Казанский (Приволжский) Федеральный университет

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ
КАФЕДРА СИСТЕМНОГО АНАЛИЗА И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

ОТЧЕТ ПО ПРАКТИЧЕСКОМУ ПРОЕКТУ

Дисциплина “Алгоритмы и анализ сложности”

Экспериментальный анализ различных методов сортировки

Выполнила:

Кузьмина Виктория Александровна
Группа 09-832

Проверил:

Доцент, к.ф.-м.н.
Васильев Александр Валерьевич

Казань, 2020

Содержание:

Введение

Для сравнительного анализа я выбрала следующие методы сортировки:

1. Сортировка пузырьком (BubbleSort)
2. Сортировка пузырьком «с флагом» (BubbleSortCheck)
3. Сортировка выбором (ChoiceSort)
4. Сортировка вставками (InsertSort)
5. Сортировка расческой (BrushSort)
6. Быстрая сортировка (QuickSort)
7. Сортировка слиянием (MergeSort)
8. Сортировка кучей (HeapSort)
9. Поразрядная сортировка (RankSort)
10. Сортировка, встроенная в язык программирования (CSort)

Вся работа выполнения на языке C++, в среде разработки Visual Studio 2017. Соответственно, под сортировкой, встроенной в язык программирования, я буду понимать алгоритм sort из библиотеки <algorithm>. Языковым стандартом не предусмотрена единая функция сортировки, но C++ требует, чтобы вне зависимости от реализации сложность этого алгоритма в худшем случае была не больше $n * \log(n)$.

Тестирования проводились над цифрами, числами (случайного размера или фиксированного), строками (случайного размера или фиксированного) и над датами. Дату я реализовала в виде отдельного класса с переопределенными операторами сравнения, поэтому этот вид подаваемых данных можно обобщить до любого класса, объекты типа которого могут быть сравнены.

Контейнером для данных был шаблонный массив, для удобства для него тоже реализован класс. Число элементов варьировалось от 50 до 500.000 штук, с различными промежуточными измерениями. Контейнер заполнялся тремя различными способами:

1. Случайным образом (даты были случайными только в рамках года с 1000 до 2500, месяцем от 1 до 12 и днем, в зависимости от количества дней в конкретном месяце)
2. Частично-упорядоченным: первая половина массива заполнялась рандомными элементами в пределах первой половины линейки данных (если цифры – от 0 до 5, если строки – по первой букве от A до M), соответственно, вторая – в пределах второй половины.
3. Одинаковым элементом.

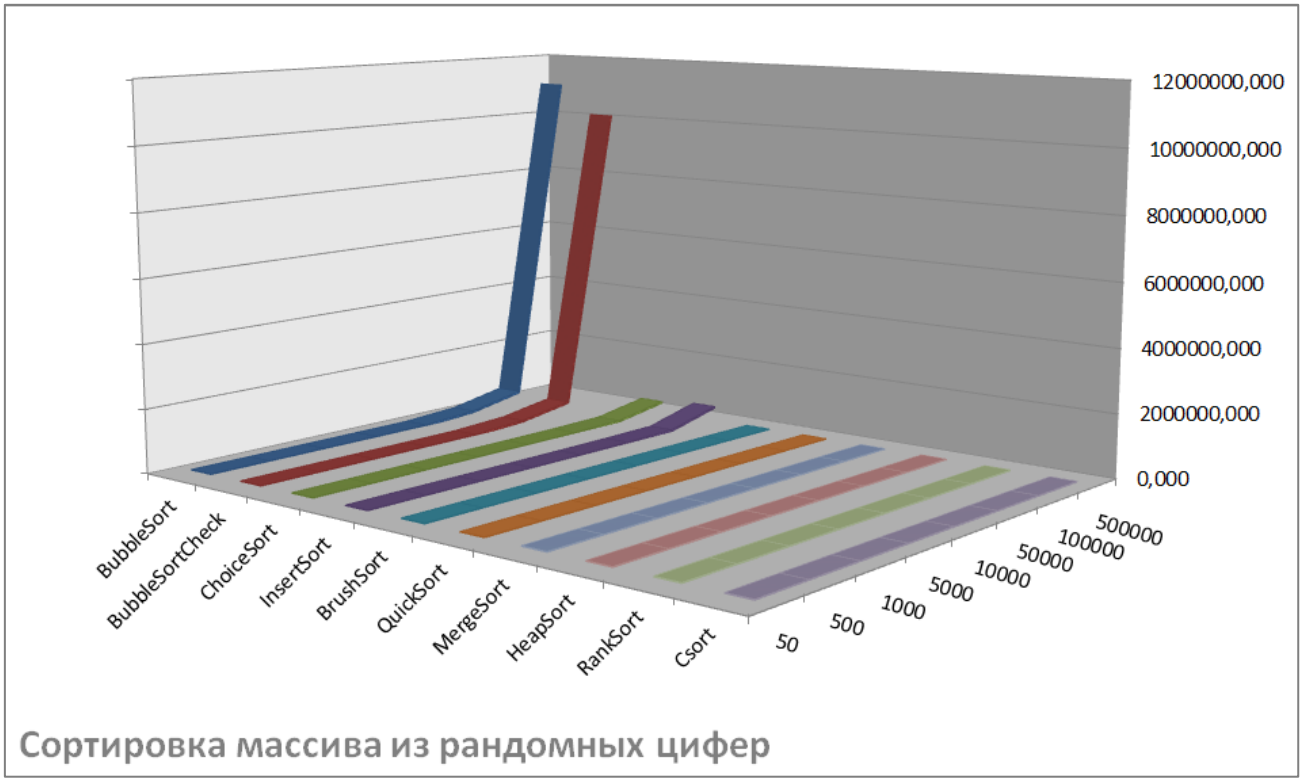
Для объективности, каждый эксперимент проводился трижды, на разных сгенерированных массивах, а в итоговую таблицу записывался средний результат. Во всех тестах время представлено в формате миллисекунд.

Для того чтобы анализ полученных результатов был структурирован, изначально я рассмотрю случайно сгенерированные массивы, потом частично-упорядоченные и однородные. Если будет необходимость, рассмотрю их в совокупности.

Случайно-сгенерированные массивы

Основное внимание я уделила этому разделу, потому что необходимость отсортировать такой массив, на мой взгляд, возникает чаще всего.

**Числа, выбранные рандомно, обычно попадали в промежуток (100.000; 10.000.000). А длина строк могла быть в промежутке (1;8).



По этому графику отчетливо видно, что рассматривать все методы сортировки в совокупности особого смысла не представляет, потому что чем больше размер массива, тем ярче заметна разница между простыми и быстрыми алгоритмами.

Оценка сложности алгоритма

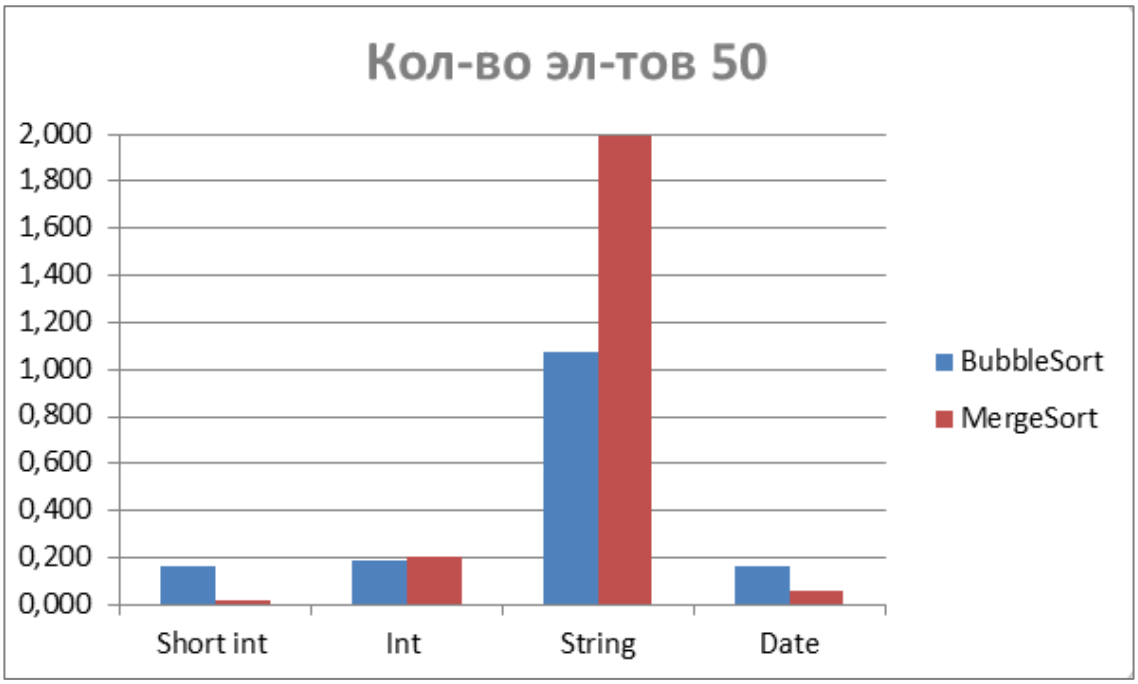
Из теории известно, что все рассматриваемые методы можно разделить на 3 подгруппы:
- простейшие сортировки, сложность которых $O(n) = n^2$
- промежуточные сортировки
- быстрые сортировки со сложностью $O(n) = n * \log(n)$

Простейшие сортировки в реализации тоже занимают квадратичное время, что заметно по квадратично-растущему времени, например, от 5.000 до 50.000. Этим можно было пользоваться для представления времени сортировки больших данных, с которыми процессор мог не справиться за несколько часов.

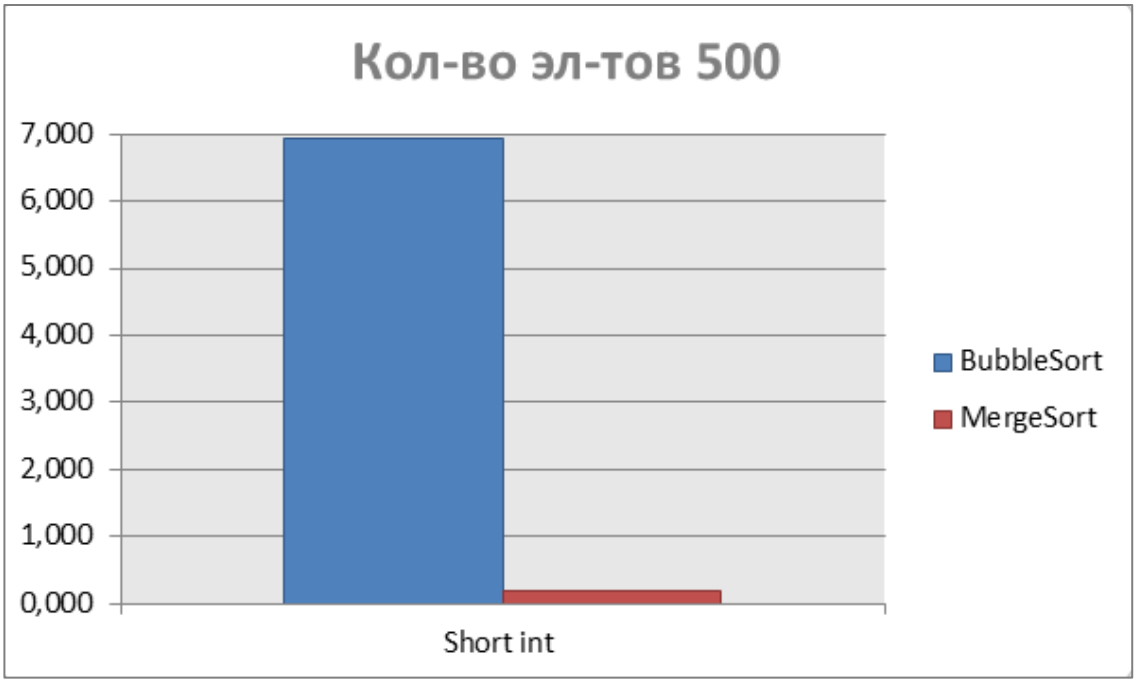
К промежуточным методам в моей работе относятся:
- Сортировка расческой (ее сложность в худшем $O(n) = n^2$, но на практике она работает значительно быстрее, поэтому буду рассматривать этот метод вместе с быстрыми сортировками)
- Поразрядная сортировка, ее сложность $O(n) = (k * (n + m))$, где k – длина одного кортежа, n – количество кортежей и m – количество символов в алфавите (если цифры, то m = 10). На практике поразрядная сортировка занимает большее время в связи с тем, что необходим еще один фактический проход по массиву для его слияния.

