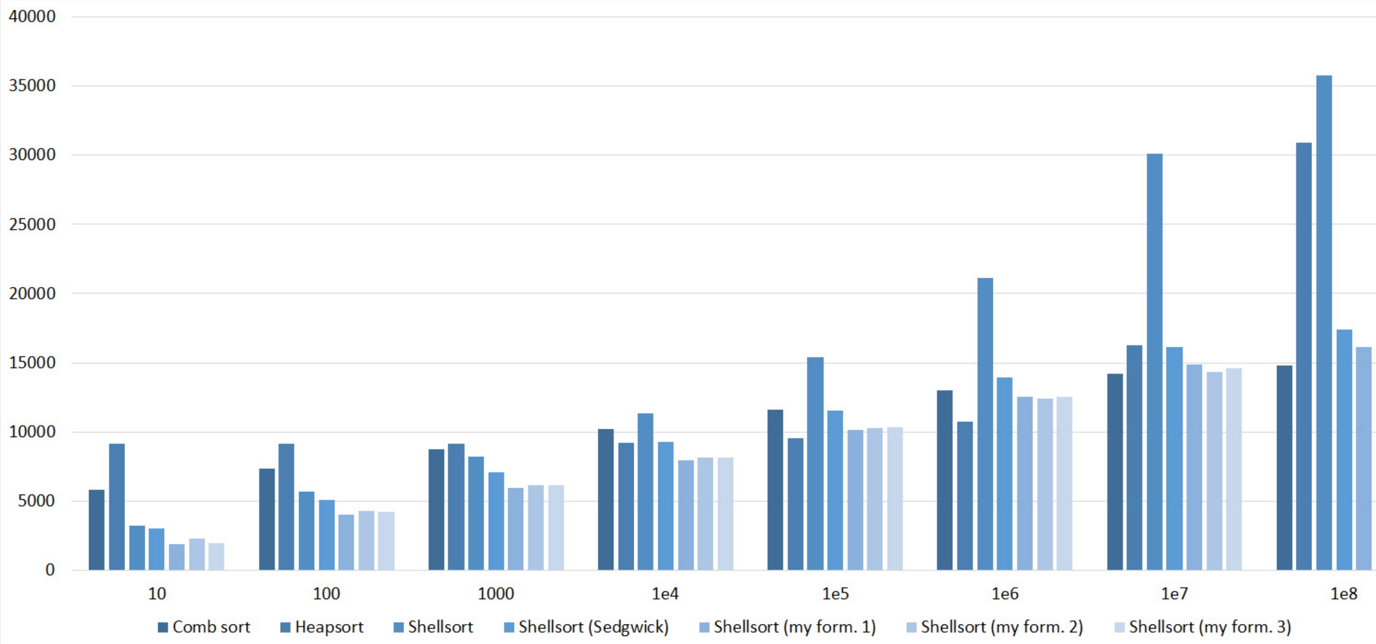

12000
10000
8000
6000
4000
2000
0
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort
10
100
1000
1e4
1e5
1e6
1e7
1e8


12000
10000
8000
6000
4000
2000
0
10
100
1000
1e4
1e5
1e6
1e7
1e8
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


12000
10000
8000
6000
4000
2000
0
10
100
1000
1e4
1e5
1e6
1e7
1e8
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


40000
35000
30000
25000
20000
15000
10000
5000
0
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my forr
10
100
1000
1e4
1e5
1e6
1e7
1e8


40000
35000
30000
25000
20000
15000
10000
5000
0
10
100
1000
1e4
1e5
1e6
1e7
1e8
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my form. 3)



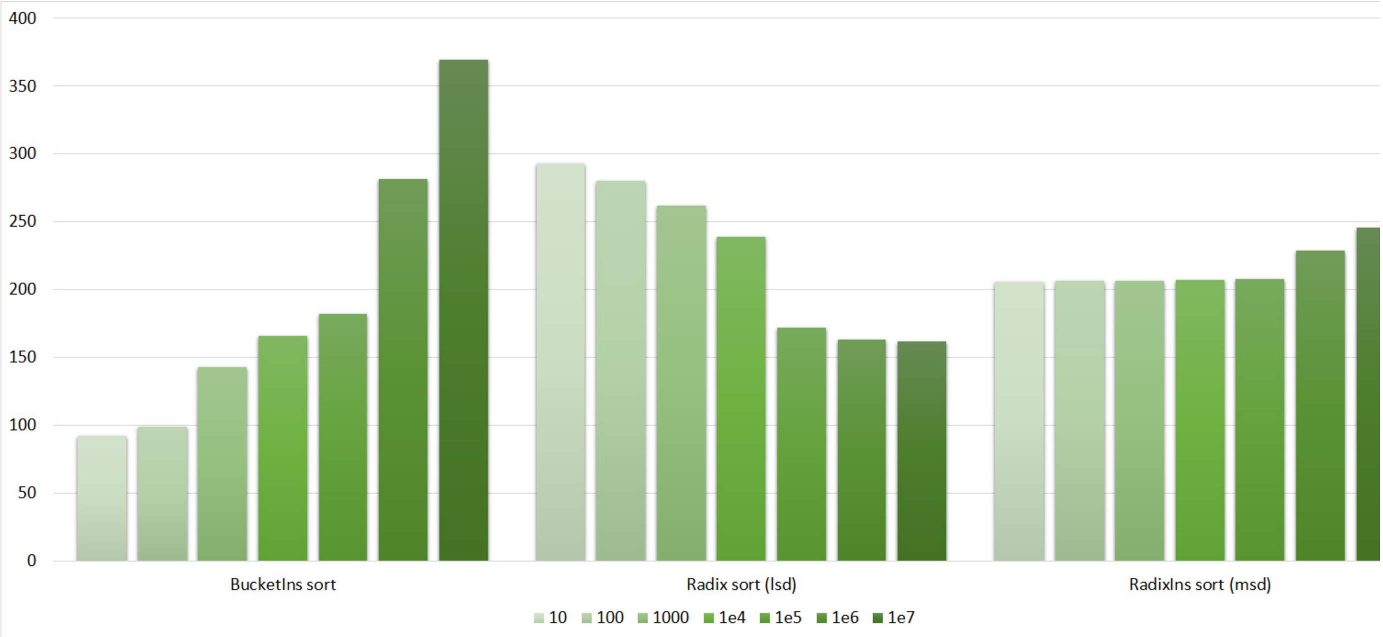
1e8	10	100	1000	1e4	1e5	1e6	1e7	1e8	Итог
Comb sort	5814,74	7333,21	8743,97	10200,24	11614,8	12991,74	14215,3	14826,87	10717,
BucketIns sort	906,79	952,2	1271,72	1648,38	1660,25	1705,08	2101,53	3461,87	1713,4
Heapsort	9131,83	9138,89	9153,11	9213,23	9552,47	10724,72	16268,85	30897,99	13010,
sort (stl)	1506,81	1505,65	1505,54	1507,3	1544,84	2049,97	5785,62	9649,37	3131,8
Quicksort	1106,74	1105,74	1104,82	1107,12	1129,15	1383,81	3139,06	8206,98	2285,4
QuickIns sort	918,4	917,33	916,99	918,68	943,98	1186,32	2773,92	7518,67	2011,7
Merge sort	4441,19	4470,06	4507,7	4553,45	4613,18	4818,8	6253,88	11367,86	5628,2
MergeIns sort	3534,43	3538,08	3546,58	3563,45	3612,43	3829,99	5505,61	10813,81	4743,0
Shellsort	3230,04	5691,05	8204,82	11368,99	15425,88	21122,98	30082,13	35759,3	16360,
Shellsort (Sedgwick)	3058,01	5119,71	7106,29	9265,85	11576,47	13908,83	16105,08	17427,69	10445,
Radix sort (lsd)	1595,4	1586,47	1589,35	1585,4	1584,67	1584,93	1589,98	1579,19	1586,9
RadixIns sort (msd)	1928,42	1927,74	1927,19	1927,32	1930,86	1954,96	2191,42	2743,86	2066,4
Timsort	898,87	1375,55	1895,99	2353,03	2881,55	3669,08	5549,25	10379,18	3625,3
Shellsort (my form. 1)	1914,5	4035,29	5948,05	7980,51	10172,53	12570,46	14886,23	16145,19	9206,6
Shellsort (my form. 2)	2264,83	4278,62	6178,63	8169,02	10300,9	12417,71	14360,11	15319,41	9161,1
Shellsort (my form. 3)	1938,07	4233,97	6181,46	8171,86	10325,88	12526,36	14595,88	15738,77	9214,0

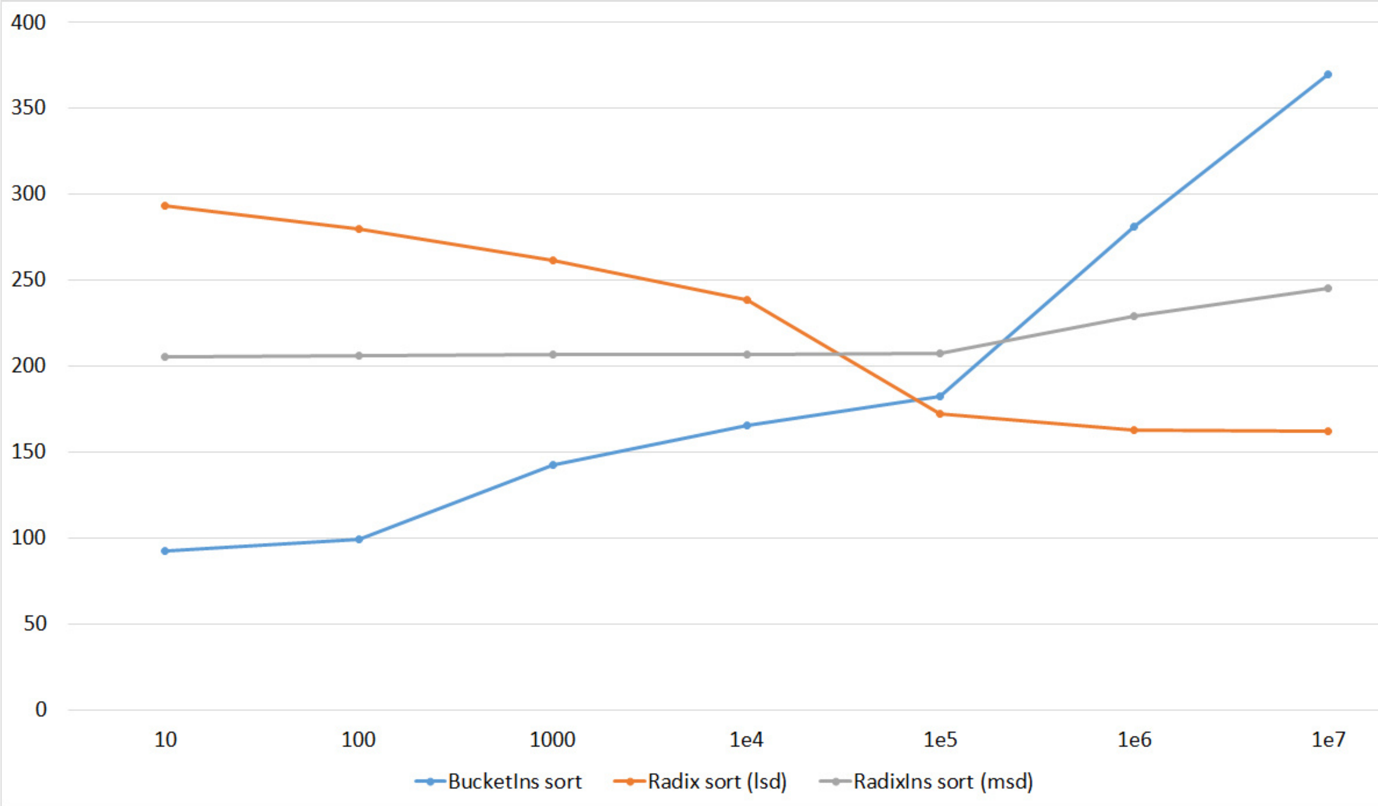
1e8	Итог
Radix sort (lsd)	1586,92
BucketIns sort	1713,48
QuickIns sort	2011,79
RadixIns sort (msd)	2066,47
Quicksort	2285,43
sort (stl)	3131,89
Timsort	3625,31
MergeIns sort	4743,05
Merge sort	5628,27
Shellsort (my form. 2)	9161,15
Shellsort (my form. 1)	9206,60
Shellsort (my form. 3)	9214,03
Shellsort (Sedgwick)	10445,99
Comb sort	10717,61
Heapsort	13010,14
Shellsort	16360,65

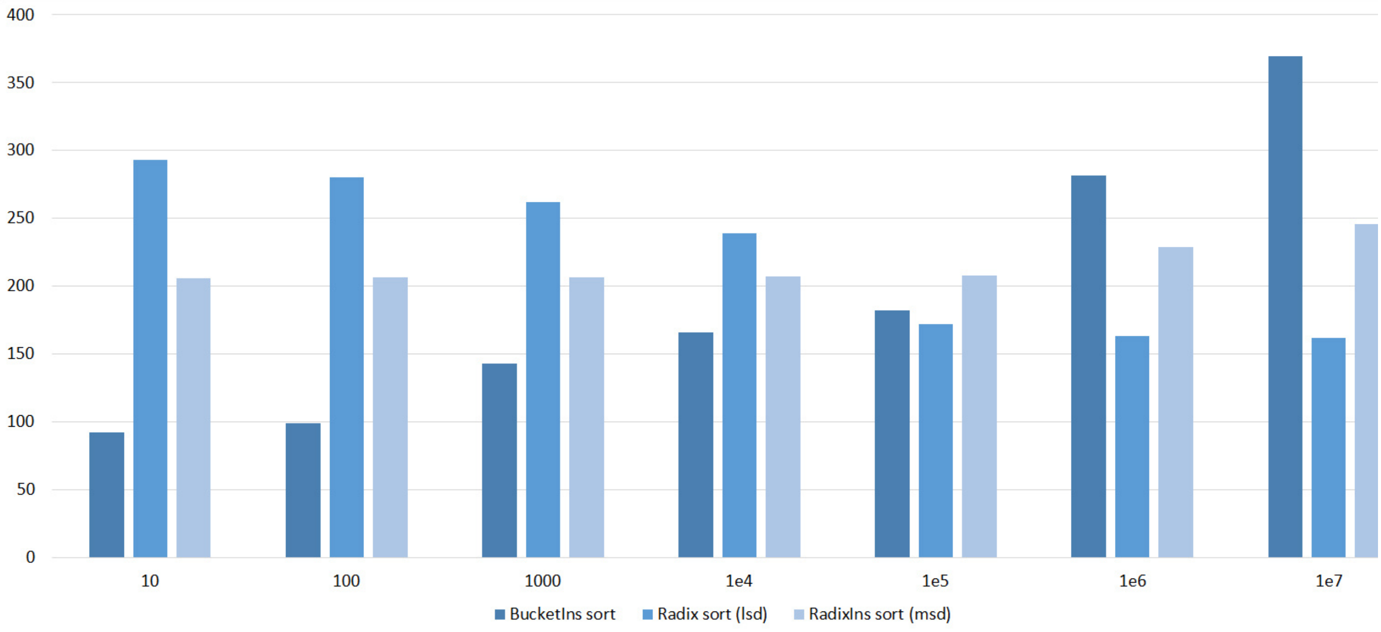
Здесь очень хорошо сработали быстрые сортировки. Это, скорее всего, объясняется удачным выбором опорного элемента. Все остальное также, как и в предыдущей группе.

