Бардуков А.А.

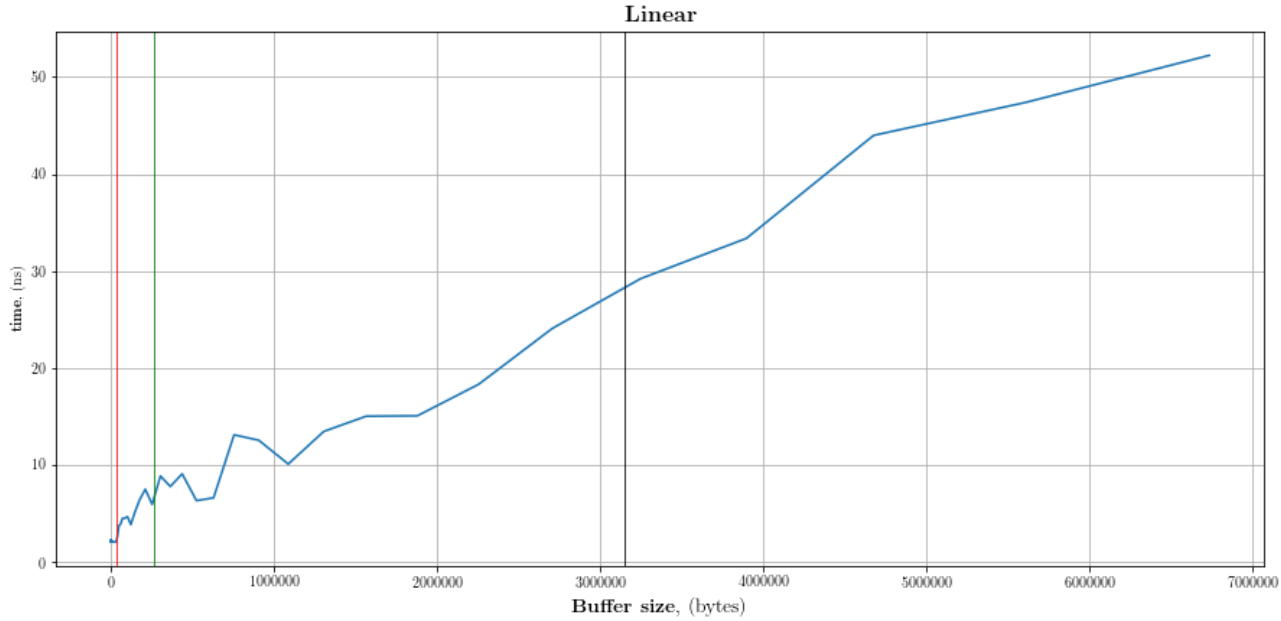
Практическое задание 3

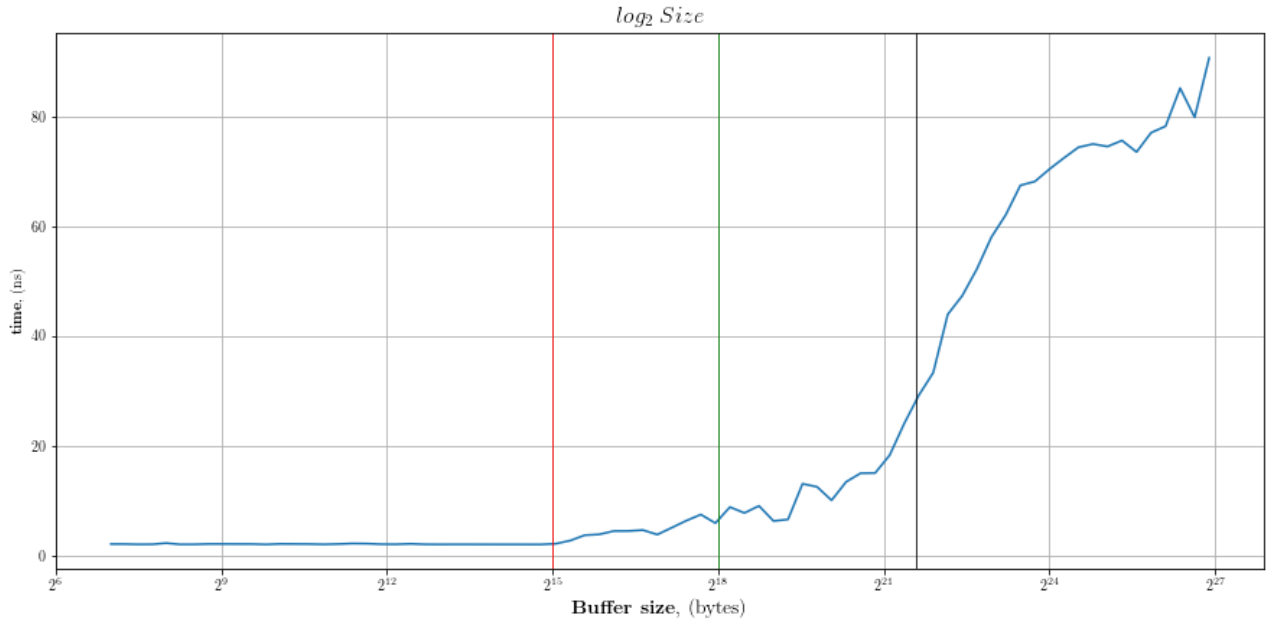
Задание 1.