Изначально, я думала, что разные типы данных будут обрабатываться примерно с одинаковой скоростью, и значимое различие будет только у поразрядной сортировки, потому что она напрямую зависит от длины слова или числа. Но на практике оказалось, что для одной итерации сравнения двух цифр процессору нужно гораздо меньше времени, чем для сравнения чисел. Еще дольше сравниваются строки. Получается, что скорость работы алгоритма сильно зависит от количества запросов на сравнение элементов в 1 шаге работы алгоритма.

На графике ниже видно, что когда скорость обработки одной итерации сравнения довольно быстрая, как у цифр или дат, то быстрая сортировка оправдывает свою скорость. Но если сравнение занимает много времени, то выигрывает сортировка, которая реже посылает такие запросы.

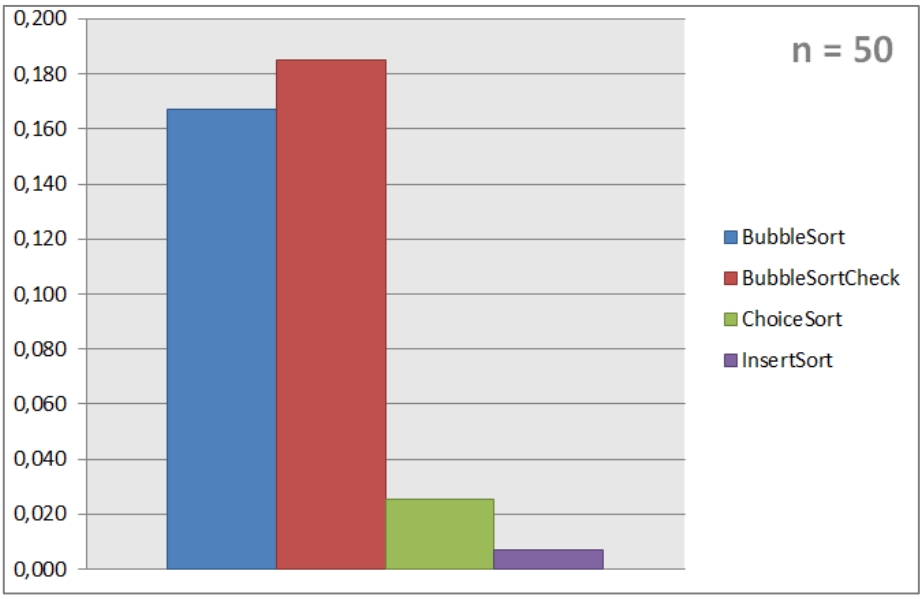


Безусловно, об этом можно говорить, рассматривая только небольшие размеры массивов, потому что с увеличением количества элементов скорость сравнения одной пары играет роль все меньше и меньше. Пример – сортировка массива из 500 цифр:

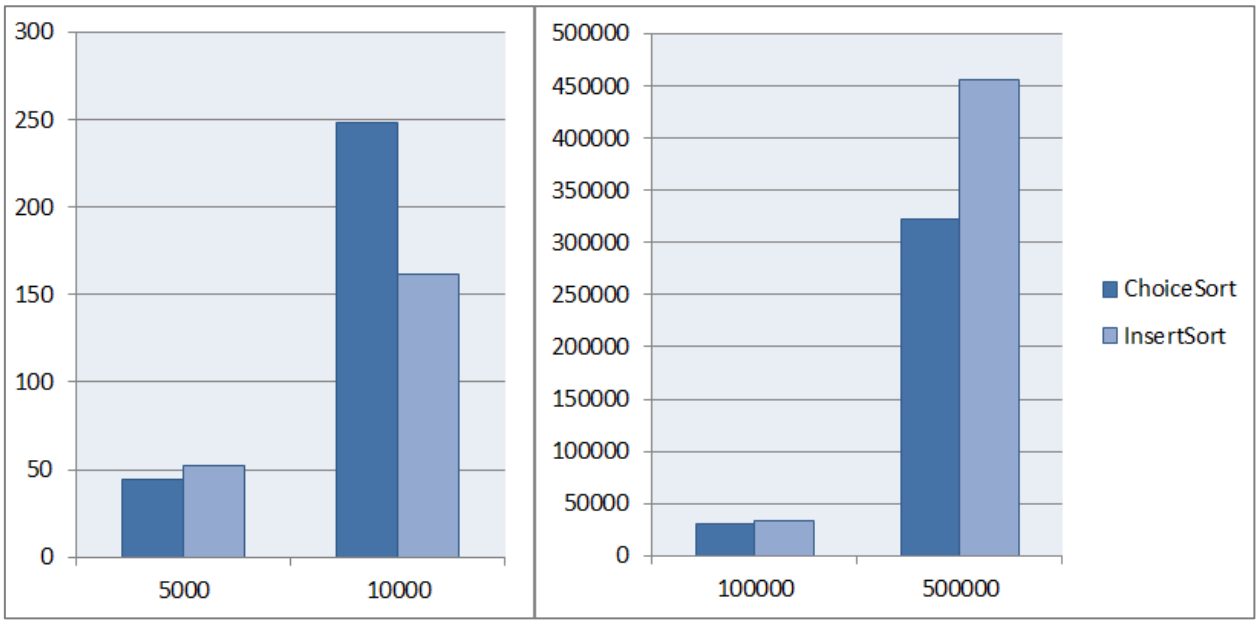


Простые сортировки

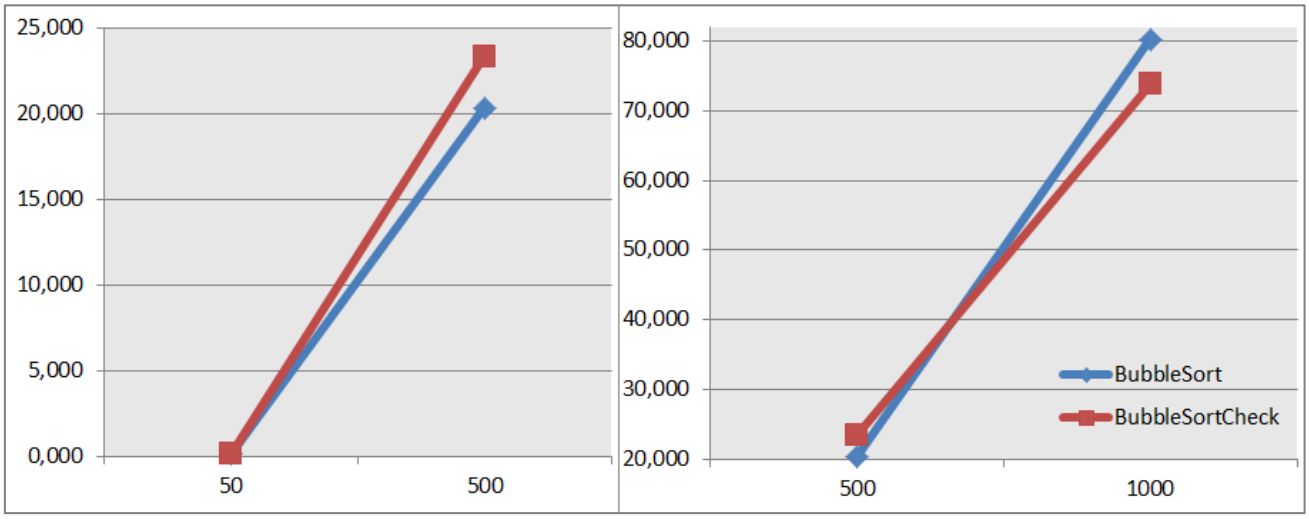
Поведение простых сортировок, в целом, оправдало ожидания. Сортировки пузырьком показали результат значительно хуже, чем сортировка вставками и выбором на любых типах данных и на любых количествах элементов в массиве.



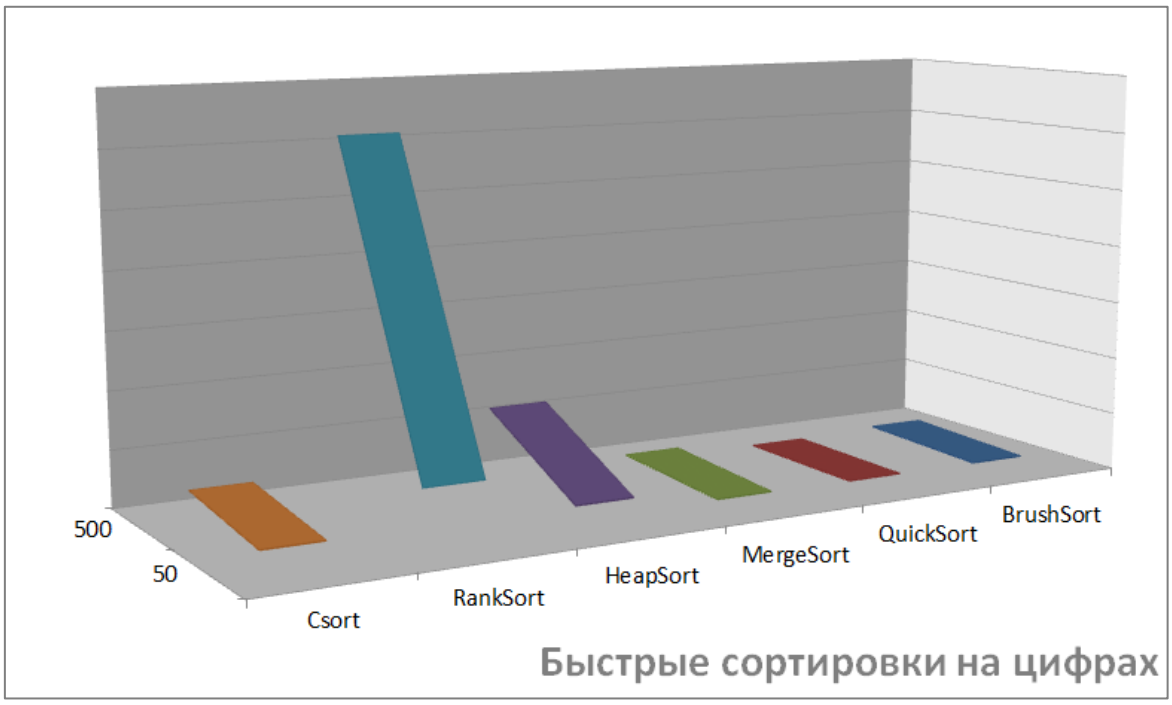
Лучшей оказалась сортировка вставками. Единственная неожиданность – на сортировке массивов коротких чисел от 0 до 9 лучший итоговый результат показала сортировка выбором, хотя их первенство постоянно чередовалась, что можно увидеть на гистограмме ниже. Мое предположение: это может быть связано с тем, что операция swap для маленьких элементов выполняется быстрее.



Еще, на протяжении всех тестирований, меня интересовал вопрос целесообразности использования сортировки пузырьком с дополнительной проверкой, не является ли массив уже отсортированным. На данном типе заполнения массива случайными элементами сортировка с проверкой никакой роли не сыграла, более того, из-за дополнительных строчек кода она иногда даже проигрывала обычной сортировке пузырьком.