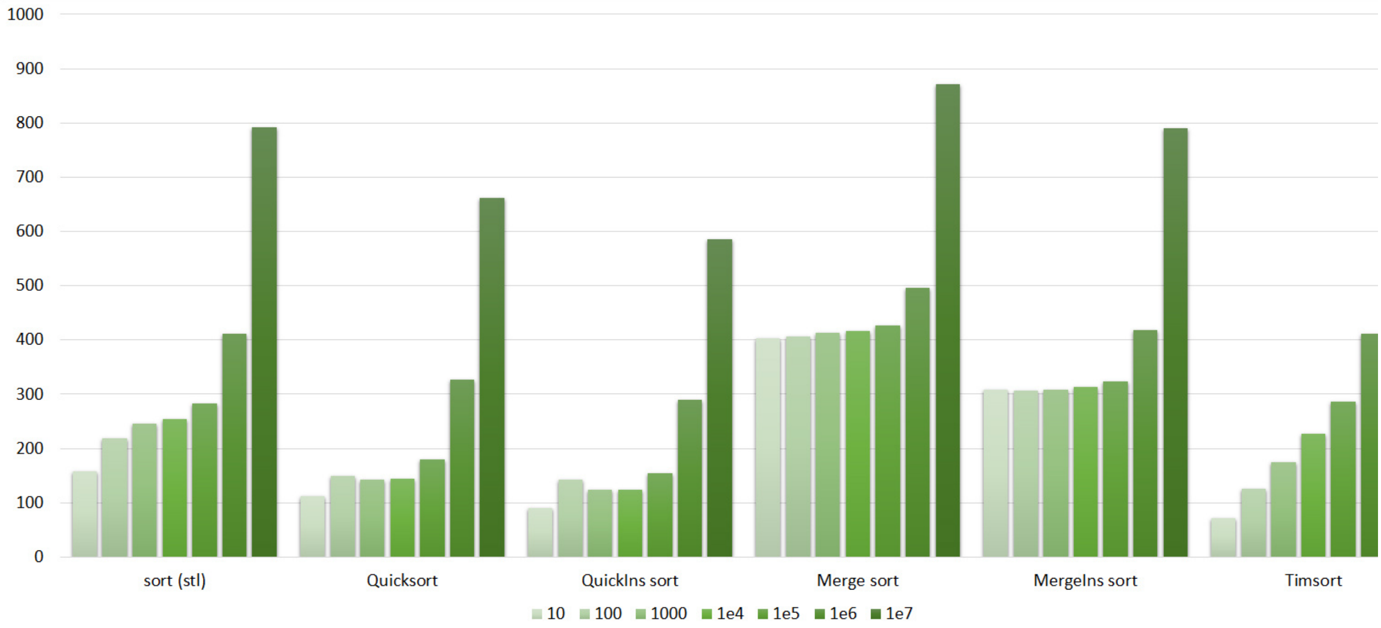
Изменения в перестановке

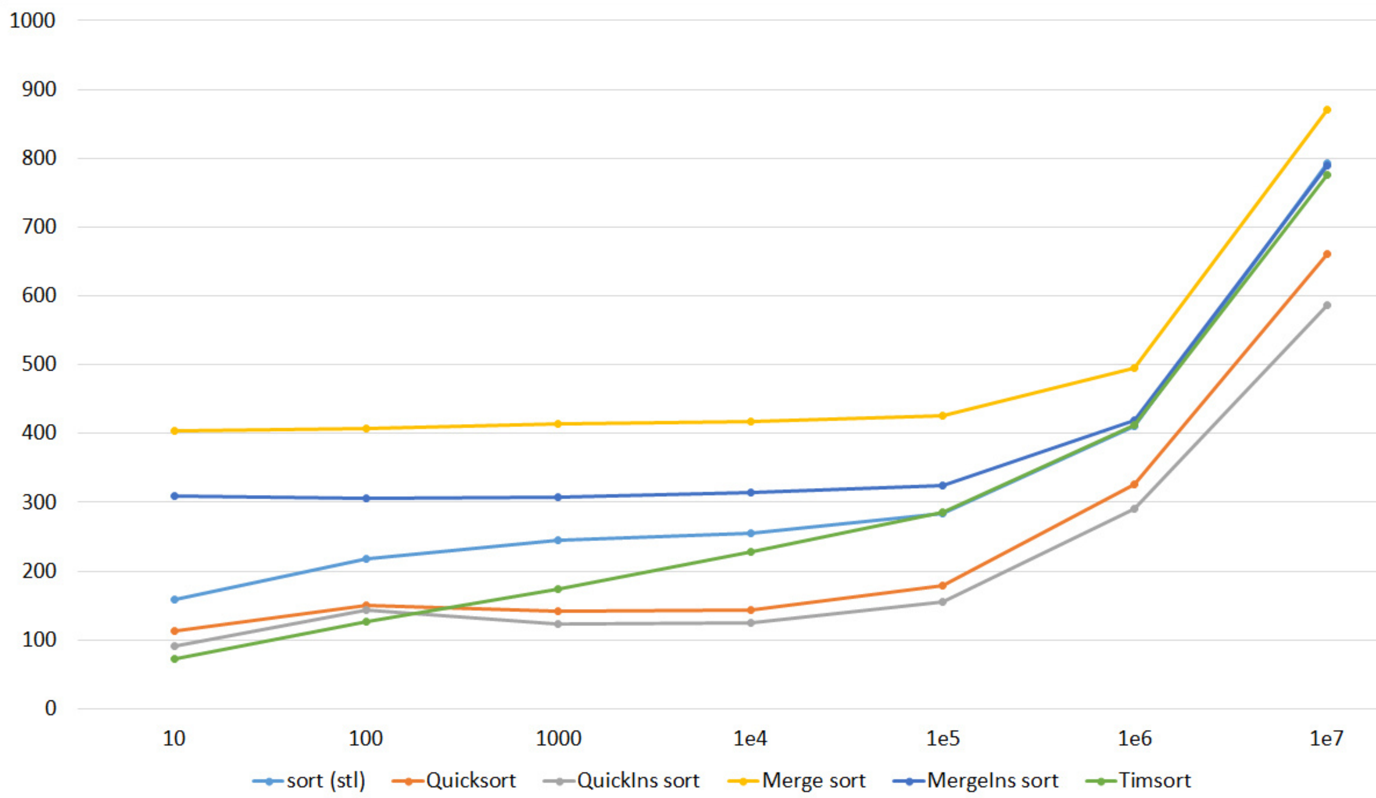
Таблицы, 1e7 элементов

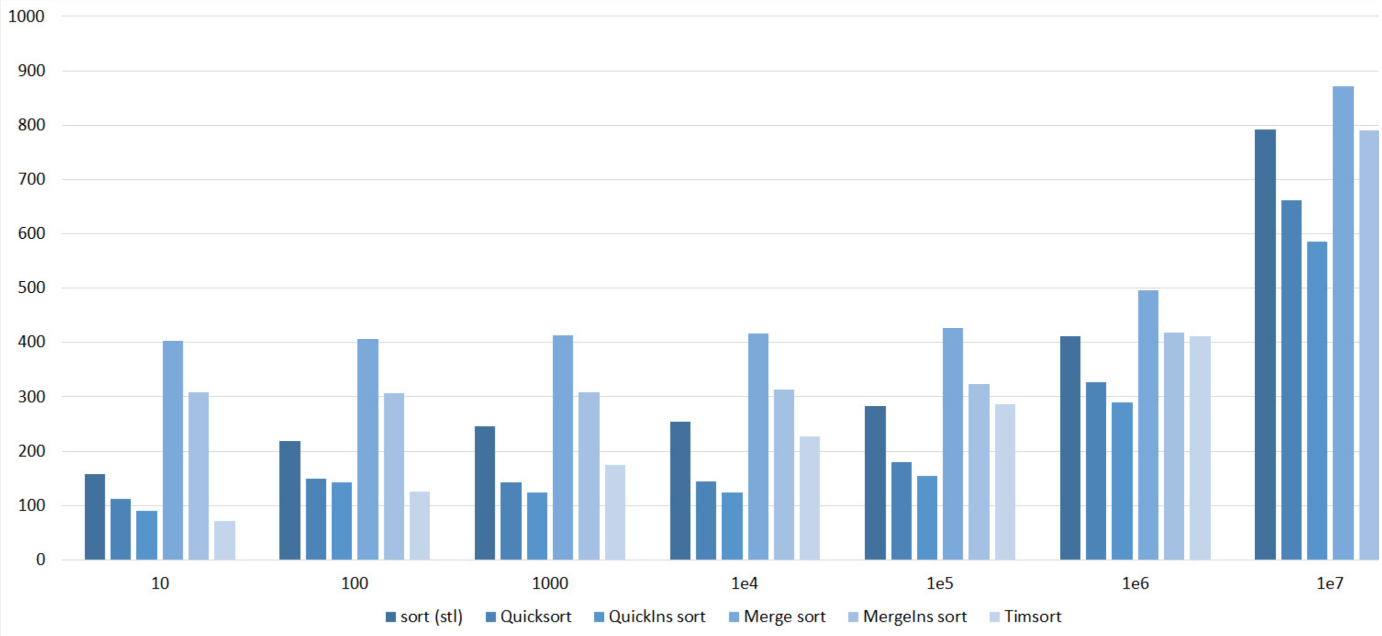

400
350
300
250
200
150
100
50
0
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)
10 100 1000 1e4 1e5 1e6 1e7

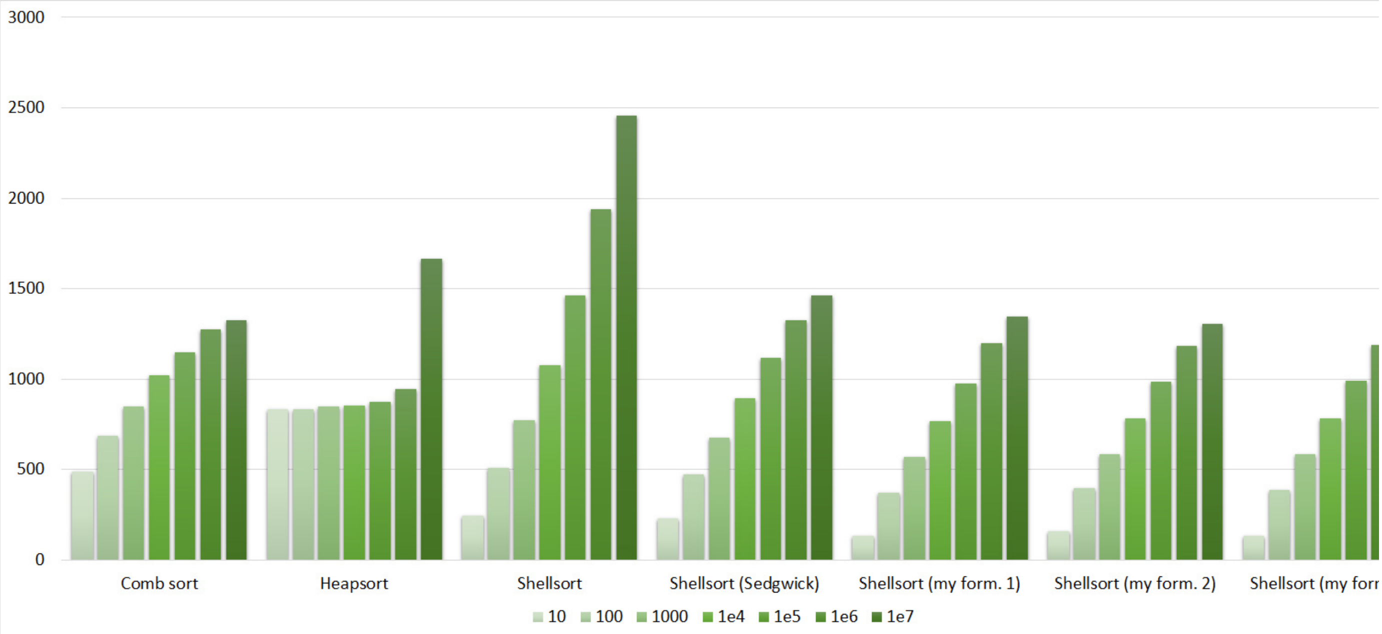

400
350
300
250
200
150
100
50
0
10 100 1000 1e4 1e5 1e6 1e7
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

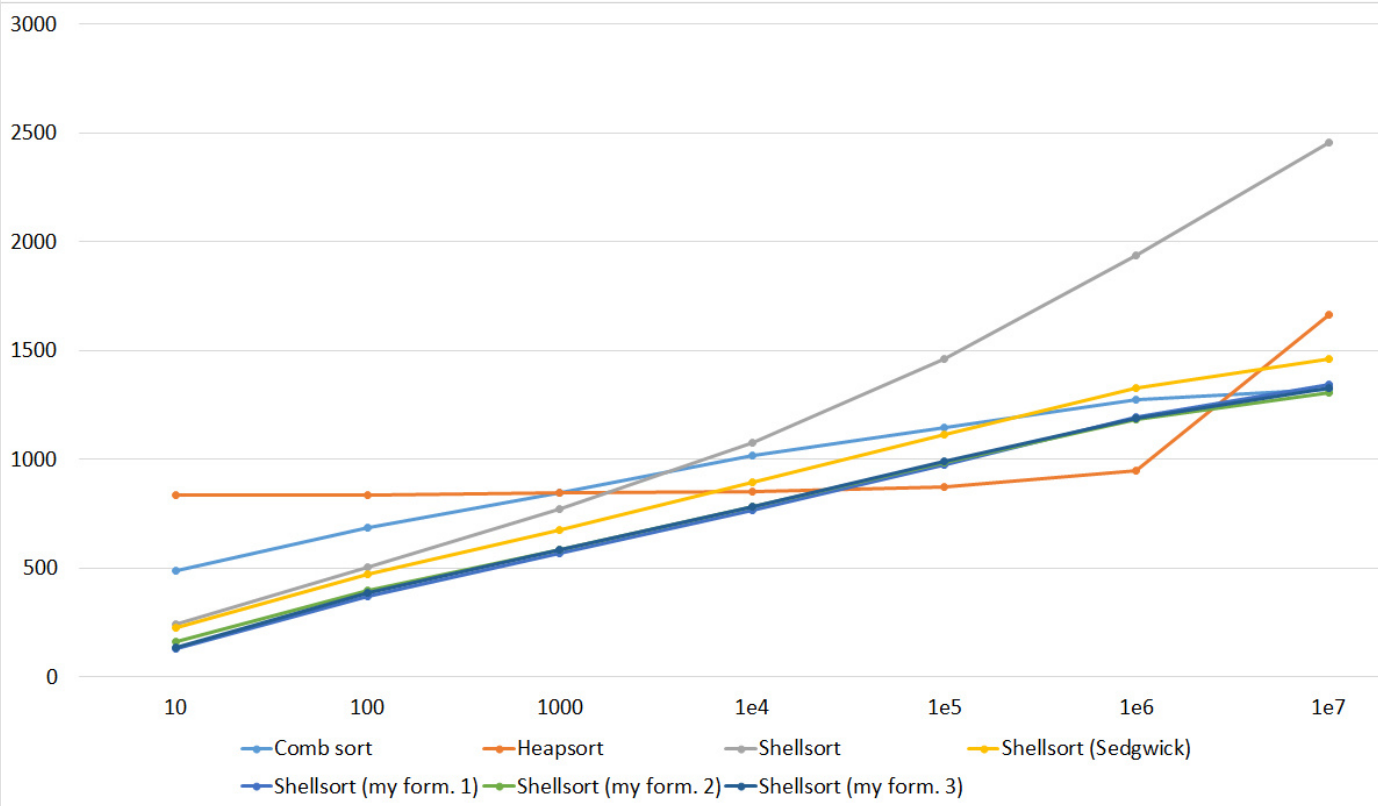

400
350
300
250
200
150
100
50
0
10
100
1000
1e4
1e5
1e6
1e7
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

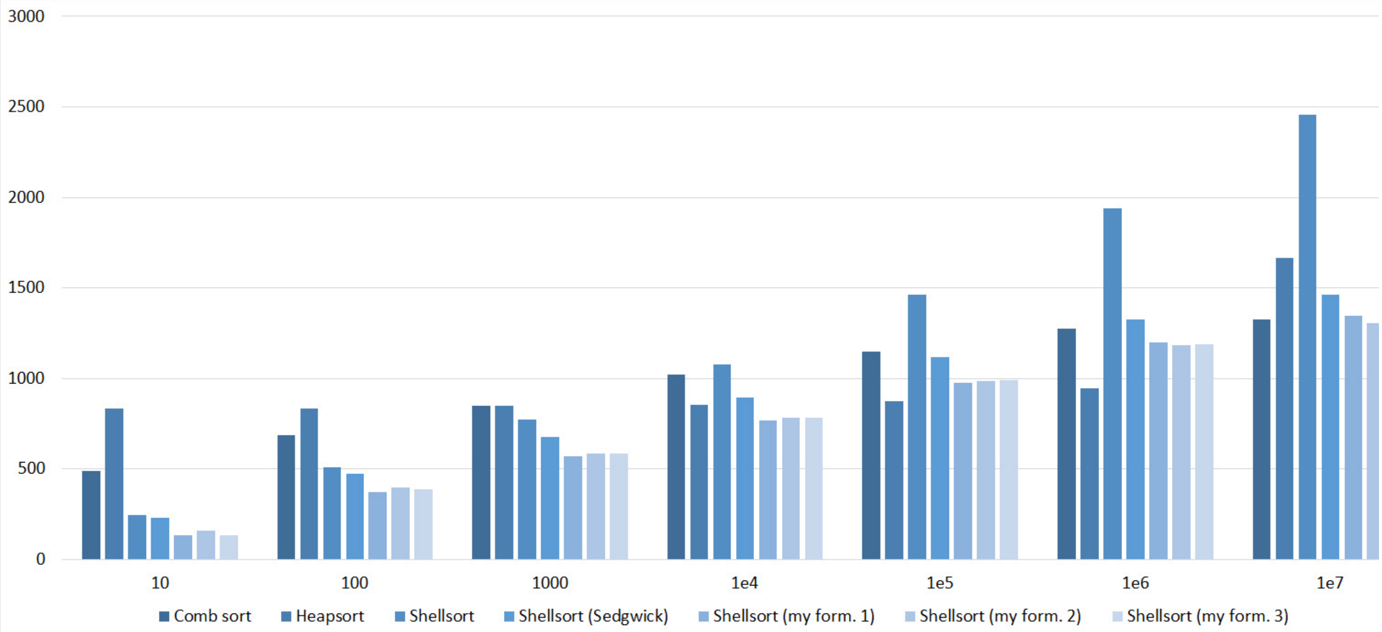

1000
900
800
700
600
500
400
300
200
100
0
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort
10
100
1000
1e4
1e5
1e6
1e7


1000
900
800
700
600
500
400
300
200
100
0
10
100
1000
1e4
1e5
1e6
1e7
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


1000
900
800
700
600
500
400
300
200
100
0
10
100
1000
1e4
1e5
1e6
1e7
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


3000
2500
2000
1500
1000
500
0
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my forr
10
100
1000
1e4
1e5
1e6
1e7