Процессор:

```
Model name:            Intel(R) Core(TM) i5-4210U CPU @
1.70GHz
CPU max MHz:           2700,0000
CPU min MHz:           800,0000
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              3072K
```

График обращений к кэшам:





Времена обращения:

```
L1 cache:      2   ns
L2 cache:      4.7 ns
L3 cache:      12  ns
RAM:           25+ ns
```

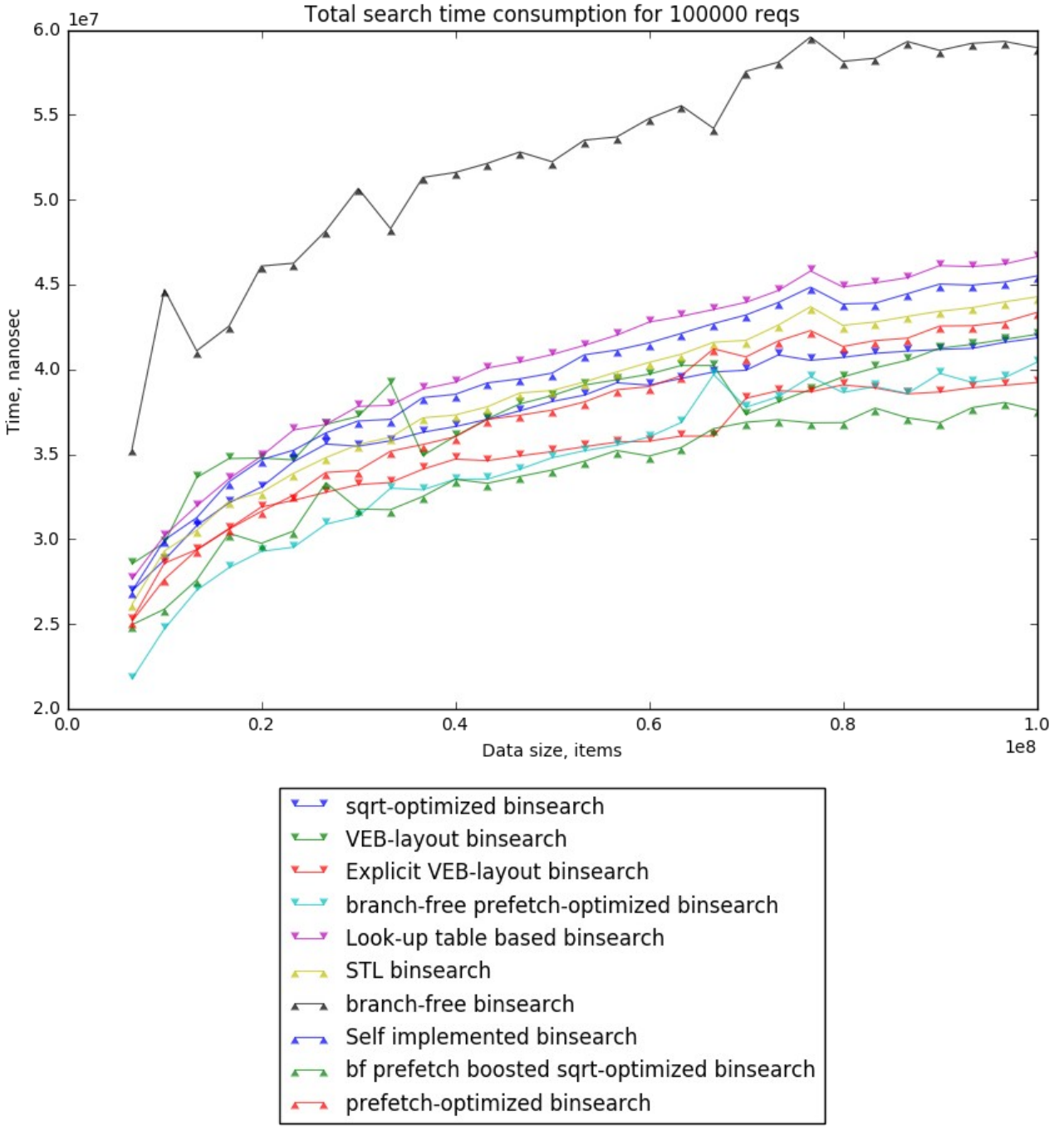
Мне не удалось найти в спецификации подробной информации о времени доступа к различным кэшам, зато я случайно нашёл аналогичную утилиту в репозитории Линуса Торвальдса (https://github.com/torvalds/test-tlb), оставив только рандомный доступ к участкам памяти, я получил следующий вывод:

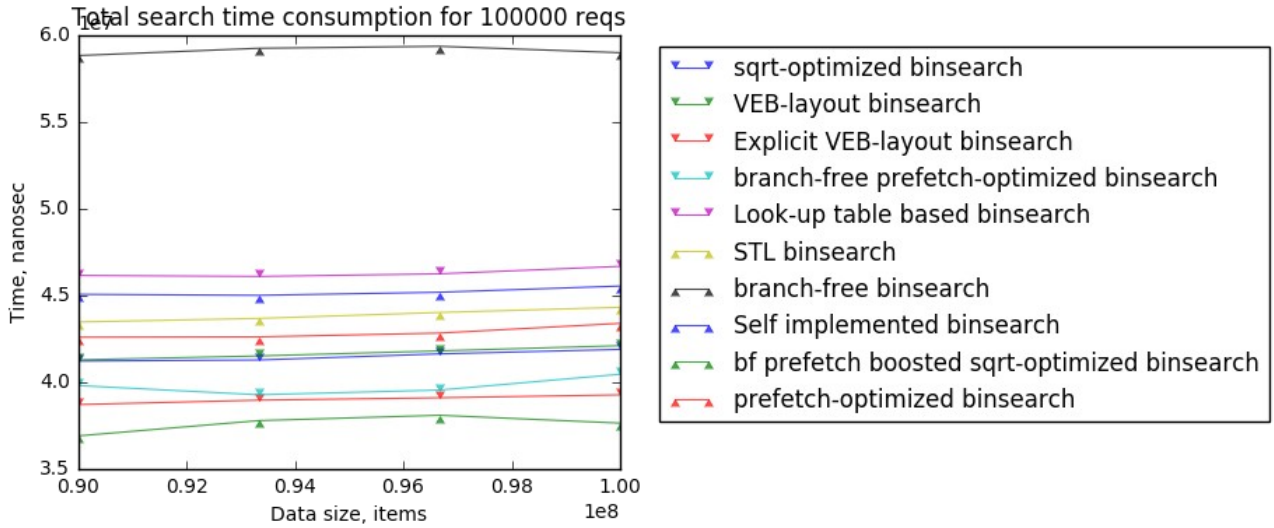
```
4k:
  1.89ns (~4.5 cycles)
8k:
  1.89ns (~4.5 cycles)
16k:
  1.90ns (~4.6 cycles)
32k:
  1.91ns (~4.6 cycles)
64k:
  4.57ns (~11.0 cycles)
128k:
  4.57ns (~11.0 cycles)
256k:
  7.89ns (~18.9 cycles)
512k:
 14.54ns (~34.9 cycles)
1M:
 15.48ns (~37.1 cycles)
2M:
 25.04ns (~60.1 cycles)
4M:
 51.57ns (~123.8 cycles)
6M:
 67.95ns (~163.1 cycles)
8M:
 73.22ns (~175.7 cycles)
16M:
 83.00ns (~199.2 cycles)
32M:
 83.34ns (~200.0 cycles)
64M:
 86.29ns (~207.1 cycles)
128M:
 90.07ns (~216.2 cycles)
256M:
 94.00ns (~225.6 cycles)
```

Средние значения примерно равны в выводах двух программ.
В исходниках добавлен iPython Notebook для получения графиков.

Задание 2.

Общее время на 100000 запросов:





Реализованы implicit и explicit версии VEB layout.
VEB строится рекурсивно: для одного объекта это он сам, для дерева отрезается часть (h / 2) и на нём проводится рекурсивная организация veb-layout, оставшаяся часть (bottom) подвешивается как поддеревья к элементам верхнего (top) дерева.
Implicit версия поддерживает вспомогательные массивы для формулы навигации (в зависимости от глубины).
В Explicit версии узлы хранят указатели на дочерние элементы (слева и справа).