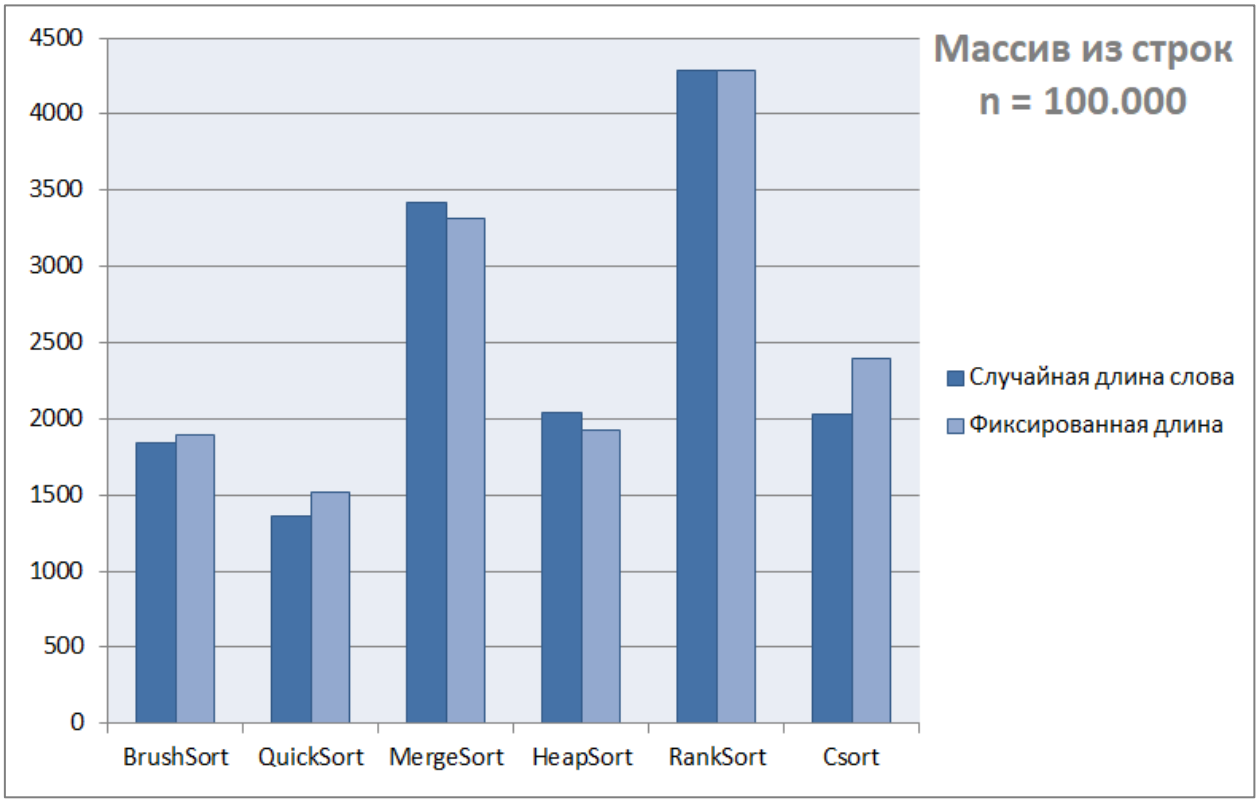
Быстрые сортировки (длинные массивы)



Замечание про ранговую сортировку:

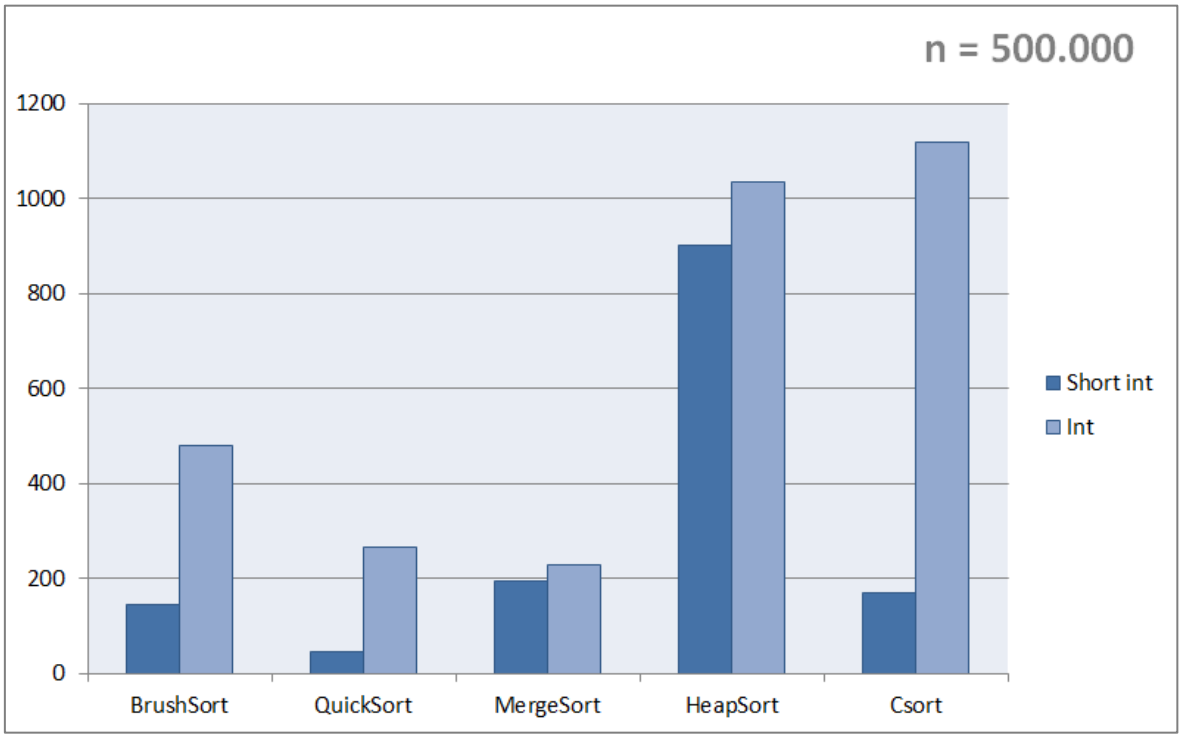
Ранговая сортировка сразу проявила себя хуже всех. Для массивов из цифр это нормально, для них целесообразнее было бы использовать сортировку подсчетом, без задействования шага «раскладки по карманам».

Но аналогичный результат она показала и на других типах данных. А ведь другие сортировки проводились для элементов со случайной длиной. Для чистоты эксперимента я протестировала быстрые сортировки на массивах из чисел длины 8 и строках длины 6.

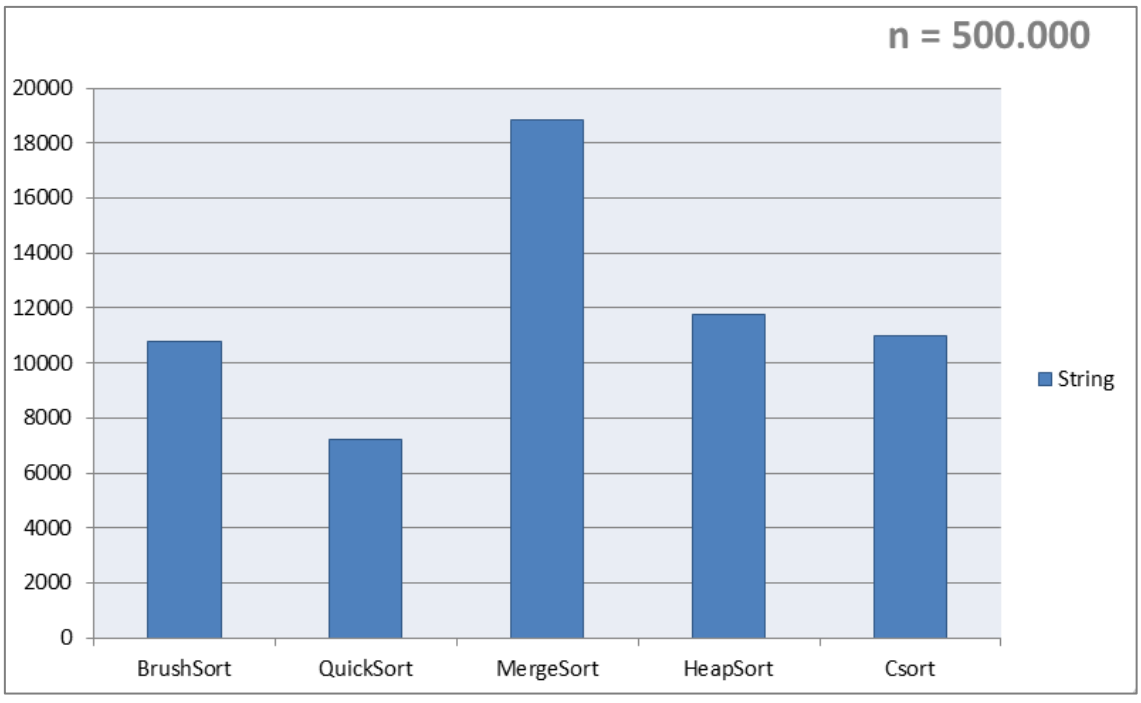


По гистограмме видно, что особой роли длина слова для других сортировок не играет. Поэтому ранговая сортировка далее рассматриваться не будет. Замечу только, что в реальной жизни этот метод очень удобно будет использовать для сортировки дат, потому что, упорядочив список дат по годам, велика вероятность получить уже полностью отсортированный массив. Если пока нет, то дальше целесообразно смотреть на месяц, в последнюю очередь на день.

Итак, среди быстрых сортировок для обработки больших числовых массивов лучше всего использовать «быструю сортировку». Она является самой универсальной из всех. По графику еще видно, что для массивов из длинных чисел целесообразнее использовать сортировку слиянием, а для коротких – сортировку расческой, несмотря на ее квадратичную сложность. Нельзя не заметить, что скорость работы встроенной сортировки на длинных числах резко взлетает, объяснения этому феномену я пока не нашла.



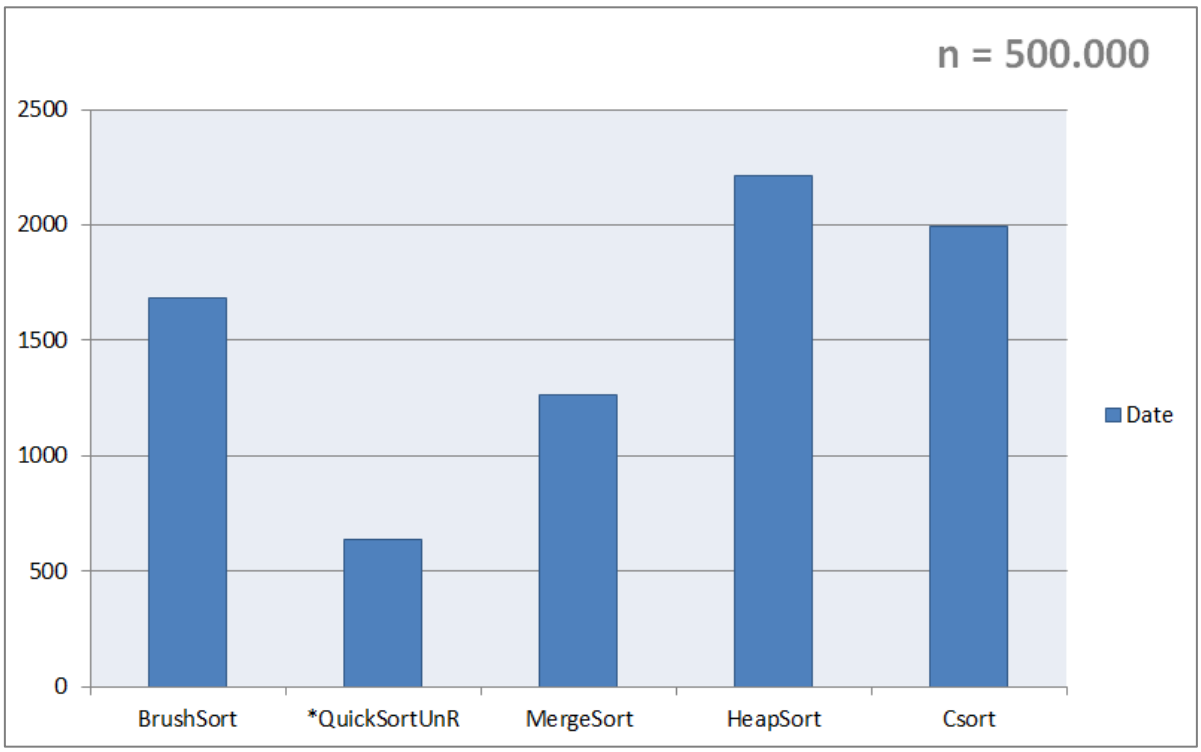
Если говорить о сортировке строк, то картина меняется:



Меньше всего времени занимает все так же метод «быстрой сортировки». Но сортировка слиянием теперь не в лидерах, а является наихудшим вариантом. Думаю, все дело в количествах запросов на сравнение двух элементов массива. В самом начале я говорила о том, что сравнение строк занимает у процессора довольно много времени. А, значит, сортировка слиянием пользуется этой итерацией чаще всего.