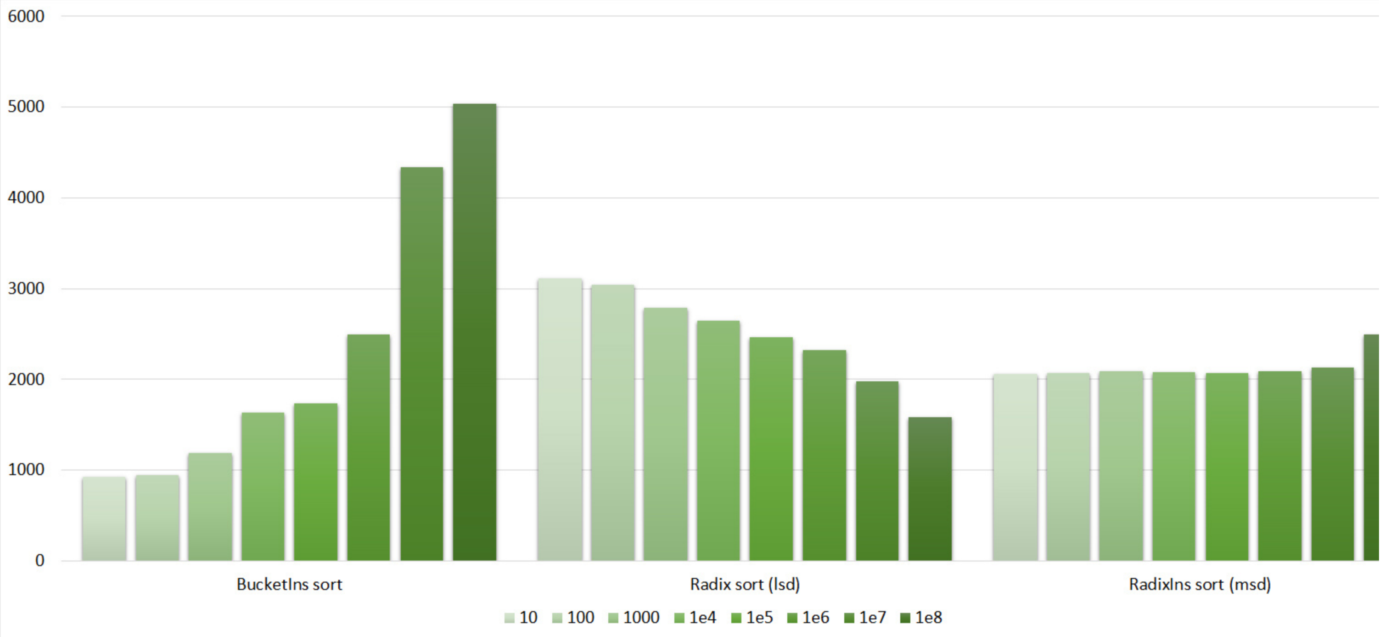

3000
2500
2000
1500
1000
500
0
10
100
1000
1e4
1e5
1e6
1e7
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my form. 3)

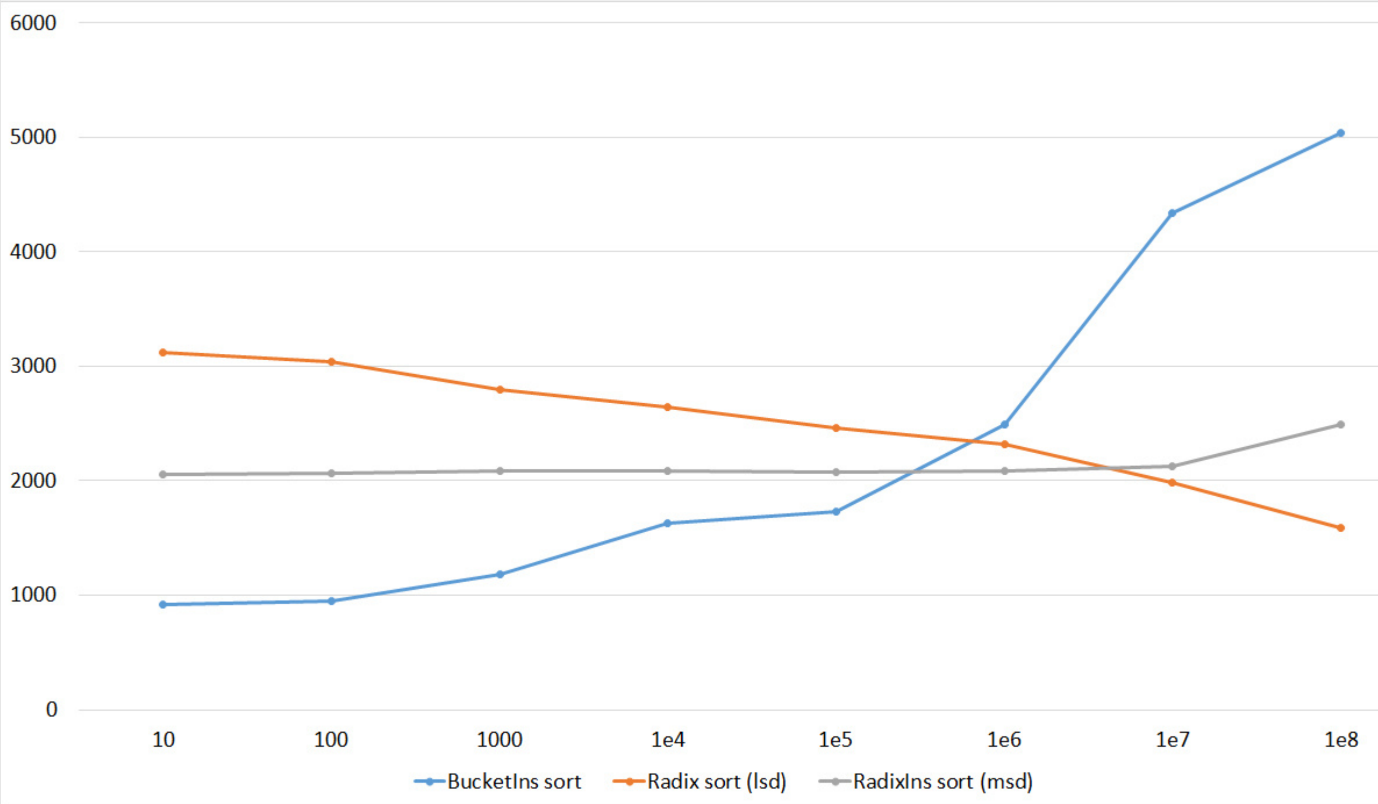


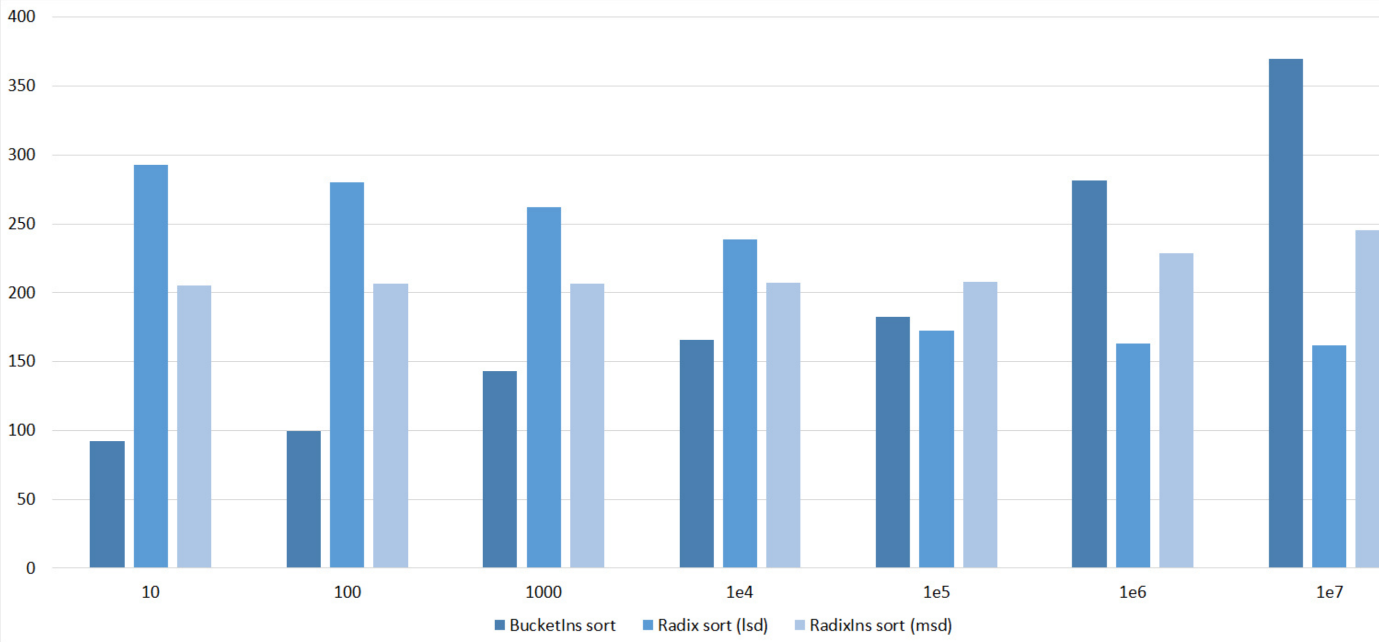
1e7	10	100	1000	1e4	1e5	1e6	1e7	Ито
Comb sort	489,83	688,88	848,40	1020,25	1146,35	1276,63	1325,43	970,
BucketIns sort	92,38	99,28	142,73	165,60	182,35	281,40	369,55	190,
Heapsort	834,20	836,03	849,25	852,83	873,25	947,60	1666,40	979,
sort (stl)	158,18	218,08	245,20	254,33	283,40	410,73	792,15	337,
Quicksort	112,75	149,60	142,05	144,20	179,30	326,63	661,28	245,
QuickIns sort	90,53	143,00	123,43	124,33	154,50	289,70	586,00	215,
Merge sort	403,30	407,10	413,48	416,80	426,63	495,25	871,05	490,
MergeIns sort	308,55	305,93	307,85	313,40	324,18	418,23	789,45	395,
Shellsort	243,15	507,38	774,13	1076,80	1459,98	1936,90	2455,15	1207,
Shellsort (Sedgwick)	228,03	473,60	676,73	892,80	1115,75	1327,48	1463,50	882,
Radix sort (lsd)	293,05	280,13	261,88	238,63	172,23	162,98	161,90	224,
RadixIns sort (msd)	205,50	206,30	206,78	207,00	207,58	228,75	245,48	215,
Timsort	72,05	125,78	174,08	227,30	286,13	411,98	776,33	296,
Shellsort (my form. 1)	131,20	373,45	569,35	765,38	973,13	1196,93	1344,40	764,
Shellsort (my form. 2)	159,85	395,35	585,03	780,58	984,70	1182,70	1305,45	770,
Shellsort (my form. 3)	135,68	389,13	585,85	782,35	989,15	1190,85	1328,55	771,

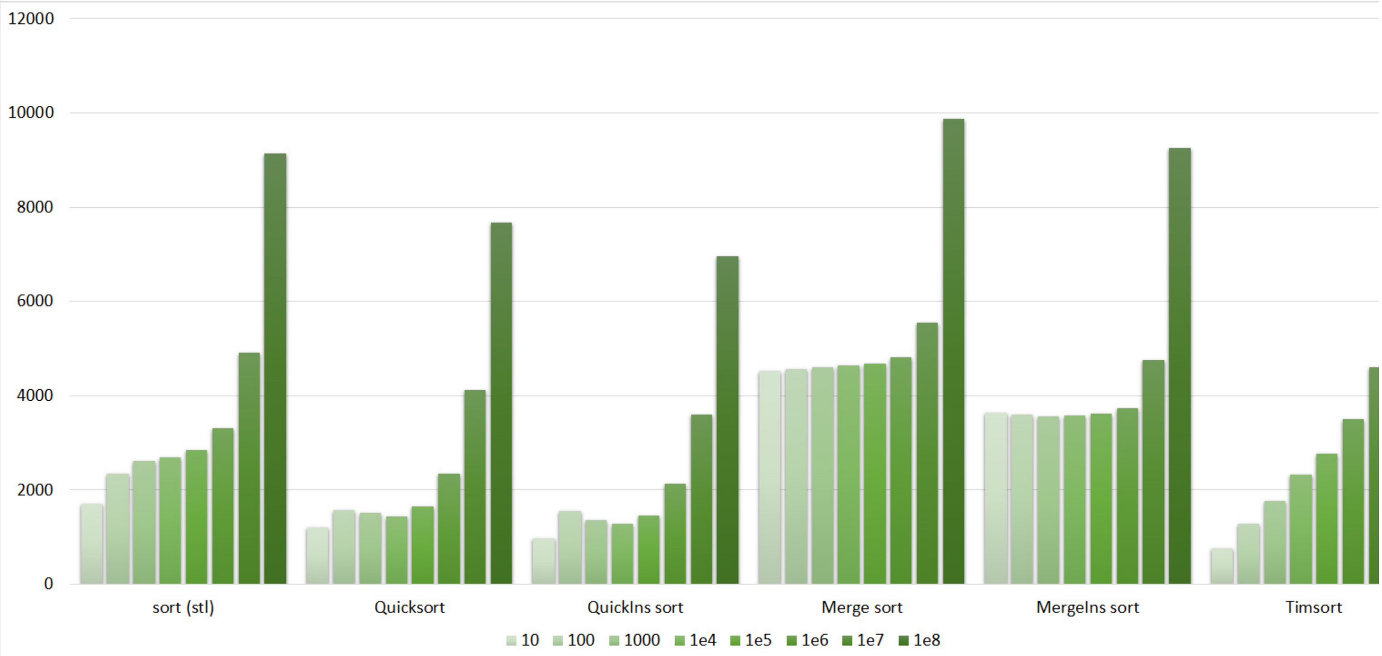
1e7	Итог
BucketIns sort	190,47
RadixIns sort (msd)	215,34
QuickIns sort	215,93
Radix sort (lsd)	224,40
Quicksort	245,11
Timsort	296,23
sort (stl)	337,44
MergeIns sort	395,37
Merge sort	490,51
Shellsort (my form. 1)	764,83
Shellsort (my form. 2)	770,52
Shellsort (my form. 3)	771,65
Shellsort (Sedgwick)	882,55
Comb sort	970,82
Heapsort	979,94
Shellsort	1207,64