Также добавлен LUT поиск (look-up table), где изначально строятся “фолды” по префиксу объектов (N bit), которые задают границы для поиска объекта.

Добавлены branch-free версии для обычного и prefetch поисков. Оптимизация состоит в том, что минимизируется ошибка предсказания ветки в if (за счёт тернарного оператора и сдвига только одной переменной (поиск происходит через точку начала и длину)). BF версия prefetch показывает себя лучше большинства алгоритмов.
Для эксперимента я модифицировал *sqrt-оптимизированный* поиск для использования *branch-free prefetch* бин. поиска после попадания в “фолд”. Такой подход показал себя очень хорошо, этот алгоритм выигрывает даже у explicit VEB. Вероятно, это можно объяснить, что он использует преимущества sqrt (помещение выборки в кэш) на первом этапе, а на втором помимо этого и branch-free оптимизации подгружает в кэш более высокого уровня следующий mid-элемент.

Время работы sqrt-бинпоиска лучше других на больших объёмах данных, так как его вспомогательный массив помещается в кэш, когда все другие поиски обращаются к RAM, ра счёт чего получается выигрыш в производительности.

Prefetch лучше справляется на средних объёмах, так как заранее компенсирует обращение к RAM/более высокоуровневым кэшам обычного бинпоиска.

На малых объёмах стандартный бинпоиск работатет лучше всего, так как большинство оптимизаций рассчитано на большие объёмы данных (малые тут 10, 100, 1000 элементов).

Задание 3.

Имплементированы *bjkst, loglog, hyperloglog*.

Оценки сложности

BJKST:

m – размер пространства элементов

Память:

$$O(\log m + 1/\epsilon^2 \cdot (\log(1/\epsilon) + \log\log m))$$

Добавление элемента:

*O(log m + log (*1/ϵ*)) [amortized]*

LogLog:

Память:

m loglog (n / m) (1 + o(1))

Добавление элемента:

O(1)

Подсчёт:

O(m)

m – число регистров

HyperLogLog:

Память:

$$O(\epsilon^{-2} \log\log n + \log n)$$

Добавление элемента:

O(1)

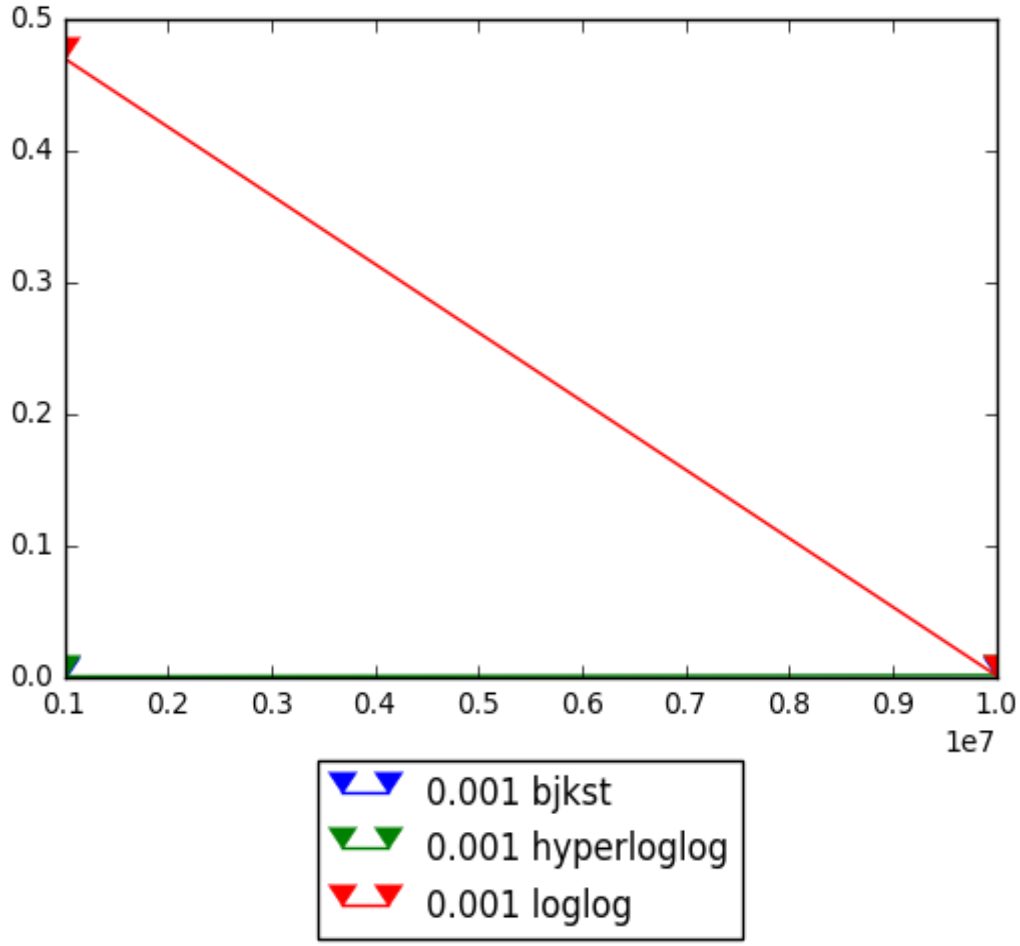
(с учётом того, что размер вывода хэш-функции фиксирован)