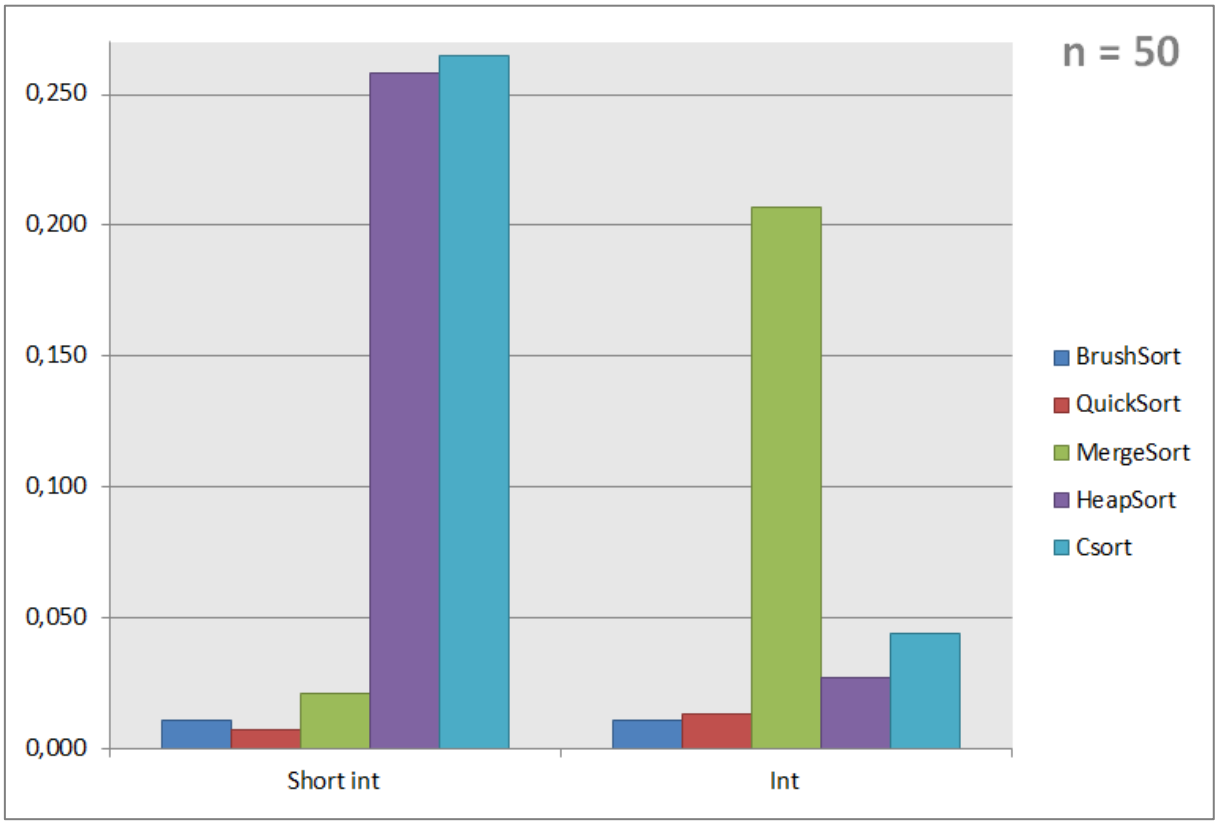
Проблема рекурсии

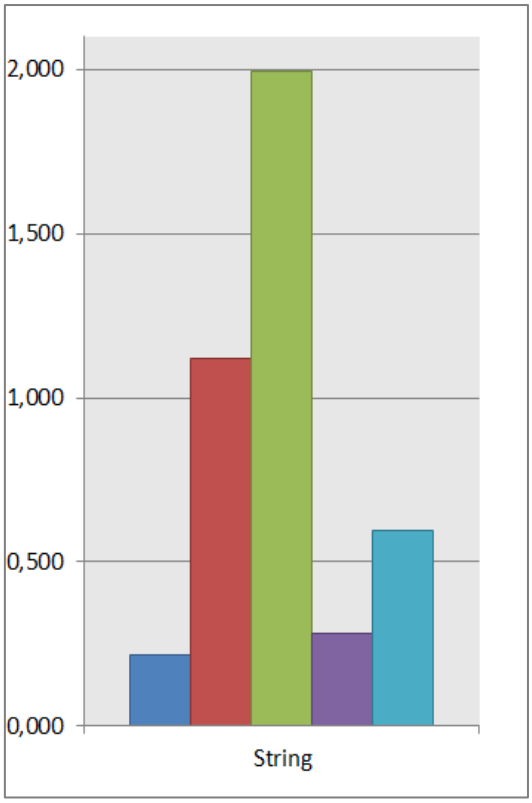
А вот на обработке дат «быстрая сортировка» дала сбой. До этого момента все рекурсивные сортировки работали бесперебойно, но на массиве длиной 500 элементов я получила ошибку: переполнение стека вызовов. Мне кажется, это связано с тем, что для того чтобы сравнить две даты, нужно обратиться к переопределению этого оператора в классе. Из-за большого количества таких вызовов и без того глубоко рекурсивный алгоритм переполняет выделенную память.
Уменьшить количество запросов к оператору сравнения я не могу, поэтому единственным выходом был не рекурсивный способ реализации алгоритма «быстрой сортировки» - QuickSortUnR.

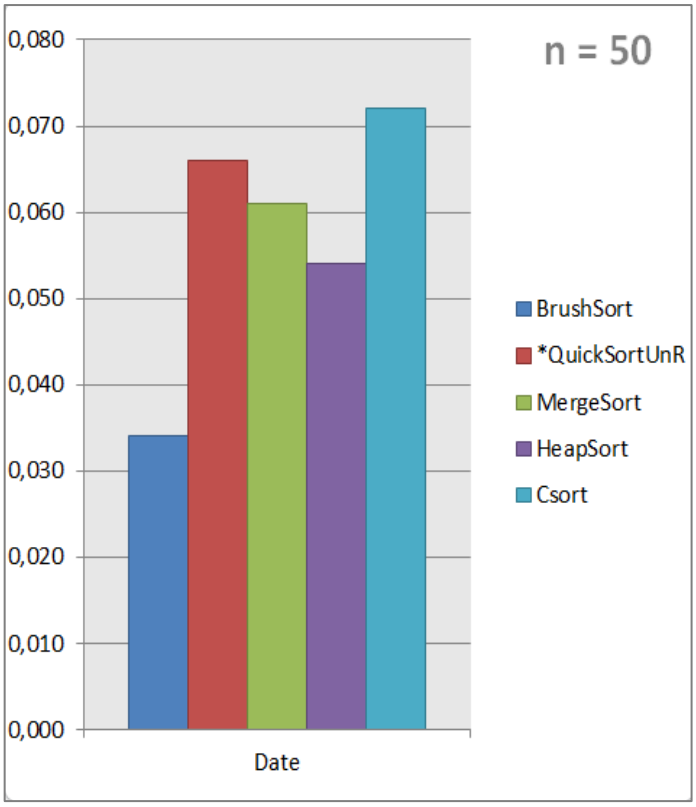


Несмотря на отсутствие рекурсии, алгоритм «быстрой сортировки» остается самым выигрышным по времени, а второе место, как и для чисел, заняла сортировка слиянием.

Быстрые сортировки (короткие массивы)

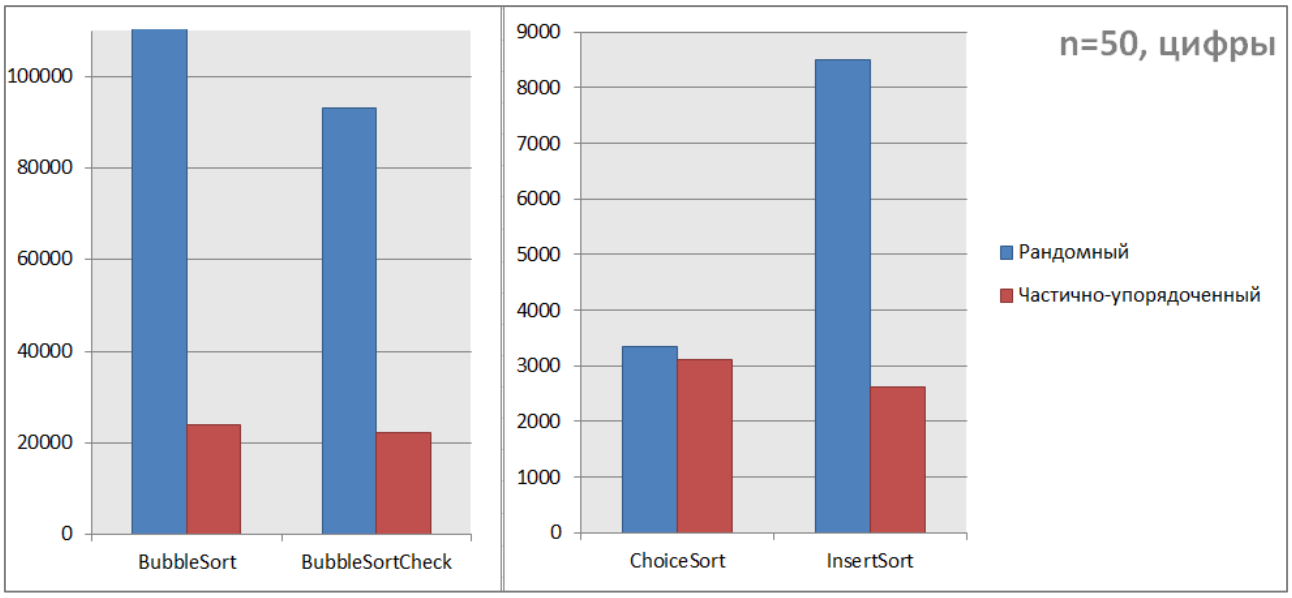






Для массивов небольшой размерности универсальным быстрым алгоритмом с наименьшим временем работы является сортировка расческой, потому что n^2 в данном случае будет сравнительно небольшим.