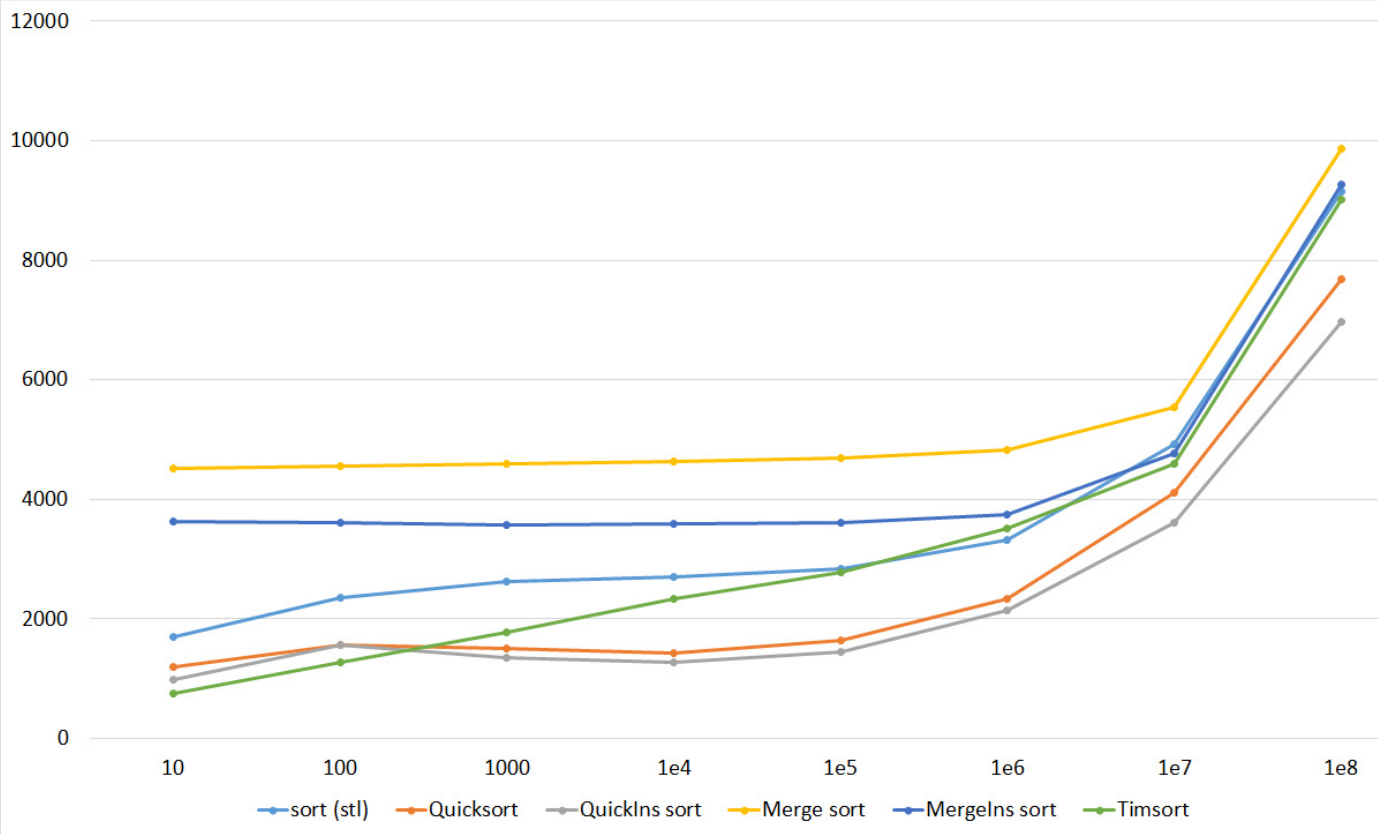
Таблицы, 1e8 элементов

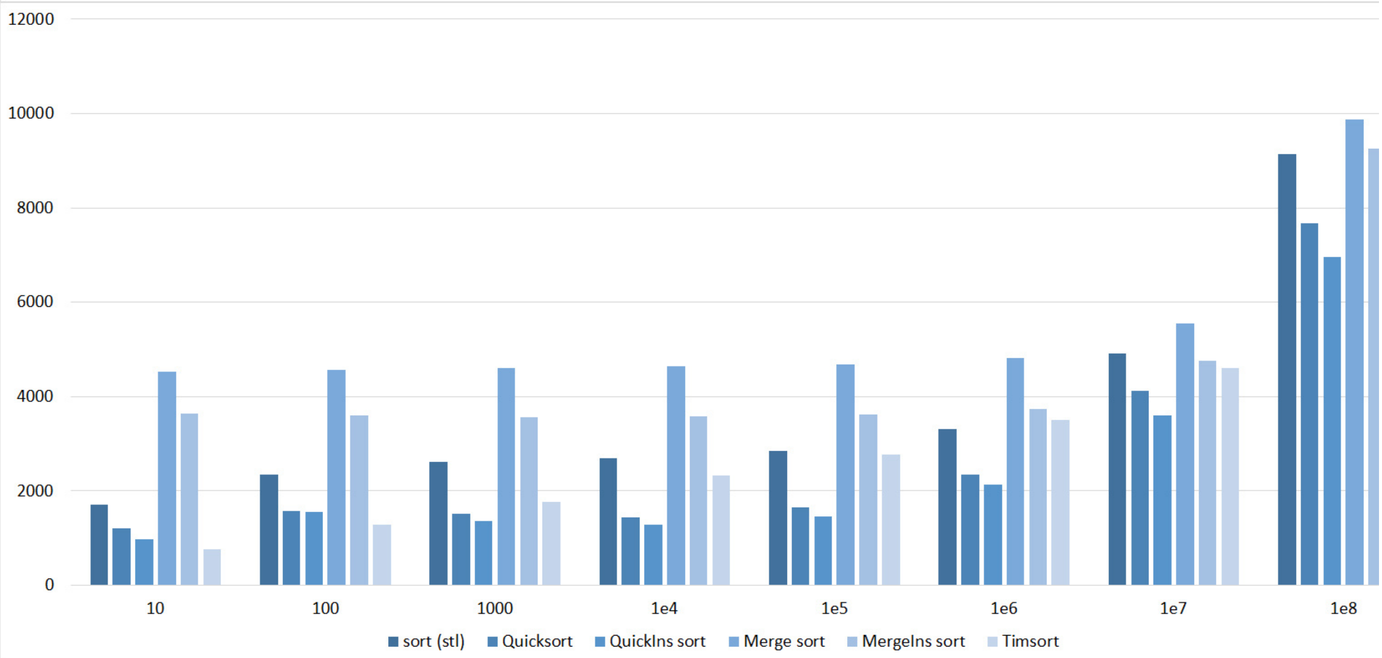


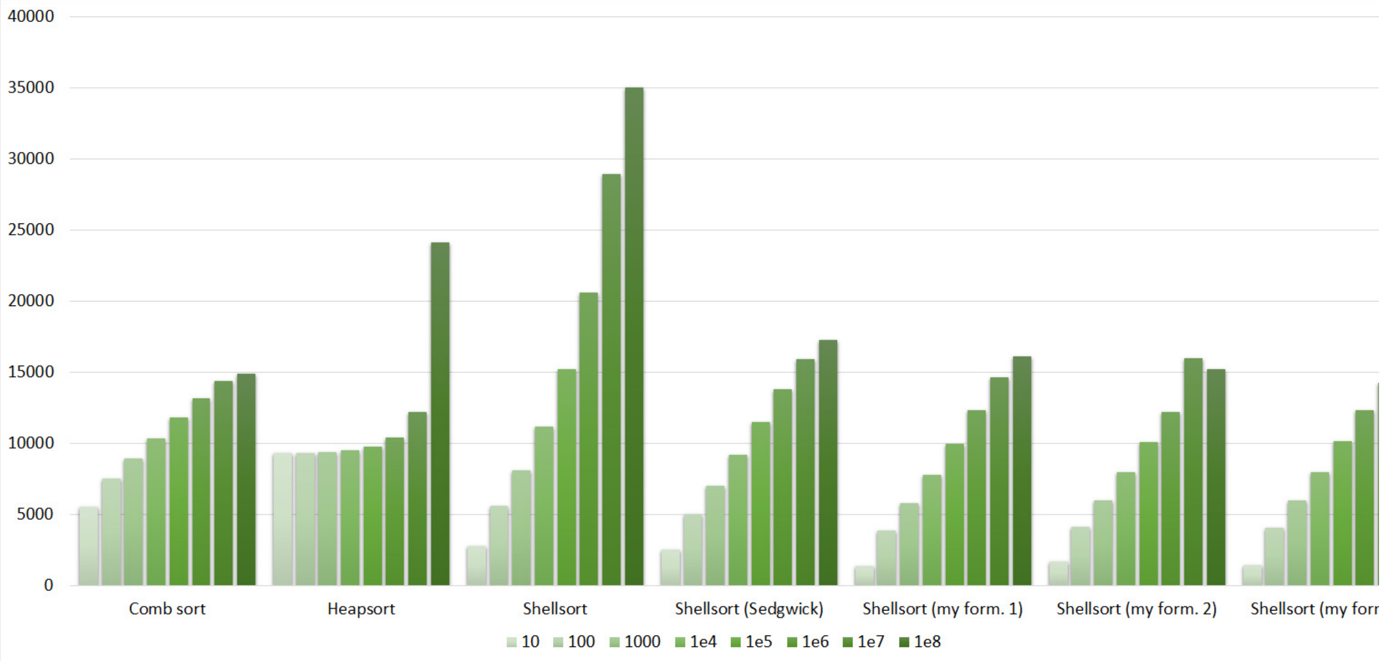


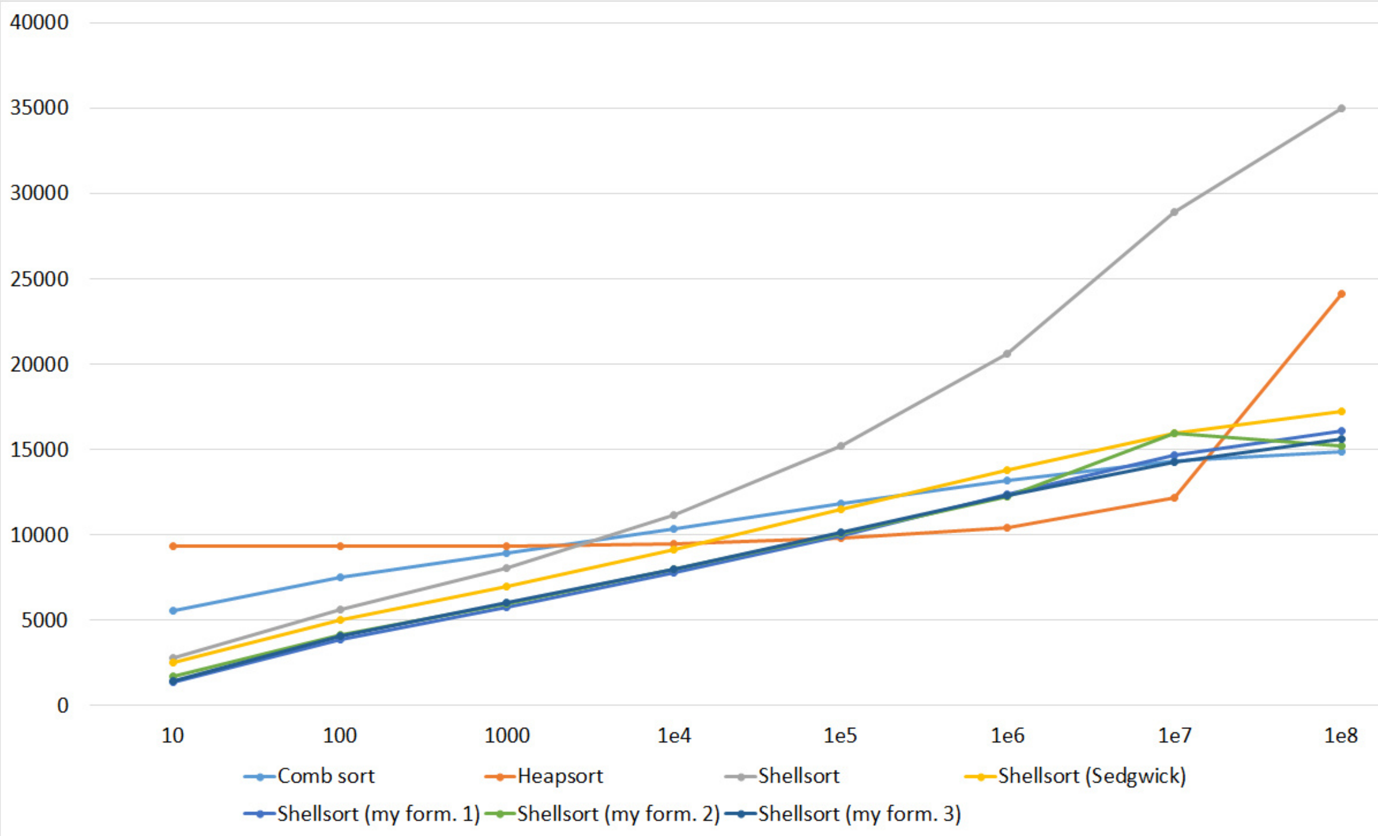

400
350
300
250
200
150
100
50
0
10
100
1000
1e4
1e5
1e6
1e7
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

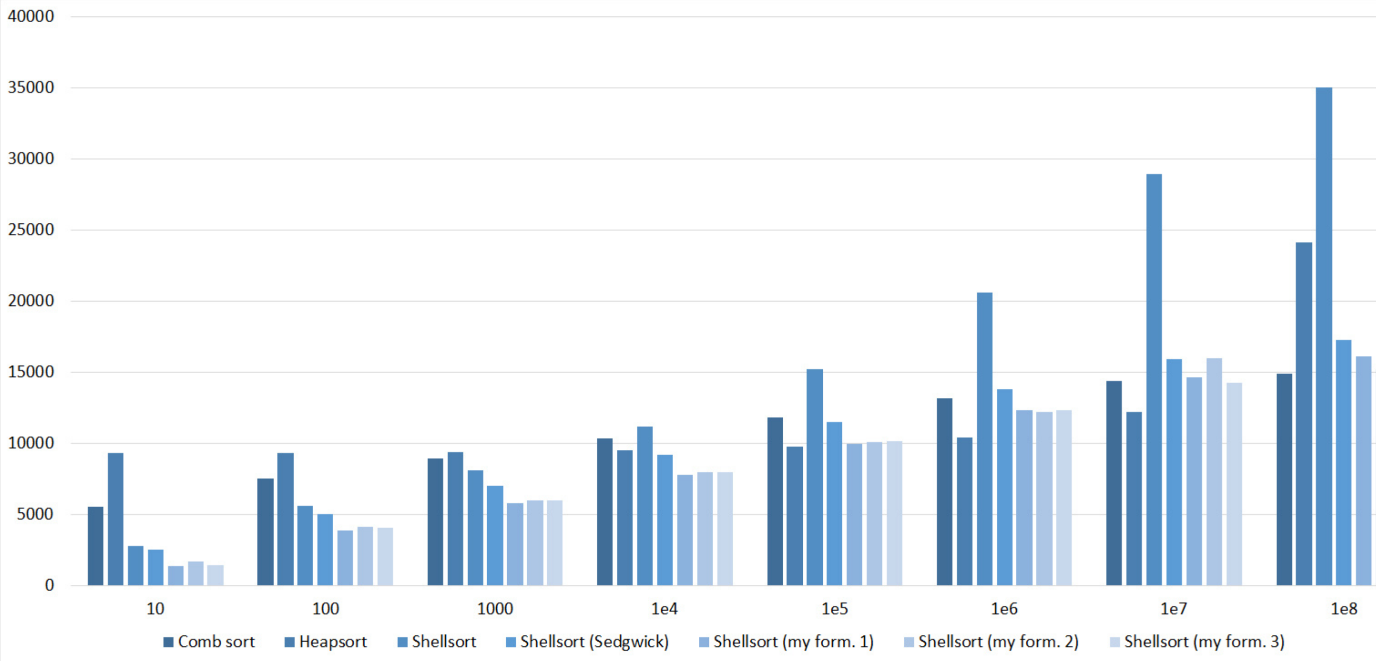

12000
10000
8000
6000
4000
2000
0
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort
10
100
1000
1e4
1e5
1e6
1e7
1e8


12000
10000
8000
6000
4000
2000
0
10
100
1000
1e4
1e5
1e6
1e7
1e8
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


12000
10000
8000
6000
4000
2000
0
10
100
1000
1e4
1e5
1e6
1e7
1e8
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


40000
35000
30000
25000
20000
15000
10000
5000
0
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
10 100 1000 1e4 1e5 1e6 1e7 1e8


40000
35000
30000
25000
20000
15000
10000
5000
0
10 100 1000 1e4 1e5 1e6 1e7 1e8
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my form. 3)



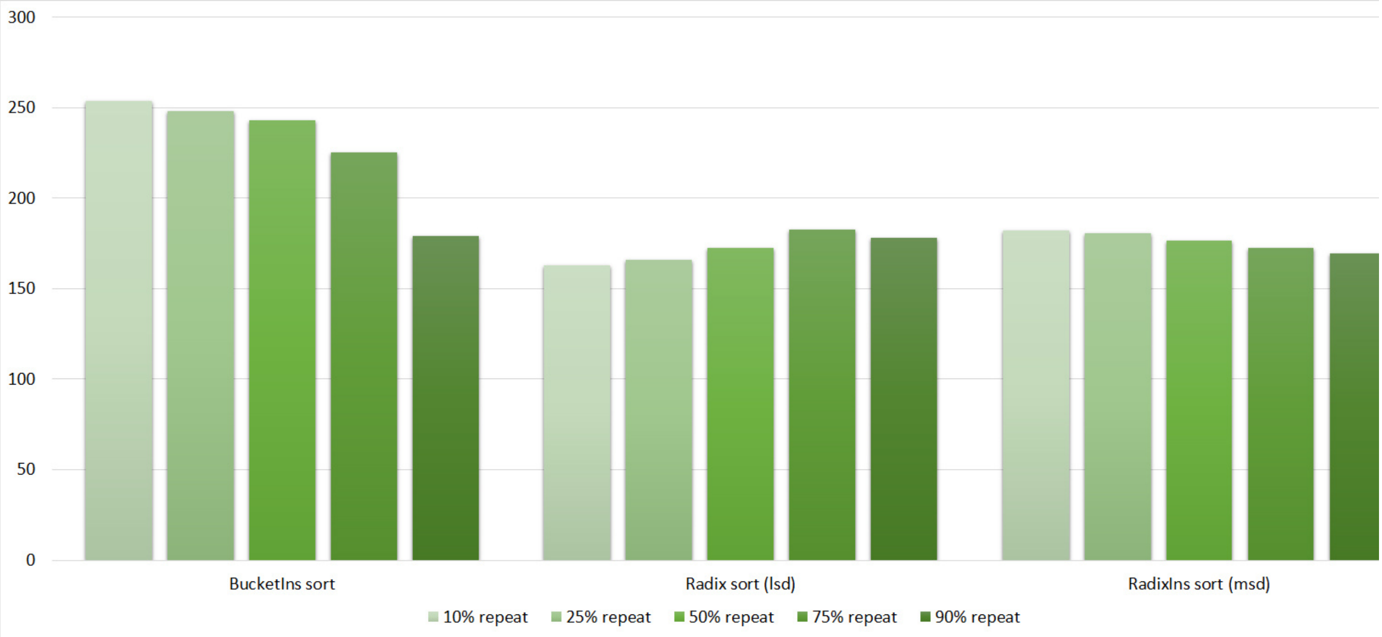
1e8	10	100	1000	1e4	1e5	1e6	1e7	1e8	Ито
Comb sort	5578,00	7520,48	8946,50	10386,50	11857,53	13166,65	14367,68	14894,88	1083
BucketIns sort	921,38	950,20	1186,18	1633,53	1732,08	2496,13	4336,45	5038,88	2286
Heapsort	9319,28	9328,78	9377,08	9514,10	9798,13	10397,43	12205,23	24167,20	1176
sort (stl)	1705,48	2348,75	2616,30	2700,63	2838,58	3313,50	4916,50	9147,53	3698
Quicksort	1198,95	1568,65	1510,90	1431,23	1648,23	2340,33	4112,15	7679,28	2686
QuickIns sort	981,13	1562,15	1354,03	1273,88	1452,48	2142,08	3609,53	6964,23	2417
Merge sort	4519,95	4556,83	4596,38	4637,15	4689,25	4817,00	5541,03	9871,78	5403
MergeIns sort	3630,58	3605,50	3570,05	3587,43	3618,30	3739,38	4767,33	9260,48	4472
Shellsort	2804,30	5639,75	8094,35	11187,35	15254,58	20614,25	28955,33	35024,08	1594
Shellsort (Sedgwick)	2501,40	5022,85	6997,63	9178,90	11495,83	13814,10	15949,75	17269,35	1027
Radix sort (lsd)	3117,90	3037,53	2793,63	2649,15	2463,10	2324,18	1982,58	1587,45	2494
RadixIns sort (msd)	2056,28	2067,95	2088,68	2084,23	2072,08	2087,98	2131,80	2493,08	2135
Timsort	760,58	1281,70	1770,98	2335,28	2770,38	3504,55	4602,68	9016,13	3255
Shellsort (my form. 1)	1399,18	3898,55	5786,43	7816,23	9976,90	12355,08	14668,05	16107,70	9001
Shellsort (my form. 2)	1701,88	4124,28	5991,68	7982,05	10100,88	12228,53	15997,00	15252,48	9172
Shellsort (my form. 3)	1451,00	4070,58	6012,43	8011,18	10145,98	12323,25	14288,73	15650,78	8994