Частично-упорядоченные массивы

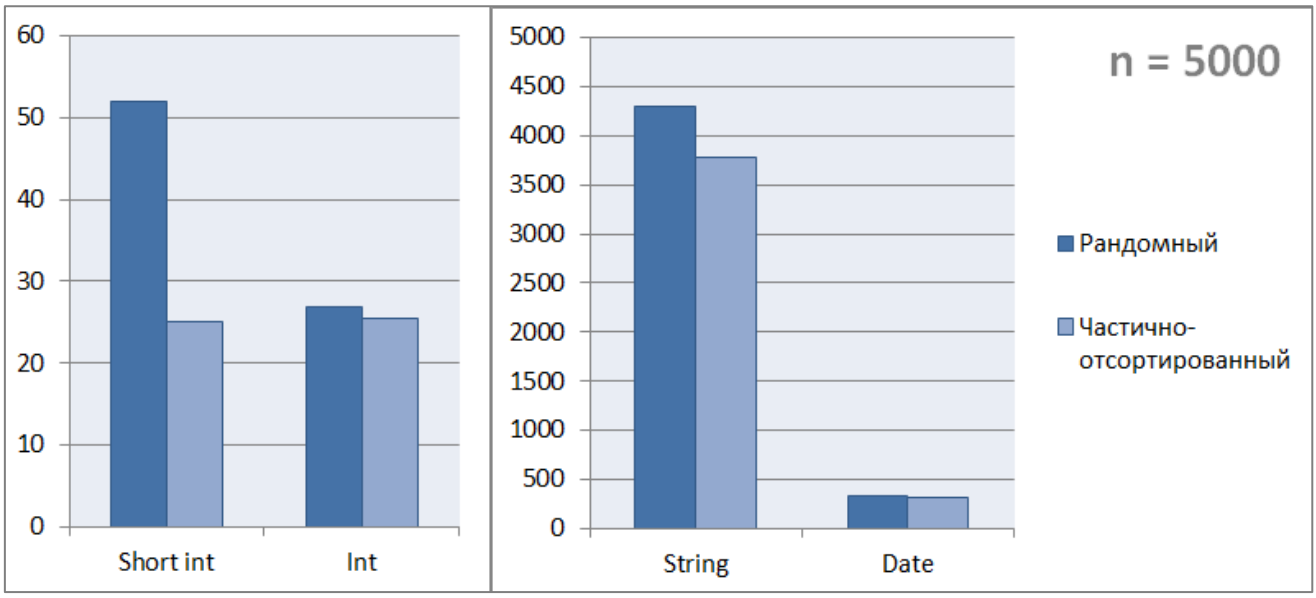


Если смотреть на результат в целом, то время работы, конечно, упало. Это характерно для любых данных, для любых размеров массивов, для любых алгоритмов. Но даже на примере этой гистограммы видно, что у одних методов сортировки время сократилось в 3 раза, а у других – совсем на незначимую величину. Поэтому есть смысл точно так же, как и для массивов случайного заполнения, рассмотреть быстрые и простые сортировки отдельно.

Простые сортировки

Как видно по гистограмме выше, время работы сортировки выбором почти не сократилось. Это вполне объясняется идеей алгоритма – в поисках минимума всегда будут рассматриваться все элементы данного шага. Частичная отсортированность алгоритму не заметна, поэтому и время работы не уменьшается.

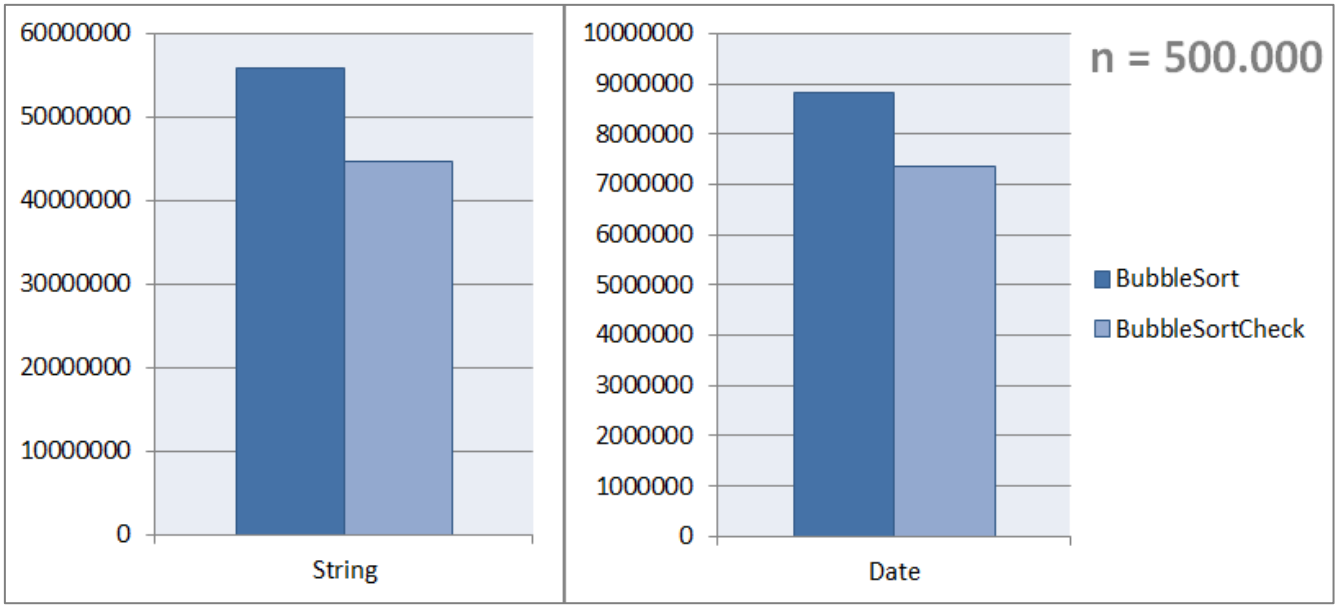
Лучшим и универсальным алгоритмом для разных размеров массива среди простых методов остается сортировка вставками, несмотря на то, что время ее работы сократилось сильнее всего на массивах, заполненных цифрами, что можно увидеть на следующей гистограмме:



Я думаю, все дело в том, что эти эксперименты я проводила над числами и строками фиксированного размера, чтобы подаваемые на вход поразрядной сортировке данные совпадали с данными для остальных сортировок. Средний размер элементов в массиве, вероятно, увеличился, поэтому эффект частичной упорядоченности заметен не так ярко.

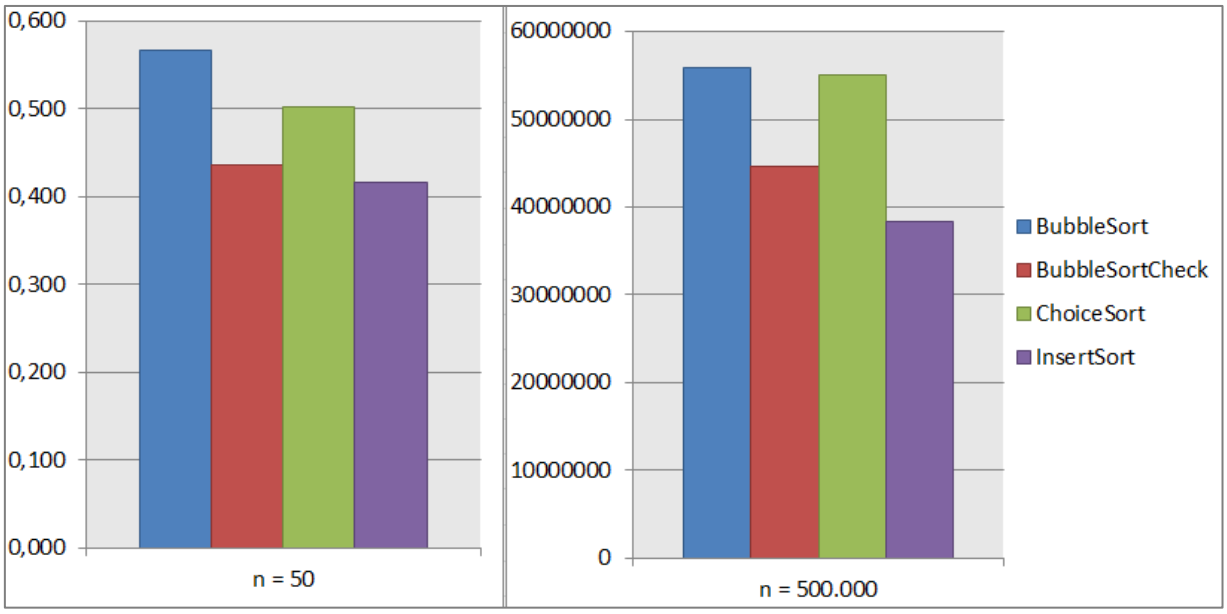
Теперь, посмотрим, появилась ли существенная разница между сортировкой пузырьком и сортировкой пузырьком с проверкой.

На массивах из цифр и чисел разница осталась весьма непримечательной. Чего нельзя сказать про данные из строк и дат. На длинных массивах преждевременная отсортированность массива может существенно выиграть время.



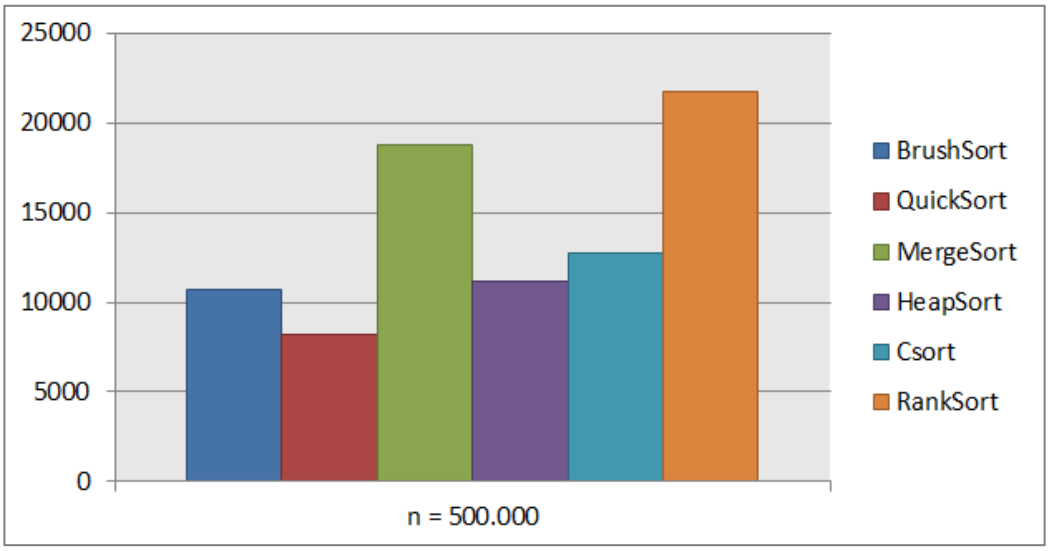
Если говорить о том, какие сортировки могут быть целесообразны для небольших массивов, то теперь сортировки пузырьком – не такой уж и плохой вариант, если брать во внимание их простую реализацию.

Более того, на больших массивах сортировка пузырьком работает столько же времени, сколько и метод с выбором. А пузырек с проверкой значительно превосходит сортировку выбором по эффективности!



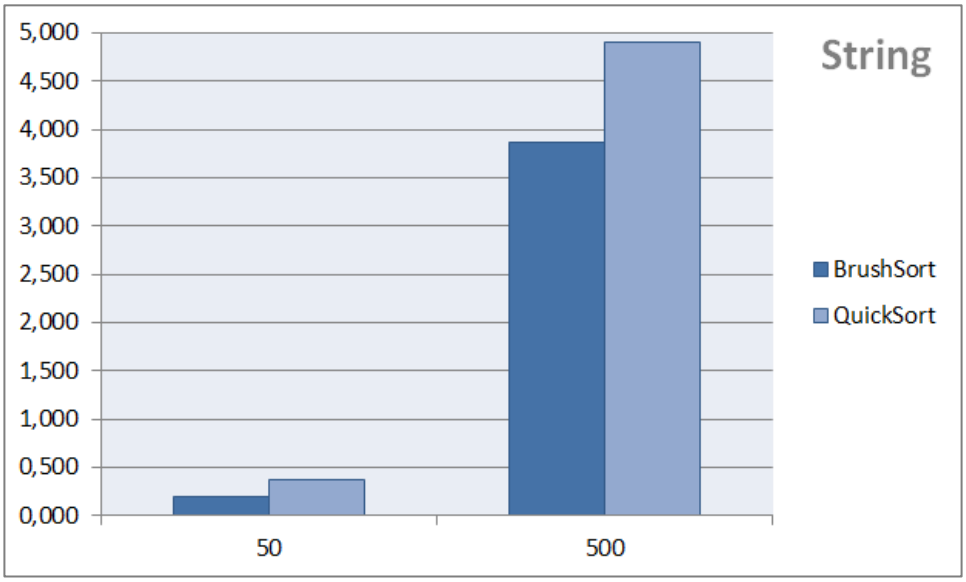
Быстрые сортировки

Худший результат вновь показывает поразрядная сортировка. Хотя при заполнении массива строками, то сортировка слиянием тоже оказывается в хвосте. Этот феномен возникал и при работе с массивами, заполненными рандомно.