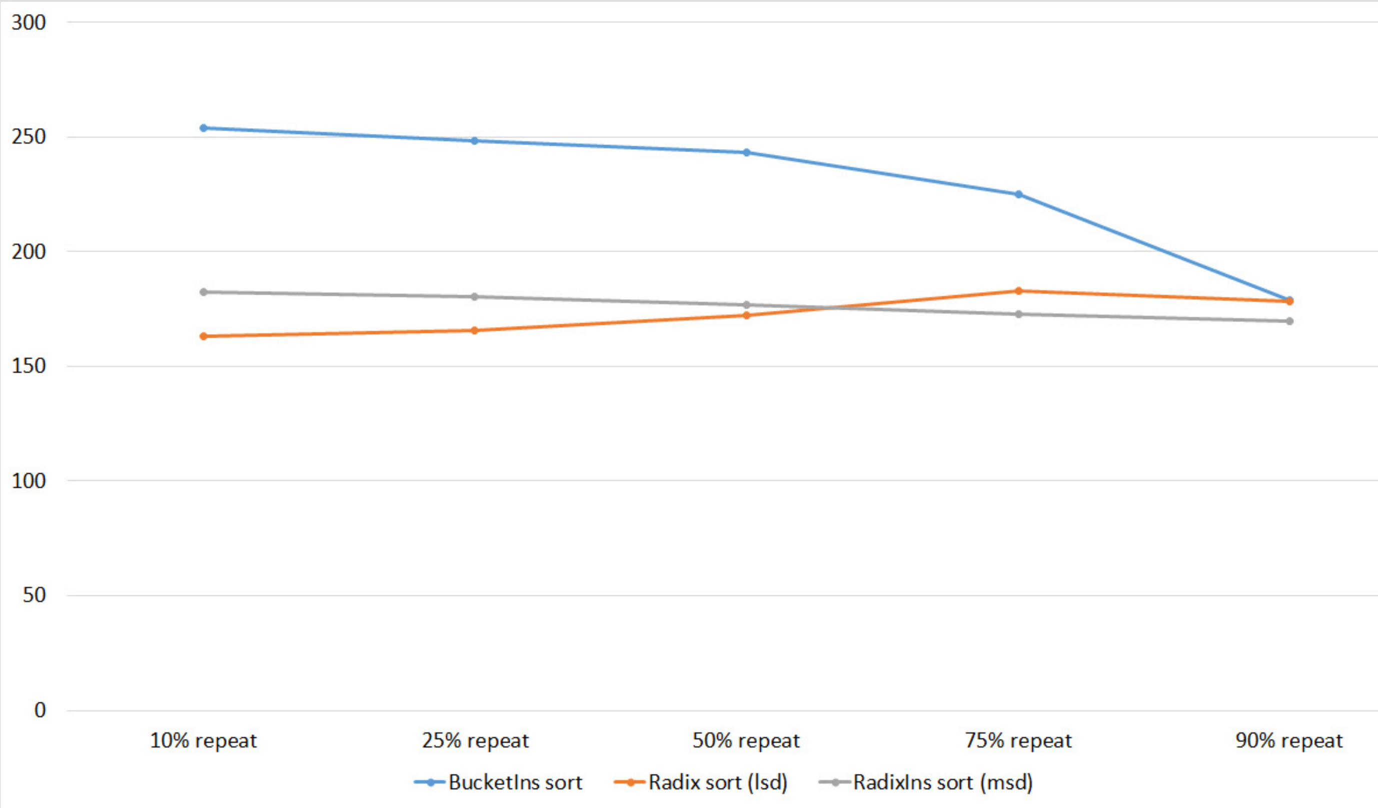
1e8	Итог
RadixIns sort (msd)	2135,26
BucketIns sort	2286,85
QuickIns sort	2417,43
Radix sort (lsd)	2494,44
Quicksort	2686,21
Timsort	3255,28
sort (stl)	3698,41
MergeIns sort	4472,38
Merge sort	5403,67
Shellsort (my form. 3)	8994,24
Shellsort (my form. 1)	9001,01
Shellsort (my form. 2)	9172,34
Shellsort (Sedgwick)	10278,73
Comb sort	10839,78
Heapsort	11763,40
Shellsort	15946,75

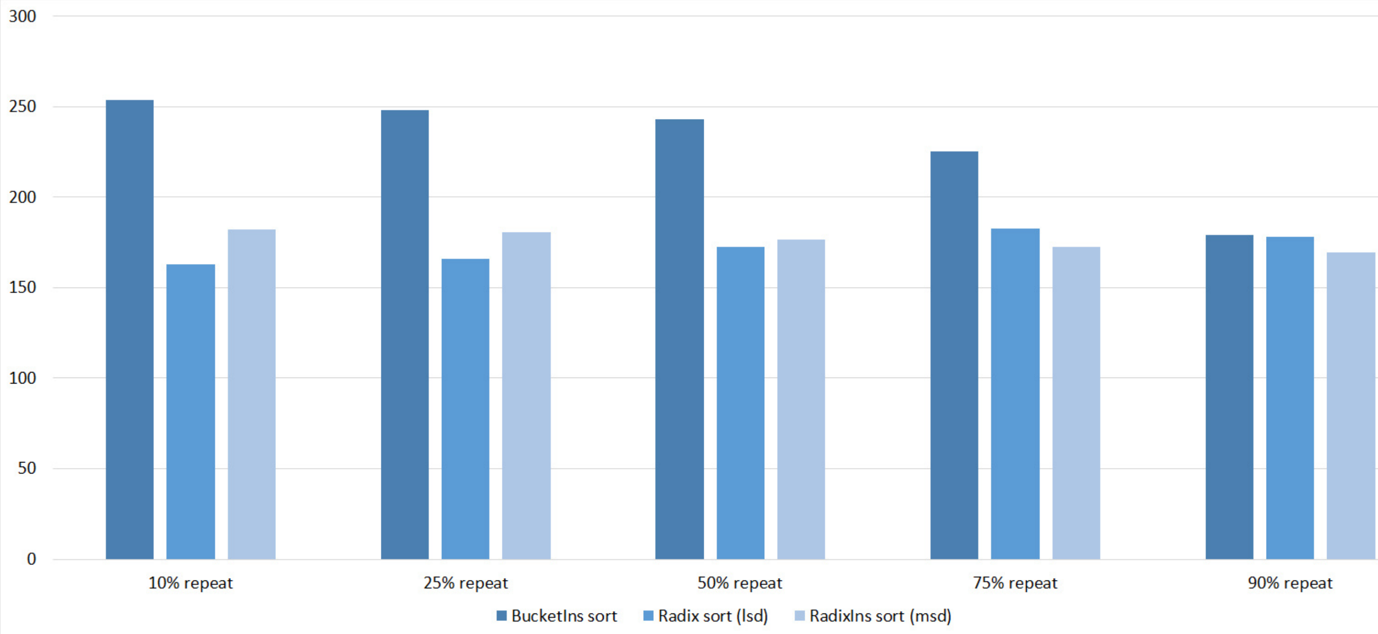
Мне удалось достичь желаемой цели — поразрядная сортировка упала даже ниже адаптированной быстрой. Блочная сортировка оказалас лучше остальных. Еще почему-то timsort обогнал встроенную сортировку C++, хотя в предыдущей группе был ниже.