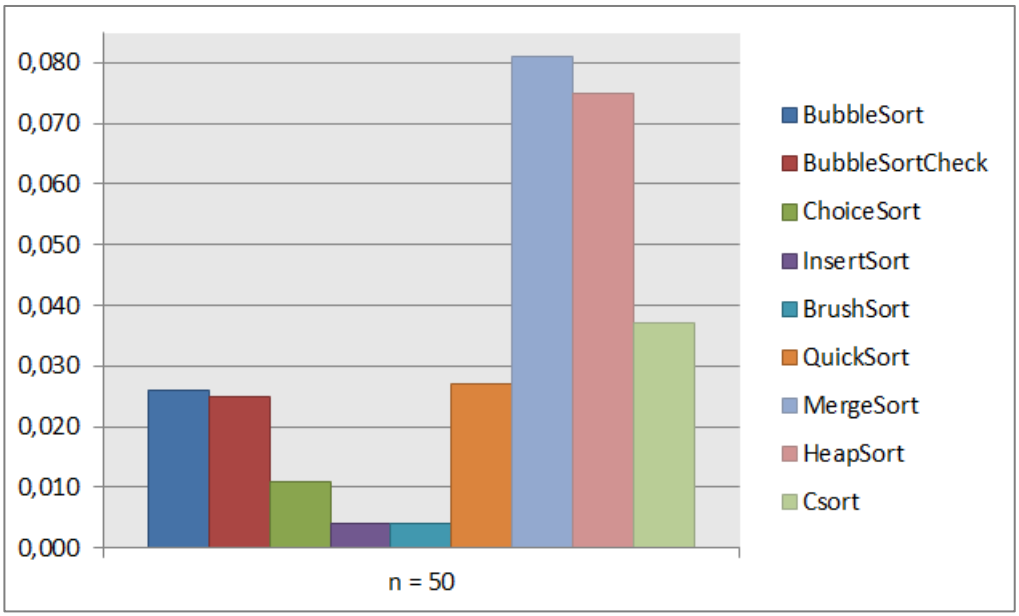


Самое маленькое время работы со списками, длина которых превышает 500 элементов любого типа, вновь засечено у алгоритма «быстрой сортировки».

Однако обработка небольших массивов в ряде случае лучше удается сортировке расческой. Этот результат был для меня ожидаемым, потому что выбранный мною способ частично отсортировать массив похож на первый шаг работы этого алгоритма – «проход толстым гребнем по списку».



Удивительная картина получается при поиске наилучшего способа отсортировать массив из 50 цифр. Впервые я могу рассмотреть все методы сортировки в совокупности. Более того, простые сортировки оказываются быстрее быстрых, что даже звучит парадоксально.



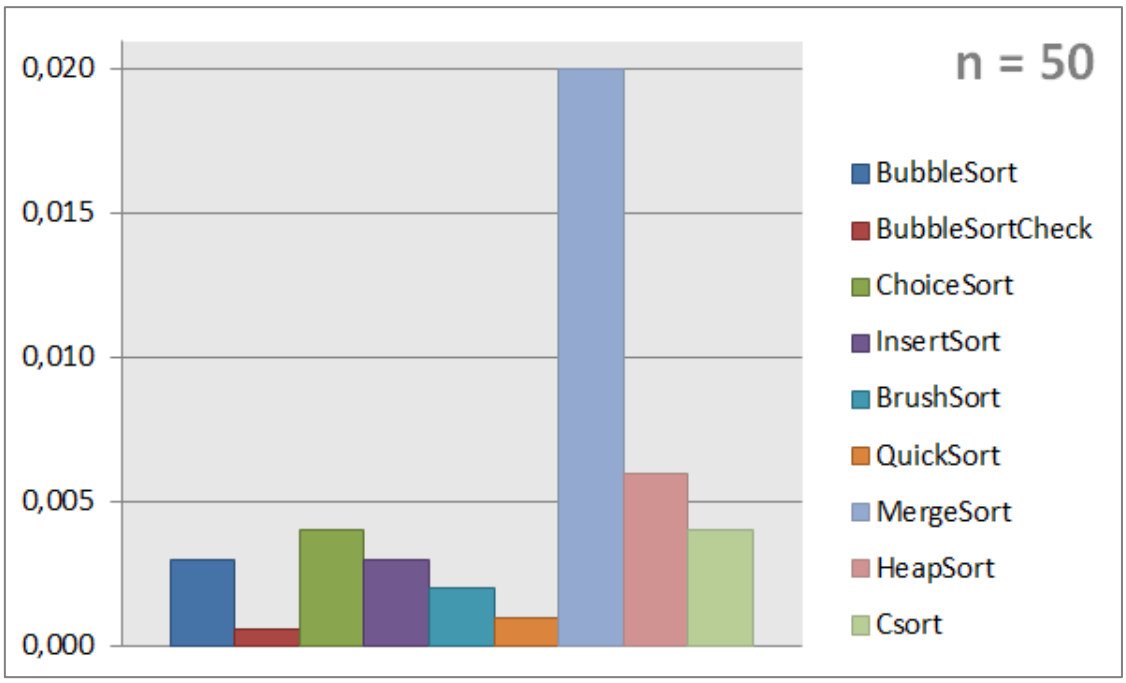
Массивы, состоящие из одинаковых элементов

Тестирование методов сортировки на таких массивах интересно тем, что время работы алгоритма будет посчитано без многочисленных преобразований вводных данных, потому что массив изначально является, в своем роде, отсортированным, и переставлять ничего в нем не надо. В этом разделе работы стоит обратить внимание на возможность или невозможность алгоритмов проверять массив на преждевременную отсортированность. Те методы, которые могут завершить обработку раньше, если она больше не требуется, наверное, будут показывать хорошие результаты.

В этом разделе удобнее и логичнее будет рассмотреть отдельно сортировки разных типов данных, без деления на быстрые и простые алгоритмы.

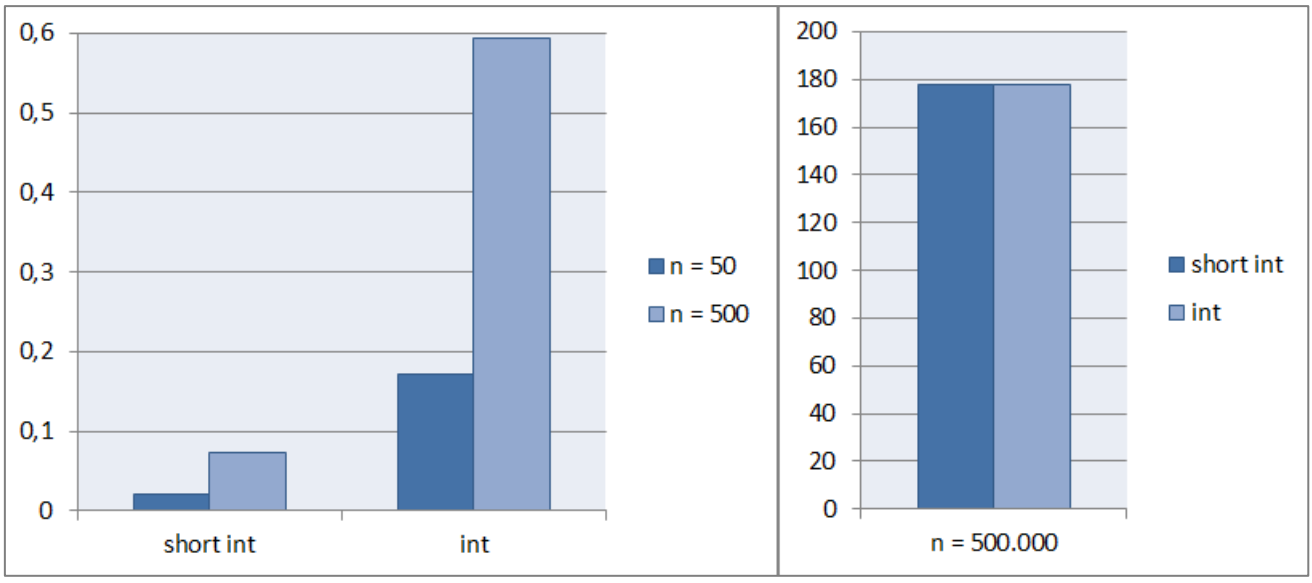
Поразрядная сортировка не имеет возможности завершить свою работу раньше, в связи с чем время ее работы остается большим. Использовать этот метод для однородно заполненных массивов совершенно не целесообразно, поэтому говорить про нее подробнее в данном разделе не буду.

Цифры и числа

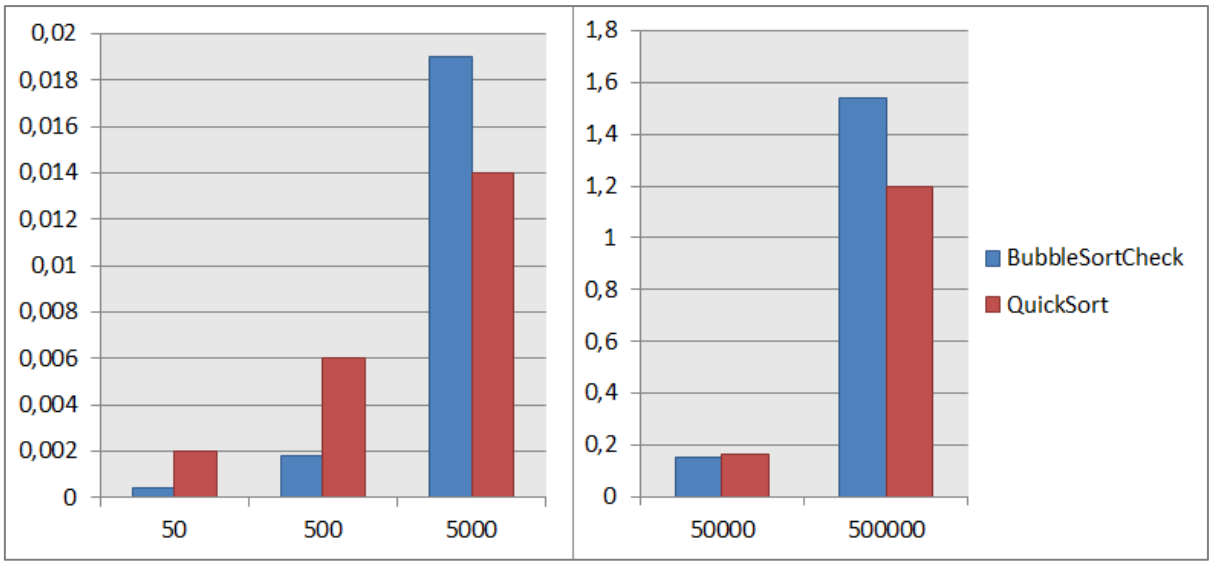


На этой гистограмме представлены результаты по сортировке небольшого массива из цифр. Неожиданностью было то, что показатели для чисел абсолютно такие же, как и для цифр, как на больших, так и на маленьких массивах, что может быть связано с отсутствием необходимости их менять местами. Откуда можно сделать вывод о том, что swap на разных типах данных тоже работает с разной скоростью.

Вопрос вызывает только сортировка слиянием, потому что на маленьких массивах время работы с цифрами и числами сильно разнится, но чем длиннее список, тем меньше заметна разница:

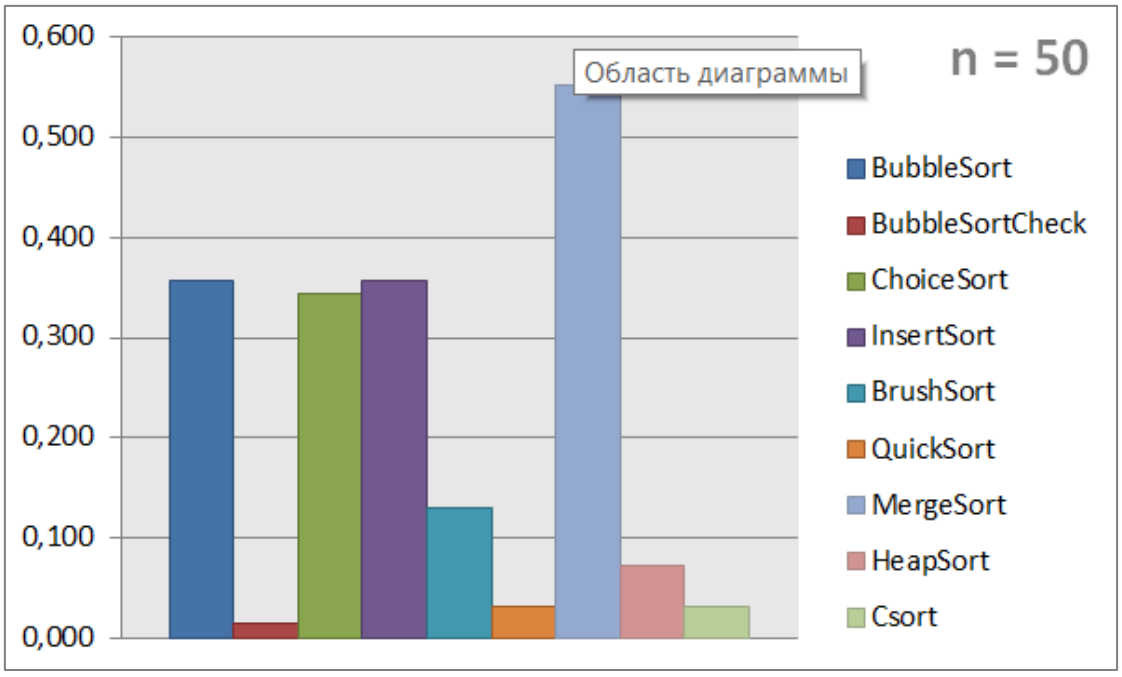


Вернусь к первой гистограмме. Для того чтобы выяснить, какой метод является наилучшим для массивов, заполненных одинаковой цифрой или одинаковым числом, надо рассмотреть отдельно преимущественных лидеров: сортировку пузырьком с проверкой и быструю сортировку.



Для начала надо заметить, что и у первого, и у второго метода присутствует необходимая проверка на наличие разных элементов и на необходимость продолжения работы алгоритма. Удивительно, но быстрая сортировка проигрывает простой сортировке на маленьких массивах. Это можно объяснить тем, что n^2 не такой большой, и алгоритму быстрее удается выяснить, что массив однороден. А быстрой сортировке для этого необходимо попытаться найти опорный элемент. Однако на больших объемах данных возрастает разница во времени между просмотром всех элементов и просмотром всех пар элементов с их сравнением.

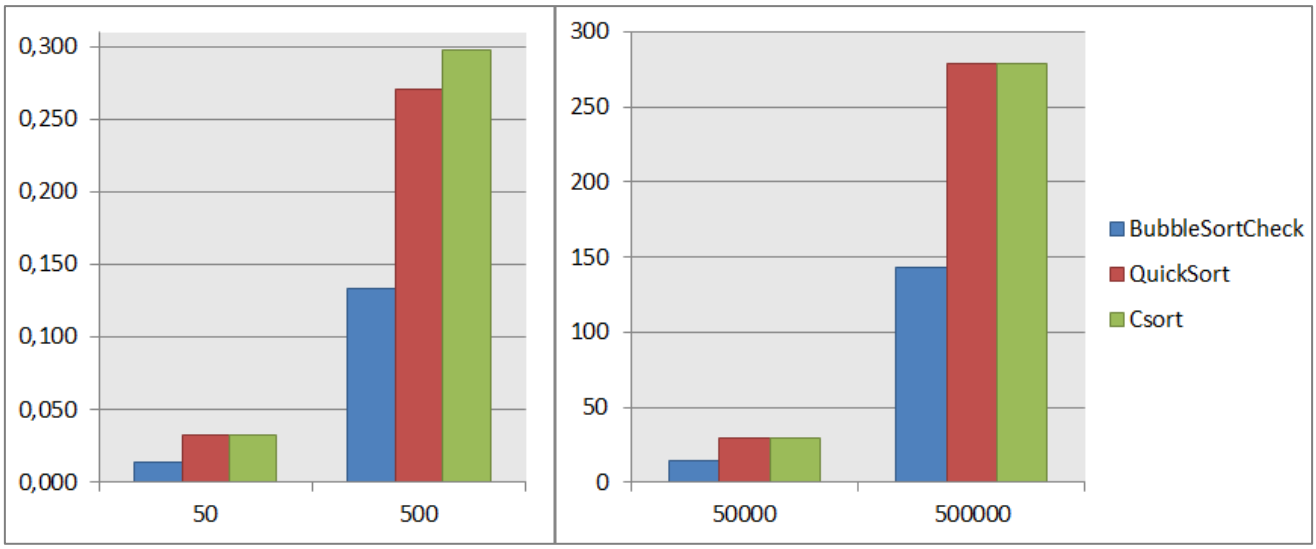
Строки



Хуже всех себя вновь проявила сортировка слиянием. Несмотря на это, ее время работы на больших массивах, безусловно, меньше чем у простых сортировок.

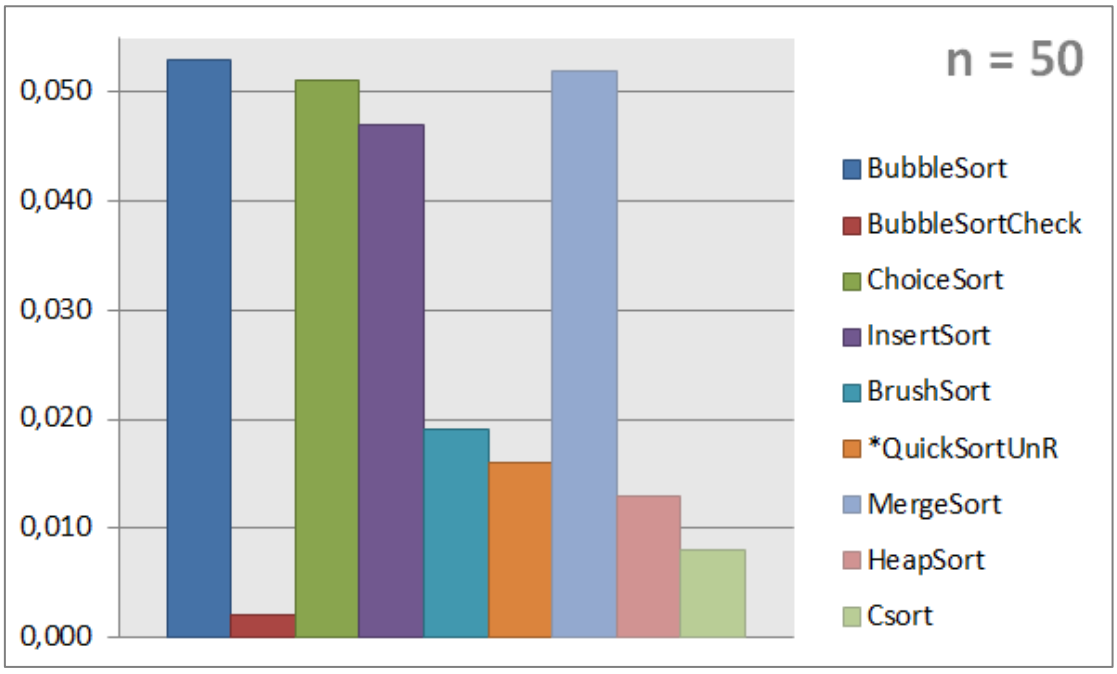
В отдельном порядке рассмотрю лидеров – к рядам эффективных методов впервые прибавилась встроенная в язык сортировка, которая до этого момента всегда занимала среднюю по скорости позицию. Это наводит на мысль о том, что в данной реализации тоже присутствует проверка на отсутствие разных элементов. Еще могу предположить, что скорость ее работы до этого была небольшой из-за необходимости работать с предоставленными по ссылке элементами, самой «подбирать» подходящую

для данного типа операцию сравнения. Если говорить про даты, то ей приходилось обращаться к переопределенной функции сравнения, это тоже могло наложить отпечаток на время работы.



Отрыв невелик, но на любых размерах массивов из строк выгоднее всего по времени использовать простую сортировку пузырьком с проверкой.

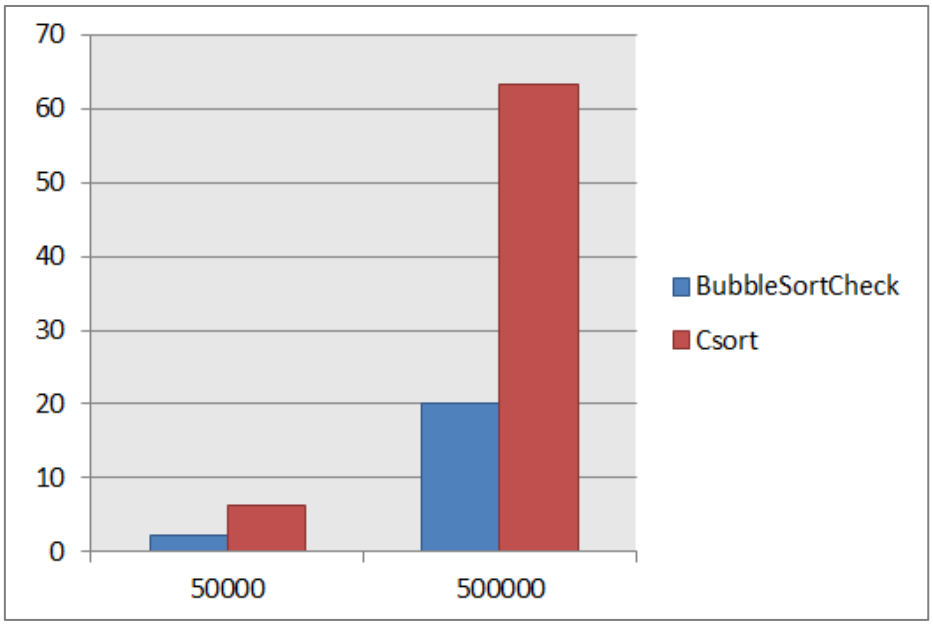
Даты



Такое же распределение эффективности характерно и для больших массивов дат. Сортировка кучей держит стабильное третье место.

Четвертой по скорости оказывается быстрая сортировка. Это доказывает то, что работа не рекурсивного варианта этого алгоритма проигрывает рекурсивному.

Если среди лучших алгоритмов выбирать наиболее подходящий не только для небольших массивов, то преимущество остается на стороне сортировки пузырьком с проверкой.



Заключение

Если подводить сухие итоги работы и попытаться собрать вместе все наиболее подходящие для разных случаев методы сортировок, то получится следующее:

1. Массив заполнен рандомно или частично упорядочен, кол-во элементов в пределах 100:
   →сортировка расческой.

2. Массив заполнен рандомно или частично упорядочен, кол-во элементов больше 500:
   →быстрая сортировка (рекурсивный метод)
   !Если для элементов массива необходим переопределенный оператор сравнения, то быстрая сортировка в ее не рекурсивном варианте.

3. Массив заполнен однородно, очень много повторяющихся элементов:
   →любая сортировка с проверкой состояния отсортированности массива на каждом шаге (сортировка пузырьком с проверкой, быстрая сортировка, встроенные сортировки).

Это всего лишь рекомендации, которые не носят принципиальный характер. Более того, необходимо брать во внимание используемый язык программирования, тип входных данных…

Однако лично для себя я сделала много выводов. Неверный выбор алгоритма может увеличить время работы программы с 30 секунд до 2х часов, поэтому если отдавать предпочтение самого короткому и простому «пузырьку», надо помнить о том, что небольшая затрата времени на написание эффективного метода может в будущем выиграть гораздо больше времени на его использовании.