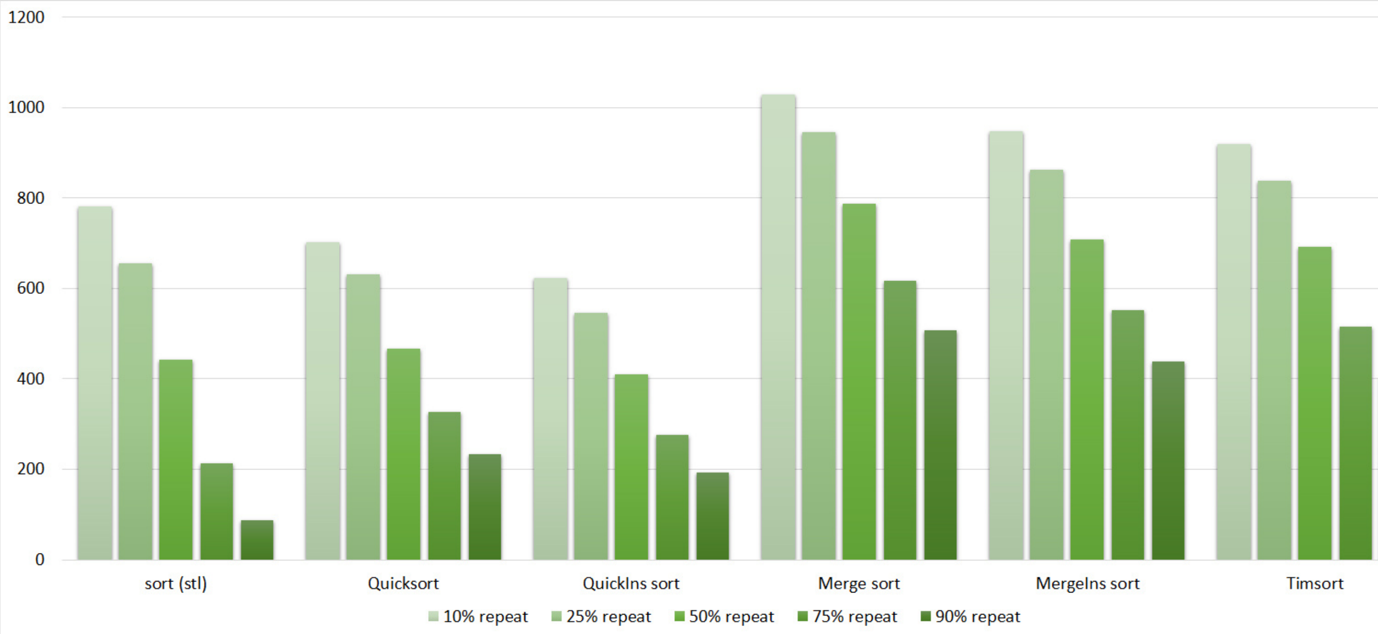
Повторы

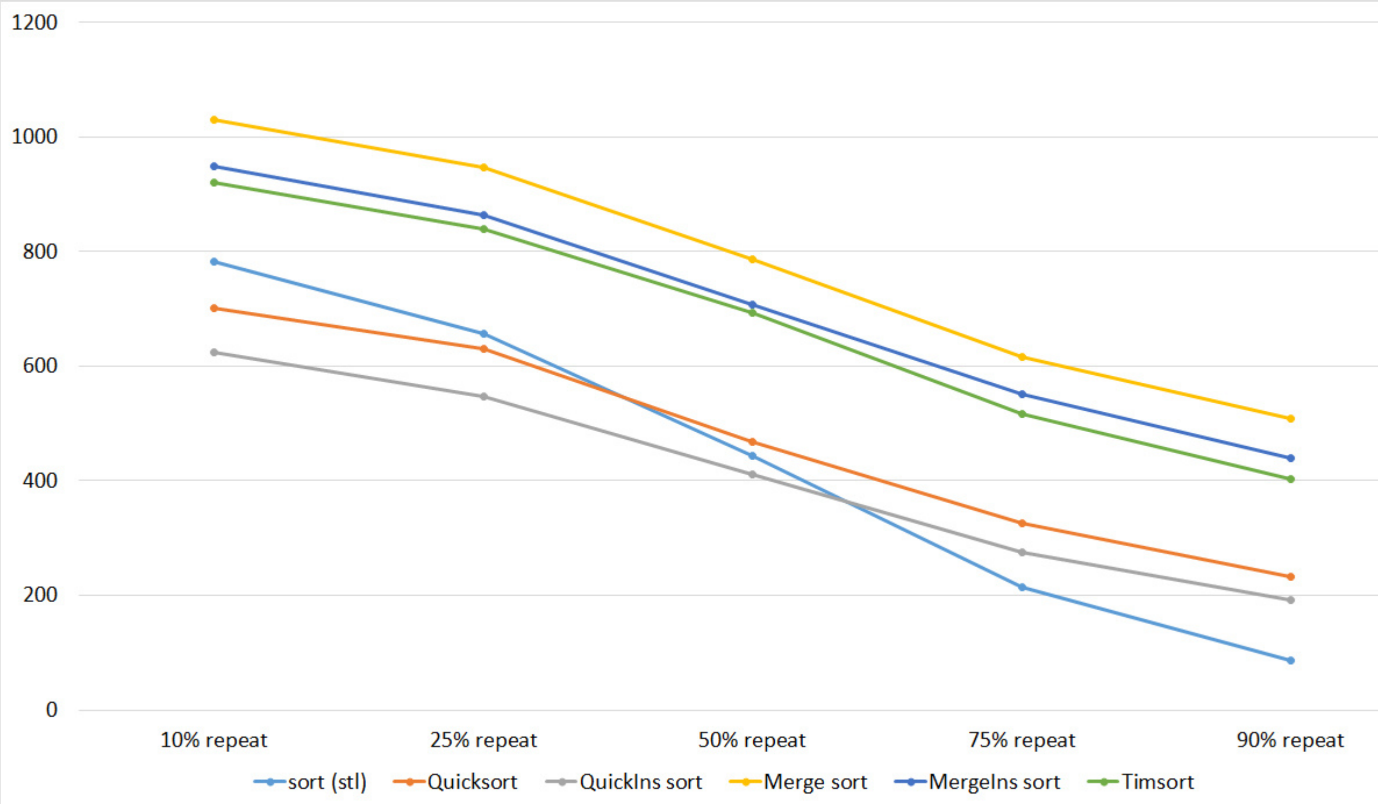
Таблицы, 1e7 элементов

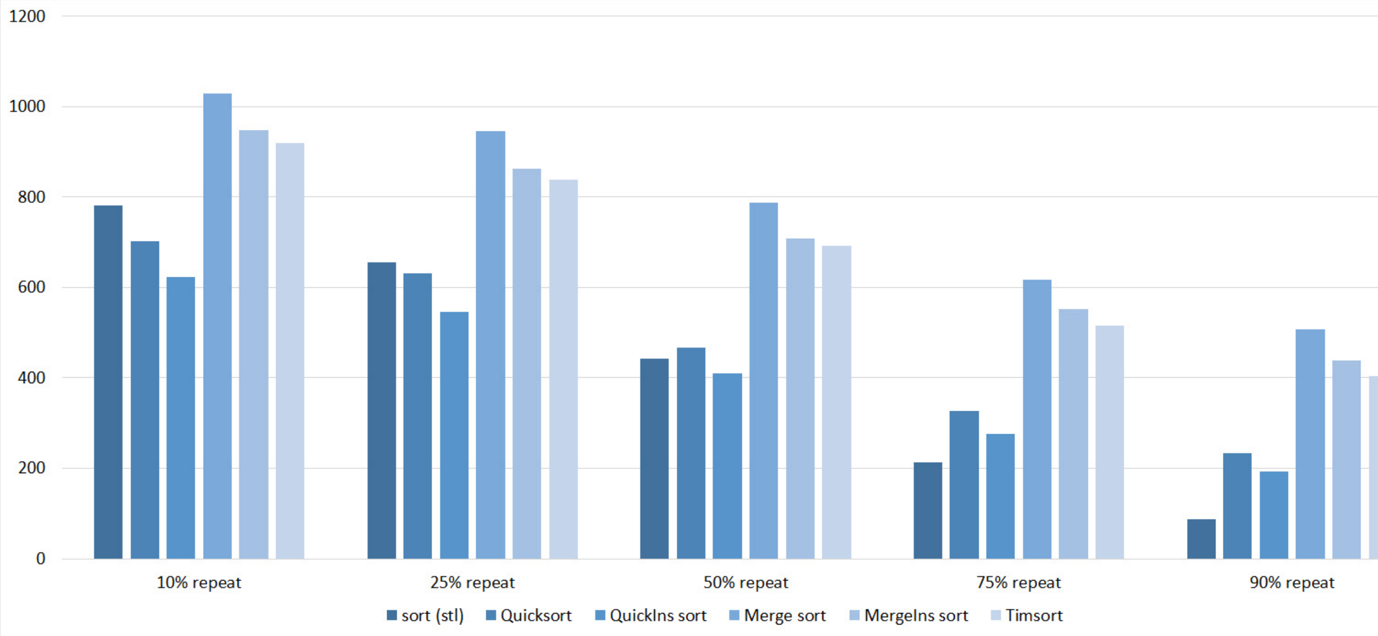


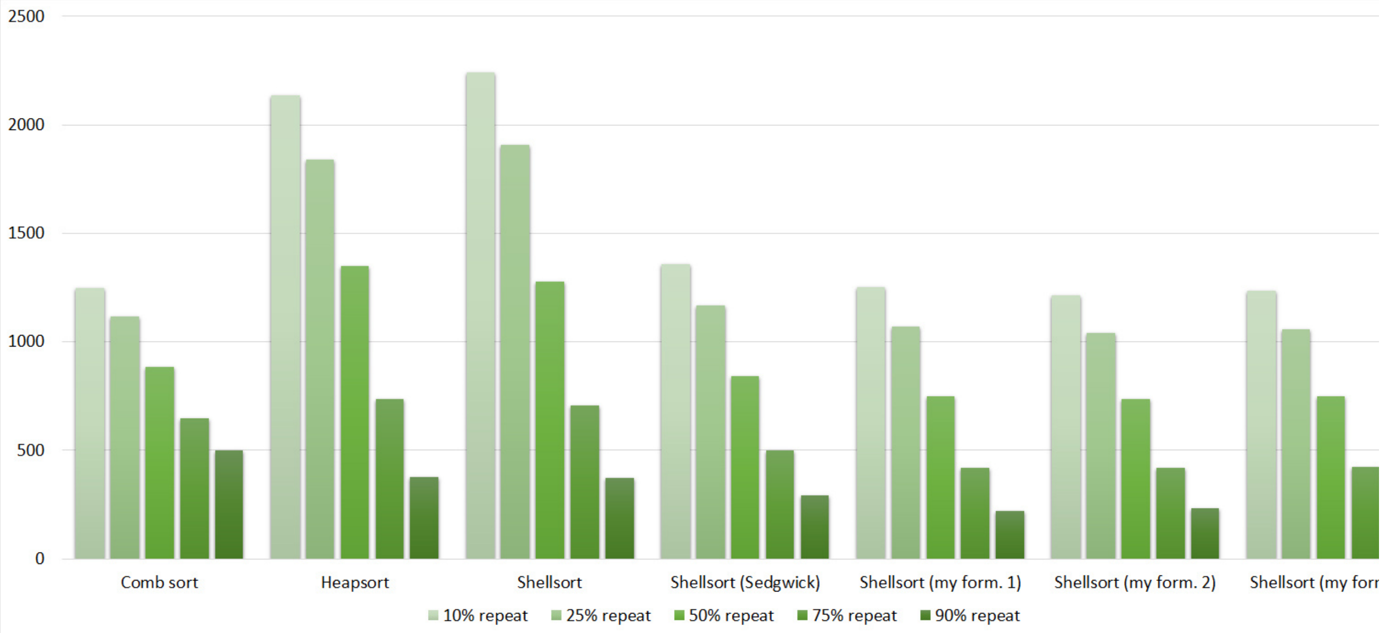


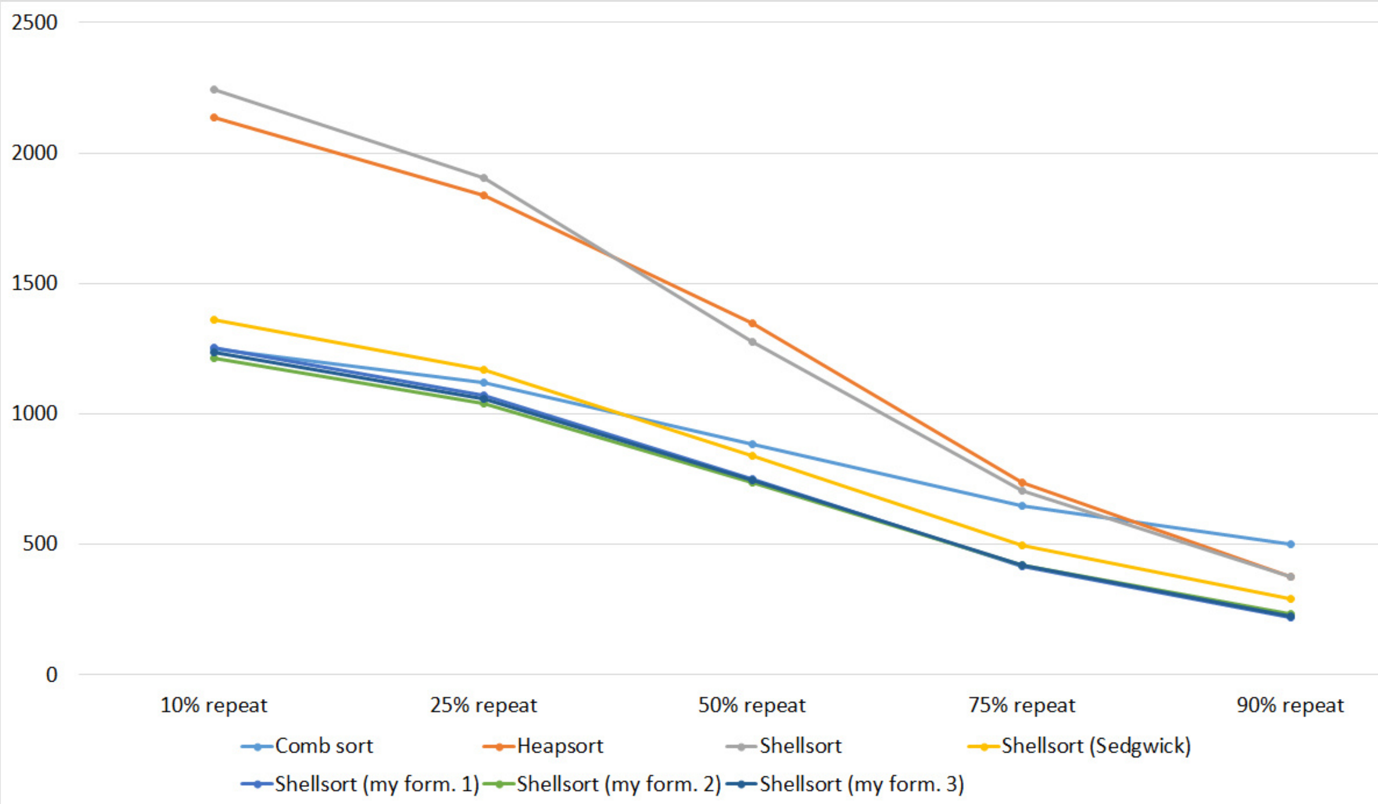

300
250
200
150
100
50
0
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

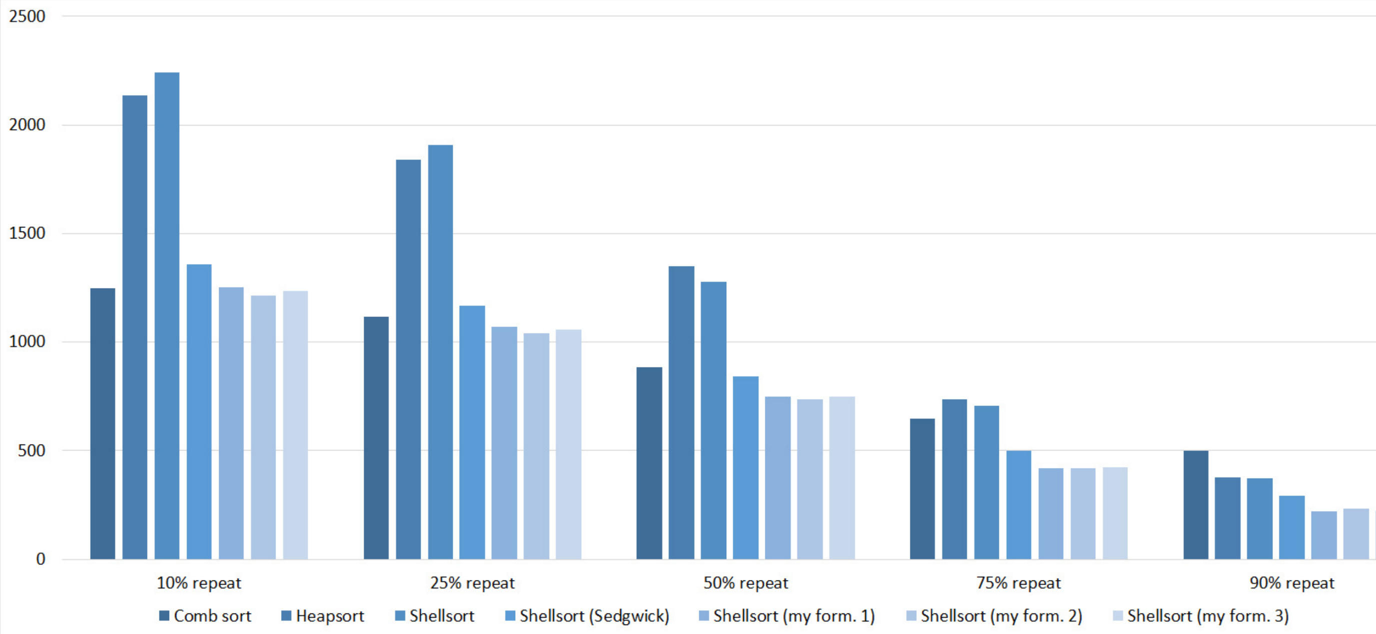

1200
1000
800
600
400
200
0
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat


1200
1000
800
600
400
200
0
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


1200
1000
800
600
400
200
0
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


2500
2000
1500
1000
500
0
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat

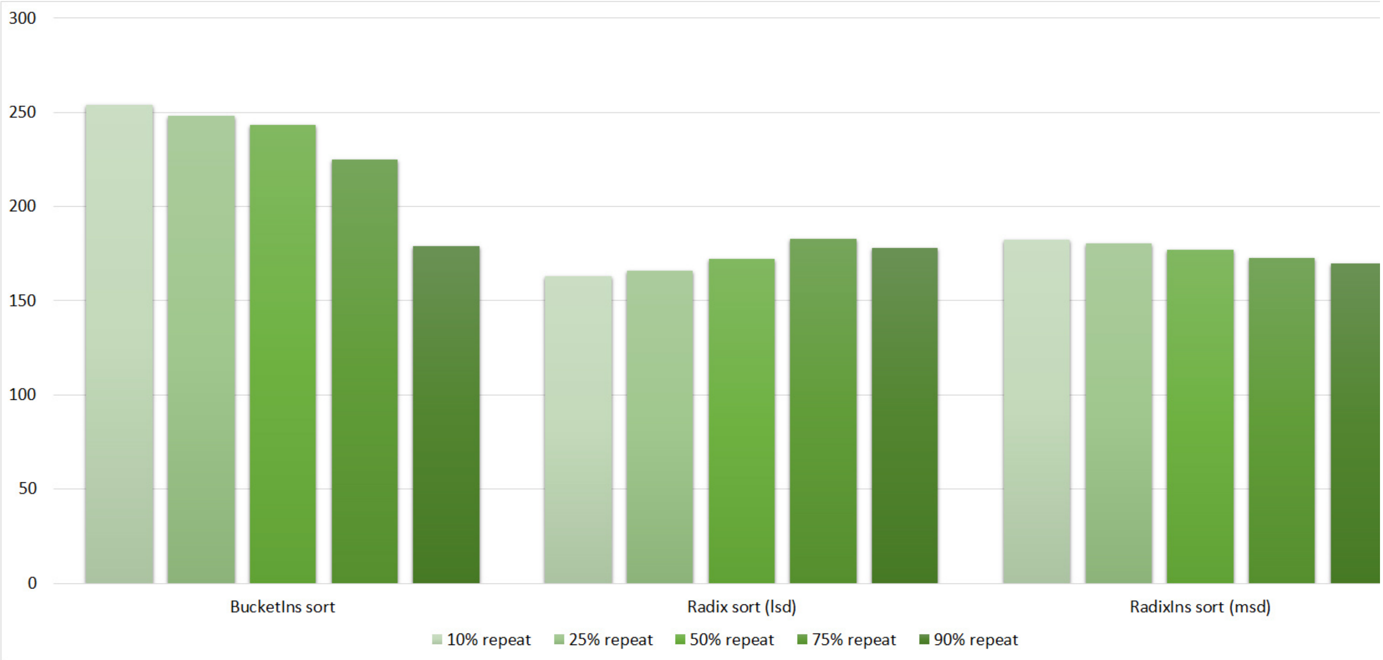

2500
2000
1500
1000
500
0
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my form. 3)

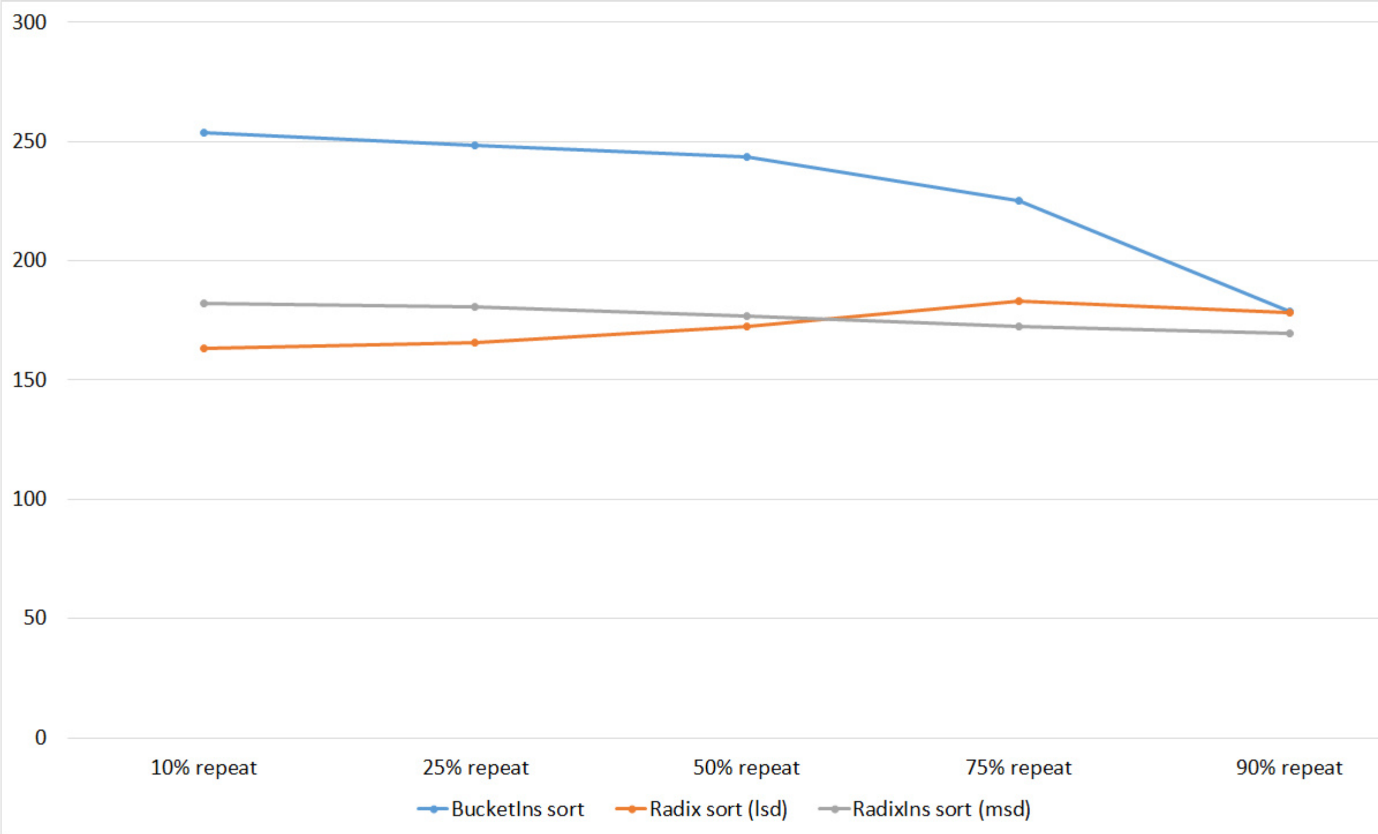


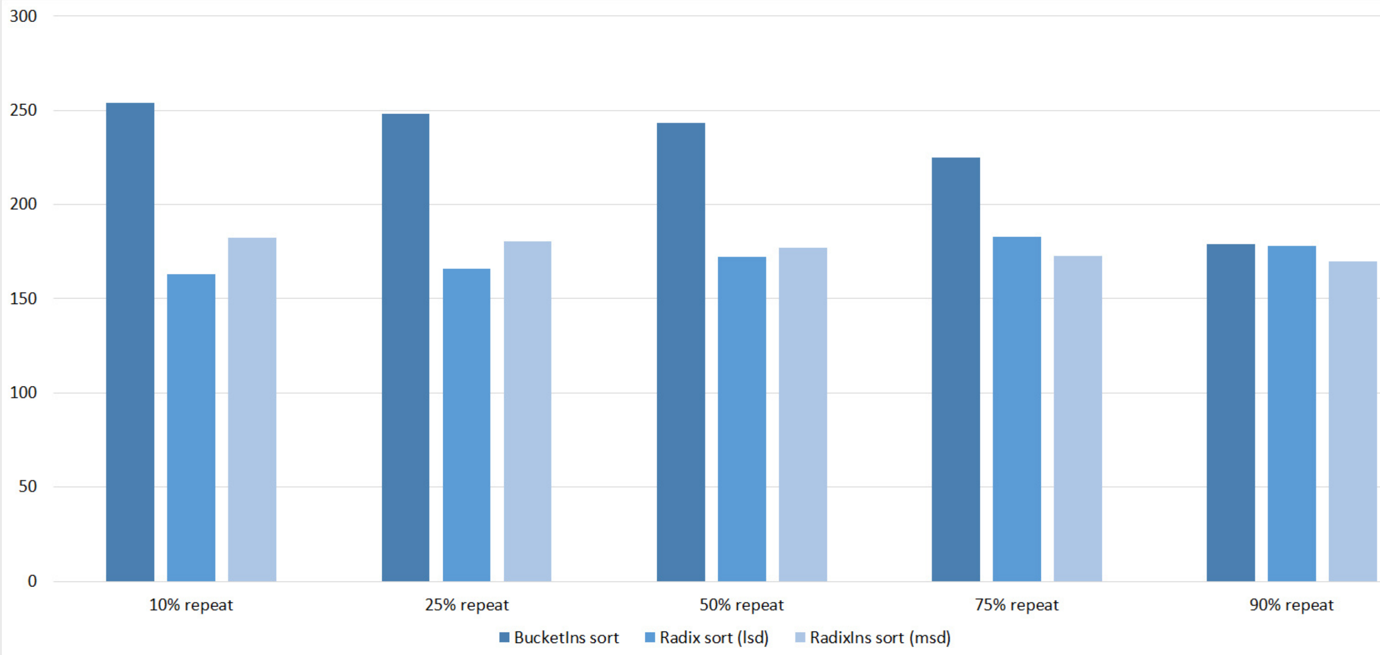
1e7	10% repeat	25% repeat	50% repeat	75% repeat	90% repeat	Итог
Comb sort	1249,78	1118,33	885,35	647,68	498,75	879,9
BucketIns sort	253,80	248,25	243,35	225,13	178,98	229,9
Heapsort	2136,67	1839,70	1347,25	737,85	374,60	1287,2
sort (stl)	781,68	656,43	443,15	213,63	86,98	436,3
Quicksort	702,08	630,77	467,58	326,50	233,00	471,9
QuickIns sort	623,72	546,62	410,58	275,33	192,37	409,7
Merge sort	1029,87	946,68	787,52	617,13	508,28	777,9
MergeIns sort	948,88	863,32	708,10	551,77	438,62	702,1
Shellsort	2241,87	1906,72	1278,07	705,27	374,27	1301,2
Shellsort (Sedgwick)	1359,18	1167,48	839,77	497,40	293,08	831,3
Radix sort (lsd)	163,17	165,92	172,43	182,92	178,20	172,5
RadixIns sort (msd)	182,23	180,50	176,83	172,72	169,58	176,3
Timsort	919,82	839,07	692,42	515,97	403,83	674,2
Shellsort (my form. 1)	1253,77	1070,38	749,05	417,67	218,30	741,8
Shellsort (my form. 2)	1213,65	1042,20	735,33	420,65	231,35	728,6
Shellsort (my form. 3)	1234,97	1059,02	747,27	422,03	224,02	737,4

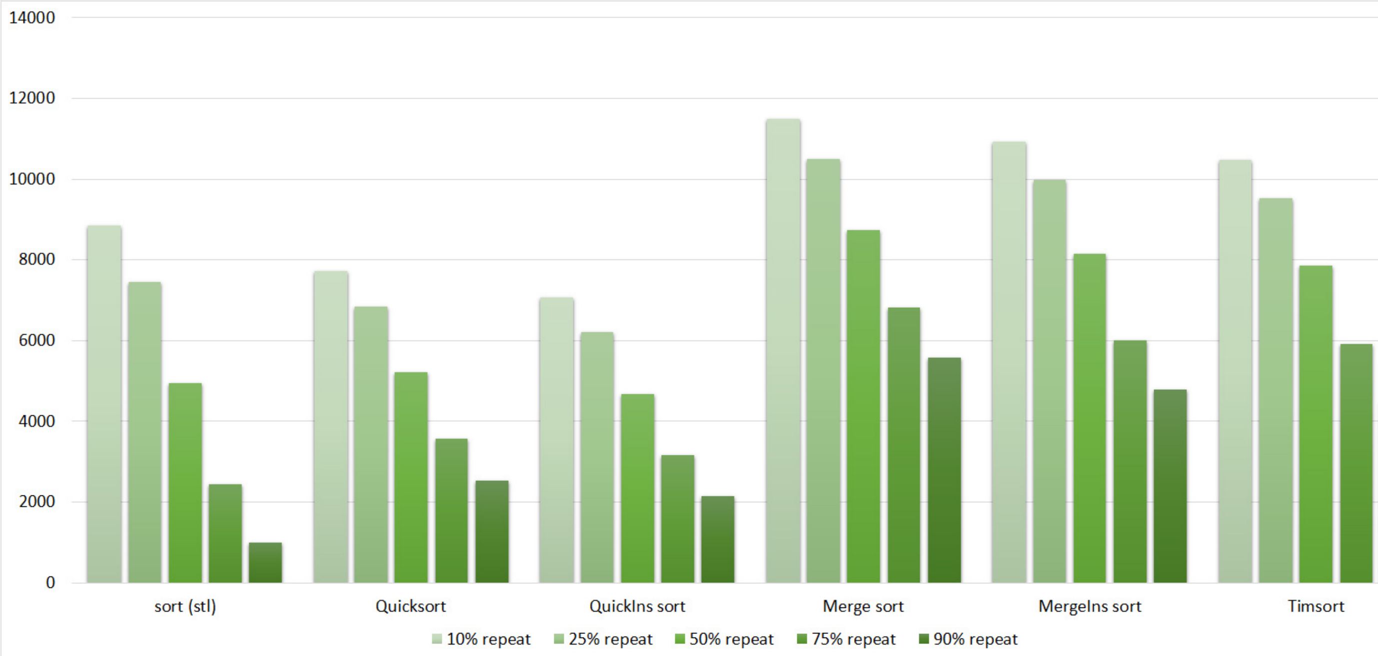
1e7	Итог
Radix sort (lsd)	172,53
RadixIns sort (msd)	176,37
BucketIns sort	229,90
QuickIns sort	409,72
sort (stl)	436,38
Quicksort	471,99
Timsort	674,22
MergeIns sort	702,14
Shellsort (my form. 2)	728,64
Shellsort (my form. 3)	737,46
Shellsort (my form. 1)	741,83
Merge sort	777,90
Shellsort (Sedgwick)	831,38
Comb sort	879,98
Heapsort	1287,21
Shellsort	1301,24

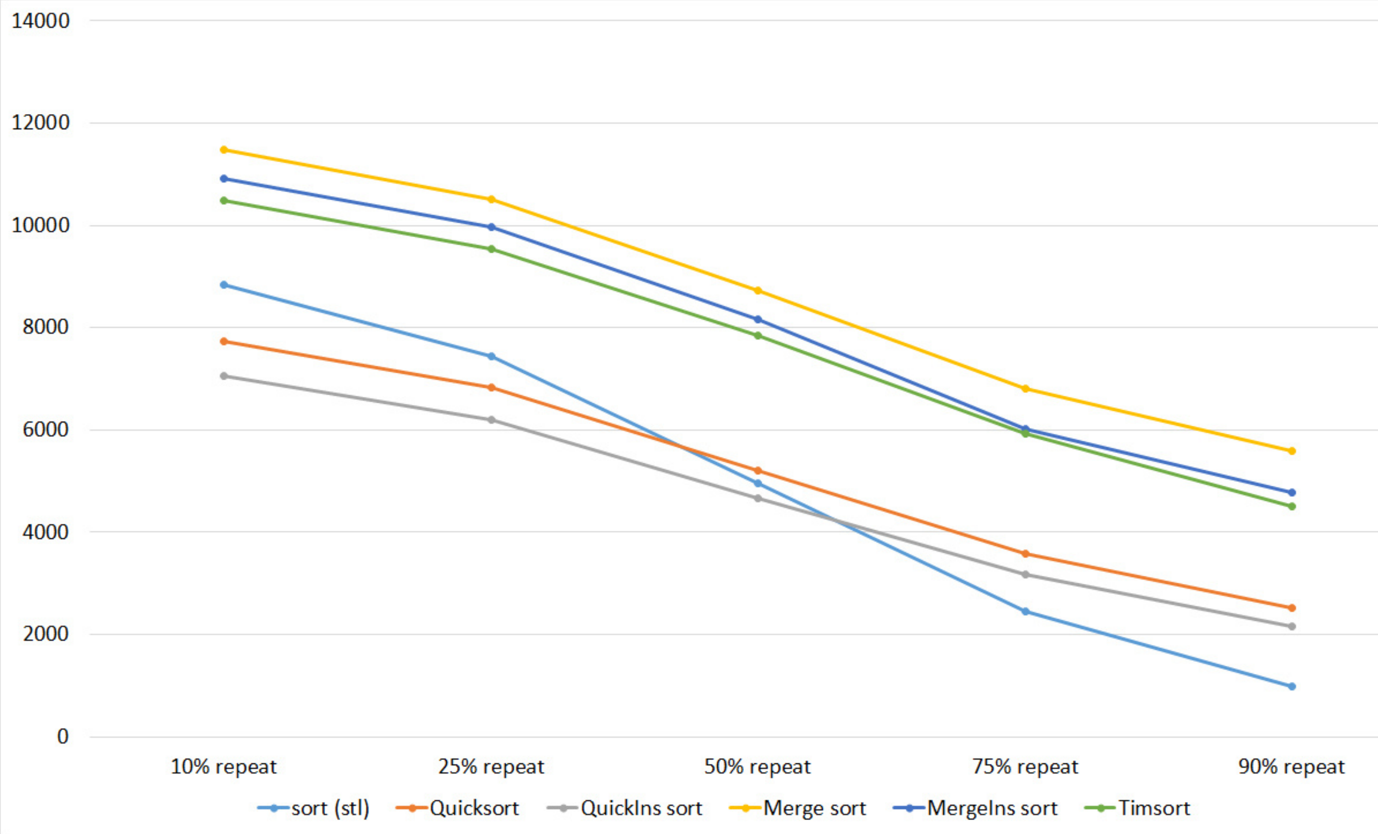
Таблицы, 1e8 элементов

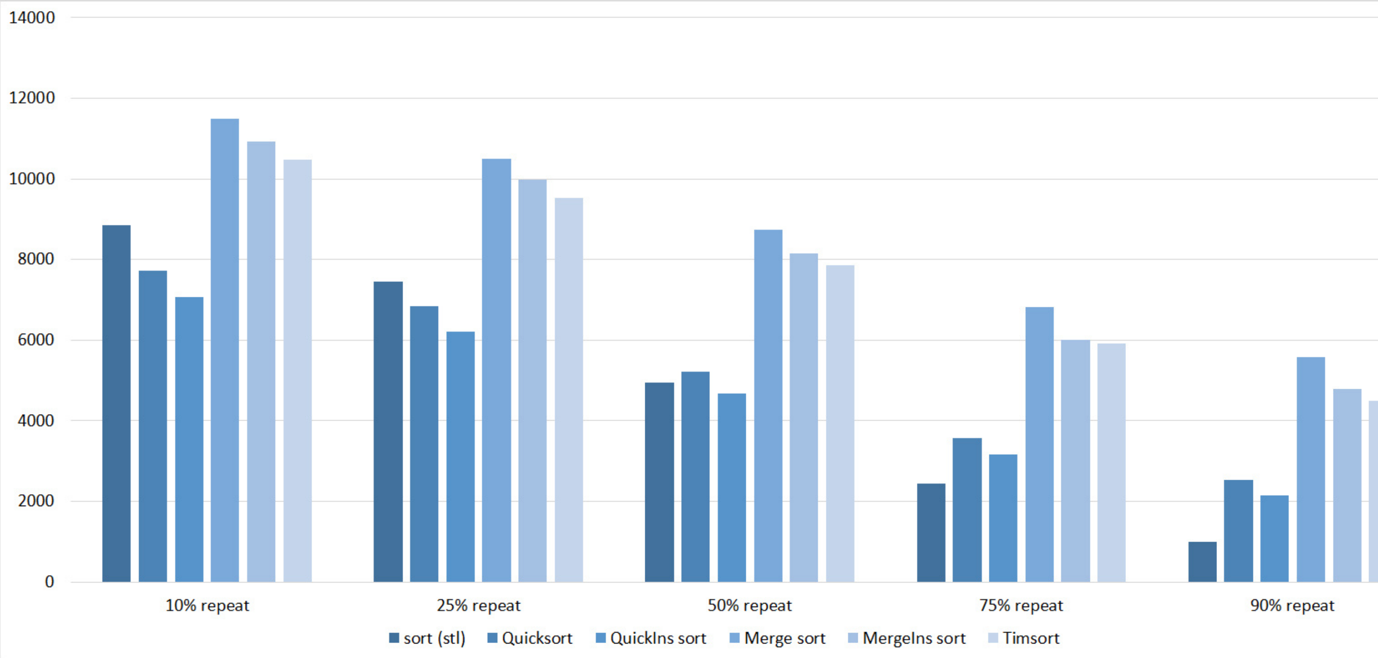

300
250
200
150
100
50
0
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat

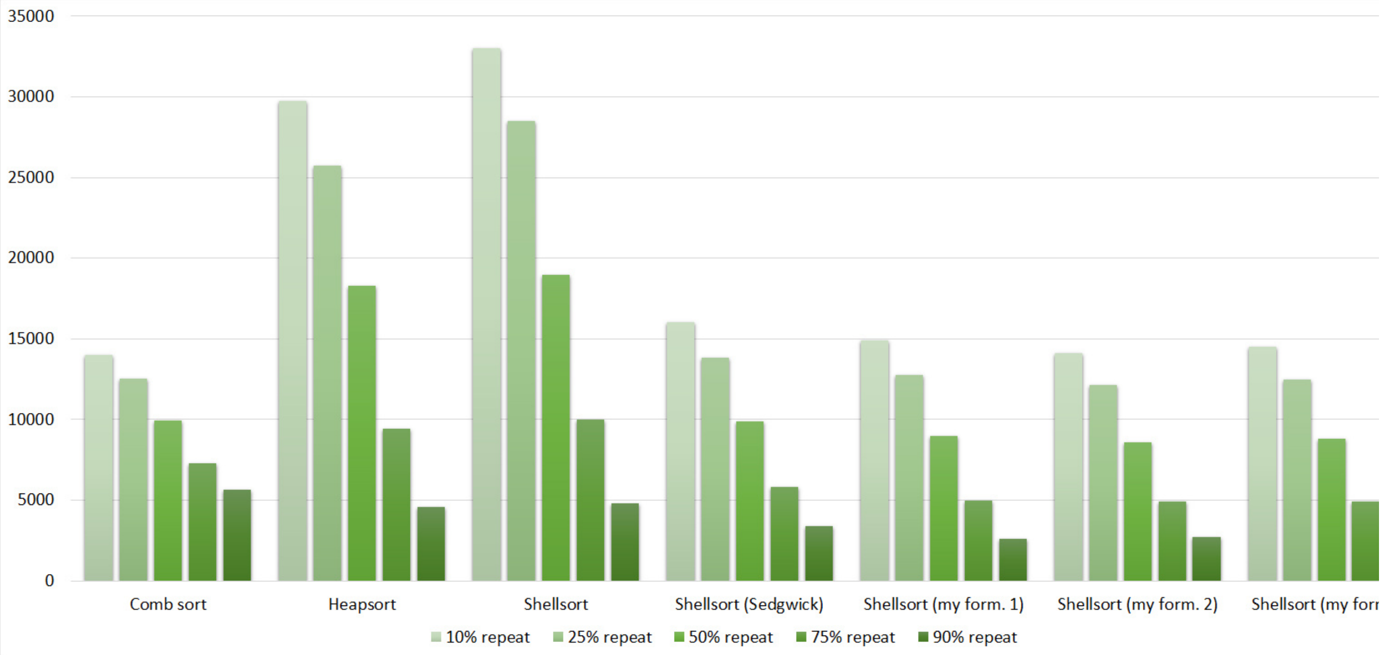

300
250
200
150
100
50
0
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

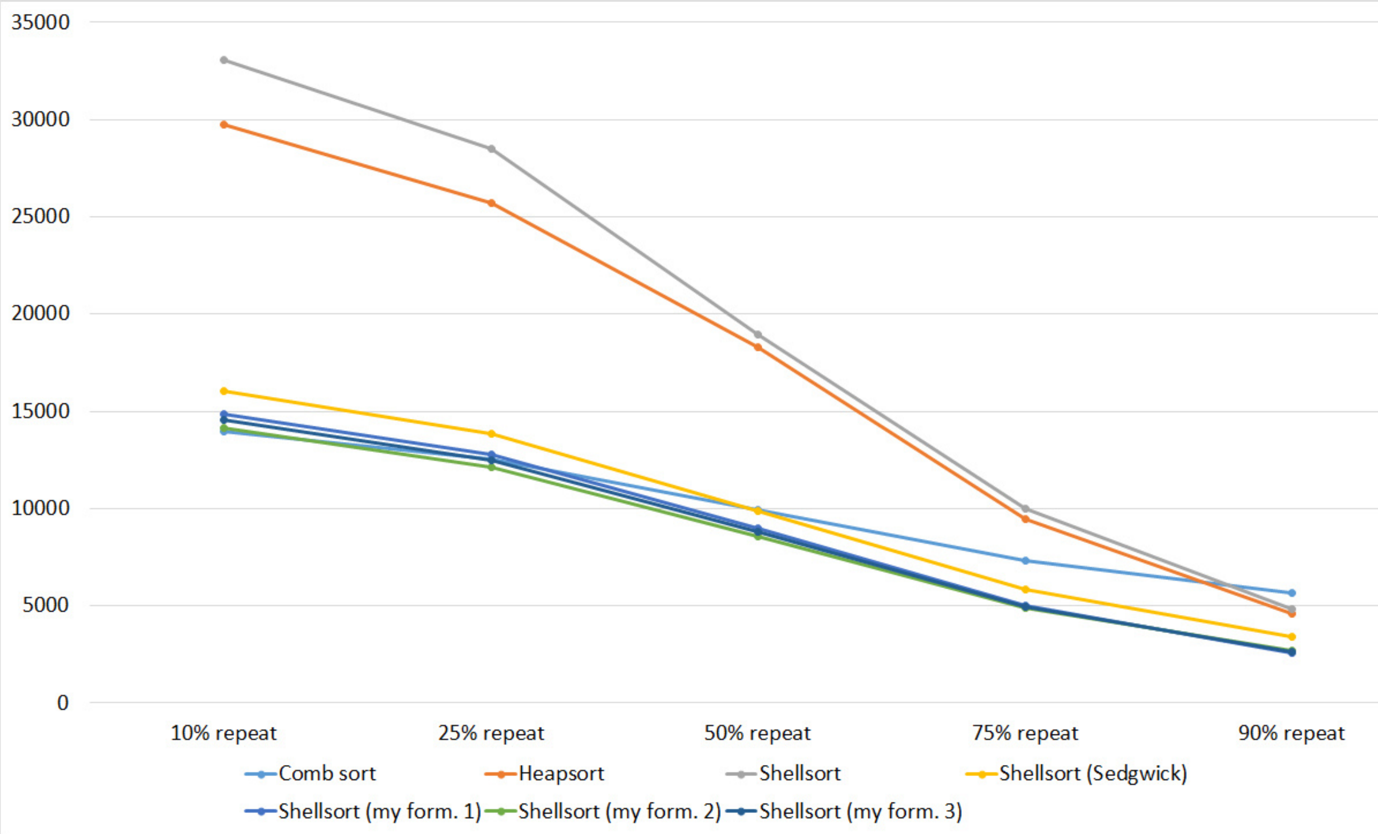

300
250
200
150
100
50
0
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat
BucketIns sort
Radix sort (lsd)
RadixIns sort (msd)

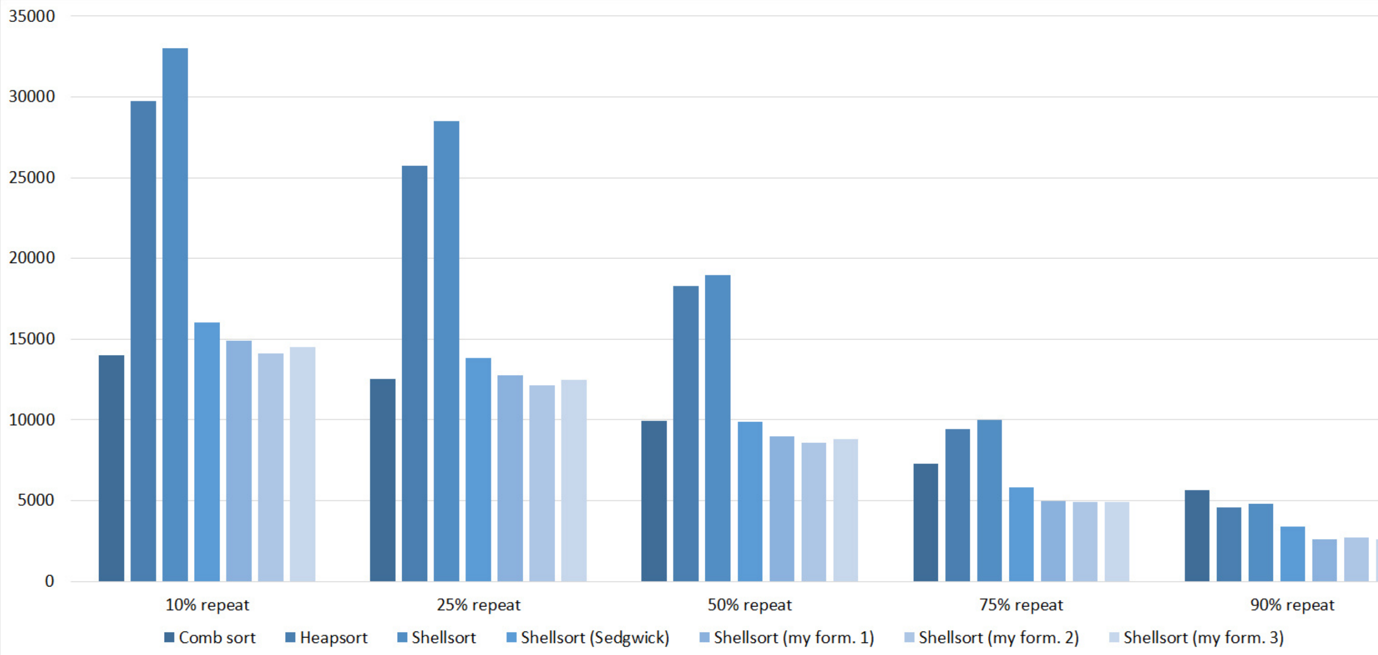

14000
12000
10000
8000
6000
4000
2000
0
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat


14000
12000
10000
8000
6000
4000
2000
0
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


14000
12000
10000
8000
6000
4000
2000
0
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat
sort (stl)
Quicksort
QuickIns sort
Merge sort
MergeIns sort
Timsort


35000
30000
25000
20000
15000
10000
5000
0
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my forr
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat


35000
30000
25000
20000
15000
10000
5000
0
10% repeat
25% repeat
50% repeat
75% repeat
90% repeat
Comb sort
Heapsort
Shellsort
Shellsort (Sedgwick)
Shellsort (my form. 1)
Shellsort (my form. 2)
Shellsort (my form. 3)



1e8	10% repeat	25% repeat	50% repeat	75% repeat	90% repeat	Итог
Comb sort	13991,25	12532,98	9923,05	7299,60	5670,35	9883,4
BucketIns sort	3380,45	3097,45	2652,22	2385,00	2112,25	2725,4
Heapsort	29761,67	25728,88	18281,65	9455,43	4570,73	17559,
sort (stl)	8846,17	7443,35	4947,13	2446,43	994,08	4935,4
Quicksort	7722,43	6837,73	5207,72	3572,57	2520,73	5172,2
QuickIns sort	7057,48	6208,30	4672,62	3170,70	2151,77	4652,1
Merge sort	11481,47	10508,30	8730,72	6809,03	5583,13	8622,5
MergeIns sort	10922,08	9978,40	8153,03	6014,62	4784,93	7970,6
Shellsort	33036,12	28484,62	18945,62	10016,62	4799,35	19056,
Shellsort (Sedgwick)	16058,00	13842,02	9899,88	5812,62	3412,48	9805,0
Radix sort (lsd)	1599,38	1672,00	1694,82	1759,62	1790,85	1703,3
RadixIns sort (msd)	2485,33	2184,45	1760,35	1560,57	1496,00	1897,3
Timsort	10475,78	9534,38	7854,37	5921,35	4503,85	7657,9
Shellsort (my form. 1)	14878,17	12753,25	8953,90	4987,50	2596,28	8833,8
Shellsort (my form. 2)	14134,73	12124,83	8588,57	4900,40	2686,60	8487,0
Shellsort (my form. 3)	14527,18	12455,02	8784,20	4931,30	2615,17	8662,5

1e8	Итог
Radix sort (lsd)	1703,33
RadixIns sort (msd)	1897,34
BucketIns sort	2725,47
QuickIns sort	4652,17
sort (stl)	4935,43
Quicksort	5172,24
Timsort	7657,95
MergeIns sort	7970,61
Shellsort (my form. 2)	8487,03
Merge sort	8622,53
Shellsort (my form. 3)	8662,57
Shellsort (my form. 1)	8833,82
Shellsort (Sedgwick)	9805,00
Comb sort	9883,45
Heapsort	17559,67
Shellsort	19056,46

Здесь все довольно тоскливо, все сортировки работают с одинаковой динамикой (кроме линейных). Из необычного можно заметить, что сортировка слиянием упала ниже сортировки Шелла.

Итоговые результаты

	1e7	Итог		1e8	Итог
1	Radix sort (lsd)	173,35	1	Radix sort (lsd)	1779,13
2	RadixIns sort (msd)	180,94	2	RadixIns sort (msd)	2144,92
3	BucketIns sort	191,96	3	BucketIns sort	2333,34
4	QuickIns sort	351,47	4	QuickIns sort	3834,59
5	Quicksort	395,66	5	Quicksort	4256,18
6	sort (stl)	441,10	6	sort (stl)	4840,12
7	Timsort	484,09	7	Timsort	5429,52
8	MergeIns sort	586,95	8	MergeIns sort	6624,72
9	Merge sort	665,46	9	Merge sort	7373,04
10	Shellsort (my form. 2)	803,19	10	Shellsort (my form. 2)	9311,80
11	Shellsort (my form. 3)	812,70	11	Shellsort (my form. 3)	9446,57
12	Shellsort (my form. 1)	821,89	12	Shellsort (my form. 1)	9611,50
13	Shellsort (Sedgwick)	897,38	14	Comb sort	10416,12
14	Comb sort	923,77	13	Shellsort (Sedgwick)	10759,37
15	Heapsort	1410,40	15	Heapsort	18977,69
16	Shellsort	1465,45	16	Shellsort	22097,73

Несмотря на мои старания, LSD-версия поразрядной сортировки все-таки заняла первое место и при 10^7, и при 10^8 элементов. Также она продемонстрировала почти линейный рост времени. Единственная ее замеченная мной слабость — плохая работа с перестановками. MSI версия сработала немного хуже, в первую очередь из-за большого количества тестов, состоящих из случайных чисел по модулю 10^9. Реализацией блочной сортировки я остался доволен, несмотря на громоздкость, она показало неплохой результат. Кстати, я слишком позд заметил, она не до конца соптимизирована, можно еще отдельно создавать массивы run и cnt, чтобы не тратить время на их удаление. Да уверенно заняли места различные версии быстрой сортировки. Timsort-у не удалось, на мой взгляд, оказать им серьезную конкуренцию, хо не сильно отстал. Далее по скорости идут сортировки слиянием, после них — мои версии сортировки Шелла. Лучше всего оказалась последовательность s * 3 + s / 3, где s — предыдущий элемент последовательности. Далее идет единственное расхождение в двух таблица сортировка расческой оказалась лучше при большем числе элементов, чем сортировка Шелла с последовательностью Седжвика. И за пос место боролись пирамидальная сортировка и оригинальная сортировка Шелла.

Выиграла последняя. Кстати, сортировка Шелла, как я потом проверил, очень плохо работает на тестах размера 2^n, так что ей просто пове что она попала в первую группу.

Если говорить о практическом применении, то хороша поразрядная сортировка (особенно lsd-версия), она стабильна, проста в реализаци очень быстра, однако не основана на сравнениях. Из основанных на сравнениях сортировок лучше всего смотрится быстрая сортировка. Е недостатки — неустойчивость и квадратичное время работы на неудачных входных данных (пусть они и могут встретиться только при намеренном создании теста). Но с этим можно бороться, например, выбирая опорный элемент по какому-нибудь другому принципу, или же переходя на другую сортировку при неудаче (например, introsort, который, если не ошибаюсь, и реализован в C++). Timsort лишен этих недостатков, лучше работает на сильно отсортированных данных, но все же медленнее в целом и гораздо сложнее пишется. Остальные сортировки на данный момент, пожалуй, не очень практичны. Кроме, конечно, сортировки вставками, которую весьма удачно иногда можно вставить в алгоритм.

Заключение

Должен отметить, что не все известные сортировки приняли участие в тестировании, например, была пропущена плавная сортировка (мне не удалось ее адекватно реализовать). Впрочем, не думаю, что это большая потеря, эта сортировка очень громоздкая и медленная, как мо видеть, например, из этой статьи: habrahabr.ru/post/133996 Еще можно исследовать сортировки на распараллеливание, но, во-первых, у ме опыта, во-вторых, результаты, которые получались, крайне нестабильны, очень велико влияние системы.

Здесь можно посмотреть результаты всех запусков, а также некоторые вспомогательные тестирования: ссылка на документ.

Здесь можно посмотреть код всего проекта

Реализации алгоритмов с векторами остались, но их корректность и хорошую работу не гарантирую. Проще взять коды функций из статьи переделать. Генераторы тестов тоже могут не соответствовать действительности, на самом деле такой вид они приняли уже после создан тестов, когда нужно было сделать программу более компактной.

В общем, я доволен проделанной работой, надеюсь, что Вам было интересно.

Метки: тестирование производительности, алгоритмы сортировки, сортировка пузырьком, шейкерная сортировка, сортировка выбором, сортировка вставками гномья сортировка, битонная сортировка, сортировка деревом, сортировка шелла, поразрядная сортировка, блочная сортировка, быстрая сортировка, сортировка слиянием, сортировка расчёской, пирамидальная сортировка

Михаил Опанасенко @OMS7
Пользователь

5 декабря 2011 в 21:42

И снова про сортировки: выбираем лучший алгоритм

+111 68,3k 312 32

Комментарии 20

ZaMaZaN4iK 18 августа 2017 в 17:11

Есть такая репа с сортировками: github.com/Morwenn/cpp-sort

Также стоило бы упомянуть такие сортировки как Boost.Sort и pdqsort, которая используется в Rust: они являются очень быстрыми.

А за бенчмарк спасибо — будет полезная ссылка.

andy_p 18 августа 2017 в 17:46

Почему производилось по 20 запусков на тест?
Что за системные глюки наблюдались?
Как измерялось время?
Имеет ли смысл сделать то же самое под Linux?

OMS7 18 августа 2017 в 19:55

Нужно было сделать столько запусков, чтобы это не заняло слишком много времени, и при этом давало достаточно надежные результаты. 20 — ı
подходящее число.
Пару раз за время тестирования система зависала на несколько минут, почему — не знаю.
Время измерялось встроенной в C++ функцией clock(), можете посмотреть в реализации.
Насчет Linux'а не знаю, у меня его нет.

bopoh13 18 августа 2017 в 18:08

> Представим каждое число в двоичном виде

А если в массиве не числа? Предлагаю изменить заголовок.

patex 18 августа 2017 в 18:08

О-нотация верхний предел его настоящей сложности
для точного и наилучшего используются другие обозначения Θ и Ω(ω)
см Введение в анализ сложности алгоритмов на хабре

overmes 18 августа 2017 в 18:36

Сейджвик как-то в курсе по алгоритмам говорил, что на производительность сильно влияет как хорошо алгоритмы работают с кэшами процессора.
Например merge sort хорошо с ними работает, a heap sort наоборот. Поэтому даже с учетом того что merge sort требует больше памяти его предпочи
heap sort.

mikhaelkh 18 августа 2017 в 18:46

Heapsort на C++ пишется в 2 строчки:

```
std::make_heap(l, r);
std::sort_heap(l, r);
```

Построение кучи требует O(r-l) времени.
Heapsort требует O(1) дополнительной памяти.

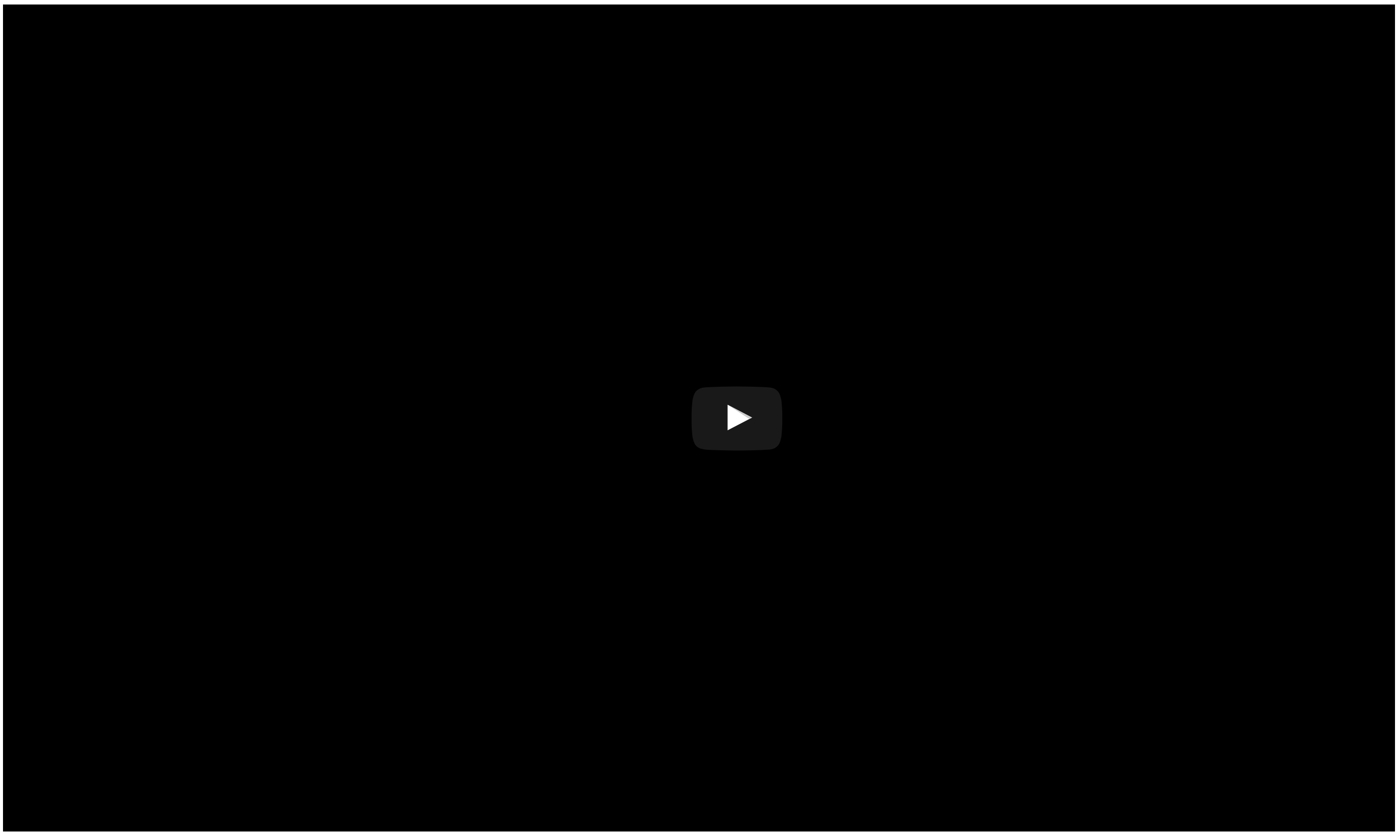
OMS7 18 августа 2017 в 20:18

Да, но это работает гораздо медленнее приведенной реализации.

mikhaelkh 18 августа 2017 в 21:35

На самом деле мой вариант даже быстрее из-за линейного времени построения кучи.
pastebin.com/BRRHx0pd

gresolio 18 августа 2017 в 19:04



Pochemuk 18 августа 2017 в 19:21

Занятно, но ощущение противоречивое…

Во-первых, мне кажется, для одной статьи это слишком большое число алгоритмов. Было бы лучше сгруппировать их в 2-3 группы. Например, по устойчивости (сохранении порядка элементов с одинаковым значением ключа) или по другому критерию.

Во-вторых, не хватает сводной таблички, в которой бы указывались такие характеристики:

1. Устойчивость — сохраняется ли порядок элементов с одинаковыми значениями ключей.
2. Естественность — сортируется ли уже упорядоченный массив быстрее, чем случайный, а случайный быстрее, чем упорядоченный в обратном по
3. Производительность в лучшем, среднем, худшем случаях.
4. Требования к дополнительной памяти (и ее порядок).
5. Области применения (например, для сортировок малых почти упорядоченных массивов и др.).

И еще… Могу, конечно, путать, но, как мне кажется, сортировка слиянием существует в двух вариантах — с дополнительной памятью размера O(n)
дополнительной памяти — т.е. размера O(1).
И производительность этих сортировок весьма различается.

mikhaelkh 18 августа 2017 в 19:30

Кстати, существует ли устойчивая сортировка с временем работы O(n*log(n)) и дополнительной памятью O(1) в наихудшем случае? По-моему где
её встречал, но не помню где.

Pochemuk 18 августа 2017 в 19:36

Неужели такое бывает? Я думал, что все устойчивые имеют временную сложность O(n^2)…

mikhaelkh 18 августа 2017 в 19:46

Нашёл!

Pochemuk 18 августа 2017 в 20:00

Круто :) Но я ничего не понял :D

DaylightIsBurning 18 августа 2017 в 20:35

Почему n^2? Если даже взять наивную реализацию, то можно добавить каждому элементу второй ключ по его порядковому номеру в изнача массиве [O(n)], а затем отсортировать по двум ключам — выходит O(2*n*log(n) + n) = O(n*log(n)). Merge Sort тоже стабильный.

AxianLTD 18 августа 2017 в 19:38

А что нового внесла статья по сравнению с книгой Д.Кнута?

Pochemuk 18 августа 2017 в 19:42

Ну, кажется у Кнута многих приведенных алгоритмов нет. Может быть в каких-то последних изданиях — исправленных и дополненных, разве что.

ixa 18 августа 2017 в 20:46

Интересно и хорошо сконцентрировано…
Мне к примеру, надо быстро сортировать 20и битовые данные и их 100 000 000.
Я раньше считал, что самый подходящий способ делёж файла на 30 частей и — Merge Sort.
Сейчас вижу есть другие варианты…
Ну а всё же — что наиболее подходит?

ZaMaZaN4iK 18 августа 2017 в 21:04

открываете репозиторий cpp-sort, запускаете на Вашем кейсе и смотрите, что показывает себя лучше. Тем и пользуйтесь :-)