Приложение

```
#include <iostream>
#include <math.h>
#include <ctime>
#include <random>
#include <string>
#include <array>
#include <chrono>
#include <algorithm>
#include <stdlib.h>
#include <cmath>
#include <list>

using namespace std;
mt19937_64 rng(time(nullptr)); //глобальный рандом
#define MAXSTACK 2048

class Timer    //класс Таймер
{…}

class Date     //класс Дата
{
private:
	int dd; int mm; int yy;
	int month[12] = { 30,28,31,30,31,30,31,31,30,31,30,31 };
public:
	Date()
	{…}
	Date ThrowRandomDate()
	{
		/*любой год с 1000 до 2500*/
		int newy = 1000 + rng() % 1500;
		/*любой месяц с 1 до 12*/
		int newm = 1 + rng() % 12;
		int newd = 1 + rng() % month[newm - 1];
		Date newdate(newd, newm, newy);
		return (newdate);
	}
	Date ThrowUnrandomDate(int a)
	{…}
	string PrintDate()
	{…}

	friend bool operator > (Date a, Date b);
	friend bool operator >= (Date a, Date b);

};
bool operator > (Date a, Date b)
{
	if (a.yy > b.yy)
		return true;
	if (a.yy < b.yy)
		return false;
	if (a.mm > b.mm)
		return true;
	if (a.mm < b.mm)
		return false;
	if (a.dd > b.dd)
		return true;
	else
		return false;
}
bool operator >= (Date a, Date b)
{…}
```

```
template <class Type>                                   //рандомное заполнение
Type* FillRandomly(Type [], size_t) {…}
template <class Type>                                  //нерандомное заполнение
Type* FillUnrandomly(Type[], size_t) {…}
template<class Type>                                     //печать
void RPrint(Type a[], size_t s) {…};
template <class Type, int size>                          //класс Массив
class Array {…};

template <class T>                                      //простой пузырек
void BubbleSort(T* ar, int n)
{
	/*создание копии для сортировки*/
	T* a = new T[n];
	for (size_t i = 0; i < n; i++)
	{
		a[i] = ar[i];
	}

	timer.start();

	for (size_t i = 1; i < n; i++)
		for (size_t j = 0; j < n - i; j++)
			if (a[j] > a[j+1])
				swap(a[j], a[j+1]);

	timer.finish();
	//RPrint(a, n);
	cout << "(size,type = " << n << typeid(a[0]).name() <<"),time of BubbleSort = " <<
timer.elapsed() << endl << endl;
}

template <class T>                                          //пузырек с проверкой
void BubbleSortCheck(T* ar, int n)
{
	/*создание копии для сортировки*/
	timer.start();
	bool made = false;
	for (size_t i = 1; i < n && !made; i++)
	{
		made = true;
		for (size_t j = 0; j < n-i; j++)
			if (a[j] > a[j+1])
			{
				swap(a[j], a[j+1]);
				made = false;
			}
	}
	timer.finish();
}

template <class T>                                                  //+выбор
void ChoiceSort(T* ar, int n)
{
	timer.start();

	for (size_t i = 0; i < n - 1; i++)
		for (size_t j = i + 1; j < n; j++)
			if (a[i] > a[j])
				swap(a[i], a[j]);
	timer.finish();

}
```

```
template <class T>                                                  //вставками
void InsertSort(T* ar, int n)
{
	T *a = new T[n];

	timer.start();

	a[0] = ar[0];
	bool push = false;
	for (size_t i = 1; i < n; i++)
	{
		push = false;
		for (size_t j = 0; j < i && !push; j++)
		/*перебор уже имеющихся в массиве эл-в*/
		{
			if (a[j] > ar[i])
			{
				for (size_t k = i; k > j; k--)
				/*двигаем все вправо*/
				{
					a[k] = a[k - 1];
				}
				a[j] = ar[i];
				push = true;
			}
		}
		if (!push)
			a[i] = ar[i];
		/*значит самое большое*/
	}

	timer.finish();
}

template <class T>                                                  //расческой
void BrushSort(T* ar, int n)
{
	/*создание копии для сортировки*/
	timer.start();

	int d = n;  //изначальный шаг
	int help = 3;
	size_t k;

	while (help > 1)
	{
		d = d / 1.247;
		if (d < 1)
			d = 1;
		k = 0;
		for (size_t i = 0; i + d < n; i++)
		{
			if (a[i] > a[i + d])
			{
				swap(a[i], a[i + d]);
				k = i;    //запоминаем последний эл-т, который меняли
			}
		}
		if (d == 1)
			help = k + 1;  //если k было 0, то менять нечего и цикл не запустится
	}

	timer.finish();
}
```

```
template <class T>
void RQuickSort(T* a, int beg, int end)                          //быстрая
{
	if (end - beg >= 1)
	{
		/*ищем опорный эл-т*/
		T pivot = a[beg];
		int pivoti = beg;
		bool b = false;
		for (int i = beg + 1; i <= end; i++)
		{
			if (a[i] > pivot)
			{
				pivoti = i;
				pivot = a[i];
				b = true;
				break;
			}
			if (pivot > a[i])
			{
				b = true;
				break;
			}
		}
		if (!b)
		{
			return;
		}

		/*раскидываем элементы по отношению к опорному*/
		int l = beg;
		int r = end;
		do
		{
			while (pivot > a[l])
				l++;
			while (a[r] >= pivot)
				r--;
			if (l < r)
				swap(a[l], a[r]);
		} while (l < r);

		RQuickSort(a, beg, l-1);
		RQuickSort(a, l, end);

	}
}
template <class T>
void QuickSort(T* ar, int n)
{
	/*создание копии для сортировки*/
	T* a = new T[n];
	for (size_t i = 0; i < n; i++)
	{
		a[i] = ar[i];
	}

	timer.start();
	RQuickSort(a, 0, n-1);

	timer.finish();

}
template <class T>
```

```
void QuickSortUnR(T* ar, long n)                              //быстрая без рекурсии
{
	/*создание копии для сортировки*/
	T* a = new T[n];
	for (size_t i = 0; i < n; i++)
	{
		a[i] = ar[i];
	}
	timer.start();

	long size = n;
	long i, j;    // указатели, участвующие в разделении
	long lb, ub;  // границы сортируемого в цикле фрагмента
	long lbstack[MAXSTACK];
	long ubstack[MAXSTACK]; // стек запросов (каждый запрос = пара значений)


	long stackpos = 1;  // текущая позиция стека
	long ppos;          // середина массива
	T pivot;          // опорный элемент
	T temp;

	lbstack[1] = 0;
	ubstack[1] = size - 1;

	do {
		// Взять границы lb и ub текущего массива из стека.
		lb = lbstack[stackpos];
		ub = ubstack[stackpos];
		stackpos--;
		do {
			// Шаг 1. Разделение по элементу pivot
			ppos = (lb + ub) >> 1;
			i = lb; j = ub; pivot = a[ppos];

			do {
				while (pivot > a[i]) i++;
				while (a[j] > pivot) j--;

				if (i <= j) {
					temp = a[i]; a[i] = a[j]; a[j] = temp;
					i++; j--;
				}
			} while (i <= j);

			if (i < ppos) { // правая часть больше

				if (i < ub) {  //если в ней больше 1 элемента - нужно
					stackpos++;   //сортировать, запрос в стек
					lbstack[stackpos] = i;
					ubstack[stackpos] = ub;
				}
				ub = j;

			}
			else {            // левая часть больше
				if (j > lb) {
					stackpos++;
					lbstack[stackpos] = lb;
					ubstack[stackpos] = j;
				}
				lb = i;
			}
```

```
            } while (lb < ub);  // пока в меньшей части более 1 элемента

        } while (stackpos != 0)// пока есть запросы в стеке

        timer.finish();
}

template <class T>
void RMergeSort(T* a, int beg, int end)                                 //слиянием
{
        //!end - индекс реально последнего, а не кол-во эл-тов
        if (beg == end) //1 эл-т
                return;
        int mid = ((beg + end) / 2);
        RMergeSort(a, beg, mid);
        RMergeSort(a, mid + 1, end);

        //cout << beg << "    " << mid << "   " << end << endl;
        int ind1 = beg;
        int ind2 = mid + 1;
        T *help = new T [end-beg+1];  //промежуточ резы здесь будут
        for (size_t k = 0; k <= end - beg; k++)
        {
                if ((ind2 > end) ||((a[ind2] > a[ind1]) && (ind1 <= mid)))
                {
                        help[k] = a[ind1];
                        ind1++;
                }
                else
                {
                        help[k] = a[ind2];
                        ind2++;
                }
        }

        for (size_t k = 0; k <= end - beg; k++)
                a[beg + k] = help[k];
}
template <class T>
void MergeSort(T* ar, int n)
{
        /*создание копии для сортировки*/

        timer.start();

        RMergeSort(a, 0, n - 1);

        timer.finish();
}

template <class T>
void HeapPush(T* a, int j, int l)                                       //кучей
{
        size_t maxi = j;
        size_t left = j * 2 + 1;
        size_t right = j * 2 + 2;

        if (left <= l)  //если есть потомки вообще
        {
                if (a[left] > a[j])
                        maxi = left;
                if ((right <= l) && a[right] > a[maxi])
                        maxi = right;
                if (maxi != j)
```

```
		{
			swap(a[j], a[maxi]);
			HeapPush(a, maxi, l);
		}
	}

}

template <class T>
void RHeapSort(T* a, int n)
{
	/*построение дерева*/
	for (int i = (n / 2) - 1; i >= 0; i--)
		HeapPush(a, i, n - 1);

	/*max в конец*/
	for (size_t i = n - 1; i > 0; i--)
	{
		swap(a[0], a[i]);
		HeapPush(a, 0, i - 1);
	}
}
template <class T>
void HeapSort(T* ar, int n)
{
	/*создание копии для сортировки*/

	timer.start();

	RHeapSort(a,n);

	timer.finish();

}

template <class T>                                                    //поразрядная
void RankSort(T* ar, int n) {};
template <>
void RankSort(int* ar, int n)
{
	/*создание копии для сортировки*/

	timer.start();

	int k = 8;
	if (a[0] < 10)
		k = 1;
	list<int> pocket [10];
	int ostk, delk; //доп коэф
	int num;
	for (size_t i = k; i > 0; i--)
	{
		ostk = pow(10, k - i + 1);
		delk = pow(10, k - i);
		//распихиваем по карманам
		for (size_t j = 0; j < n; j++)
		{
			num = (a[j] % ostk) / delk;
			pocket[num].push_back(a[j]);
		}
		int listi = 0;
		int j = 0;
		//соед карманы
		while(j < n && listi<10)
```

```
		{
			while (!pocket[listi].empty())
			{
				a[j] = pocket[listi].front();
				pocket[listi].pop_front();
				j++;
			}
			listi++;
		}
	}

	timer.finish();
}
template <>
void RankSort(string* ar, int n)
{
	/*создание копии для сортировки*/

	timer.start();

	int k = 6;
	list <string> pocket[26];
	int num;
	for (size_t i = k; i > 0; i--)
	{
		/*распихиваем по карманам*/
		for (size_t j = 0; j < n; j++)
		{
			num = int((a[j])[i-1]) - 97;
			pocket[num].push_back(a[j]);
		}
		int listi = 0;
		int j = 0;
		/*соед карманы*/
		while (j < n && listi < 26)
		{
			while (!pocket[listi].empty())
			{
				a[j] = pocket[listi].front();
				pocket[listi].pop_front();
				j++;
			}
			listi++;
		}
	}

	timer.finish();
}

template <class T>                                                    //встроенная
void CSort(T* ar, int n)
{
	/*создание копии для сортировки*/
	timer.start();

	sort(a,a+n);  //не даты
	//sort(a, a+n, [](const Date& a, const Date& b) { return b > a; });

	timer.finish();
}
```