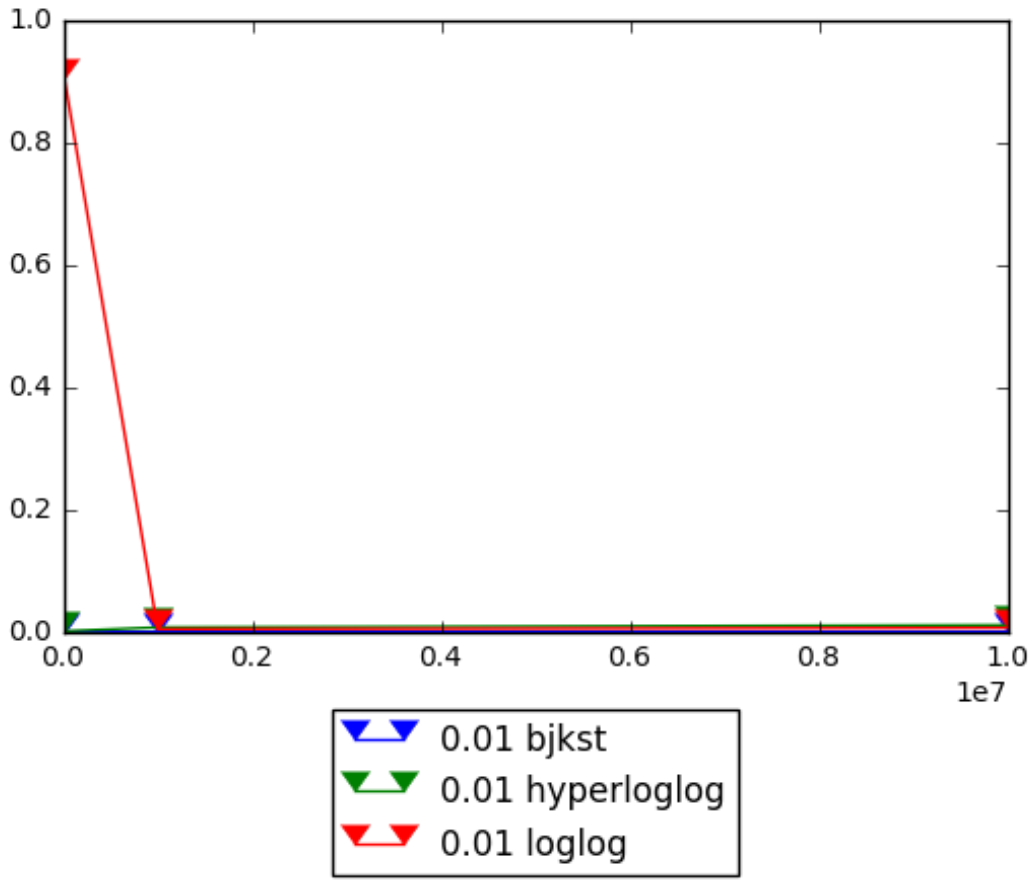
Подсчёт:

O(m)

m – число регистров

Точность:





Видно, что loglog работает плохо при точности (0.01) и малом количестве объектов, однако быстро улучшает значение с ростом количества объектов.

BJKST не ошибается на небольших значениях количества объектов.

HyperLogLog хорошо показывает себя при всех параметрах, но видно, что ошибка растёт с ростом количества объектов.

Использованные хэш-функции: *murmurhash3*, *jenkins* hash.

В случае использования последнего в силу его детерменированности ошибка колеблется мало, но достаточно велика для loglog алгоритма.

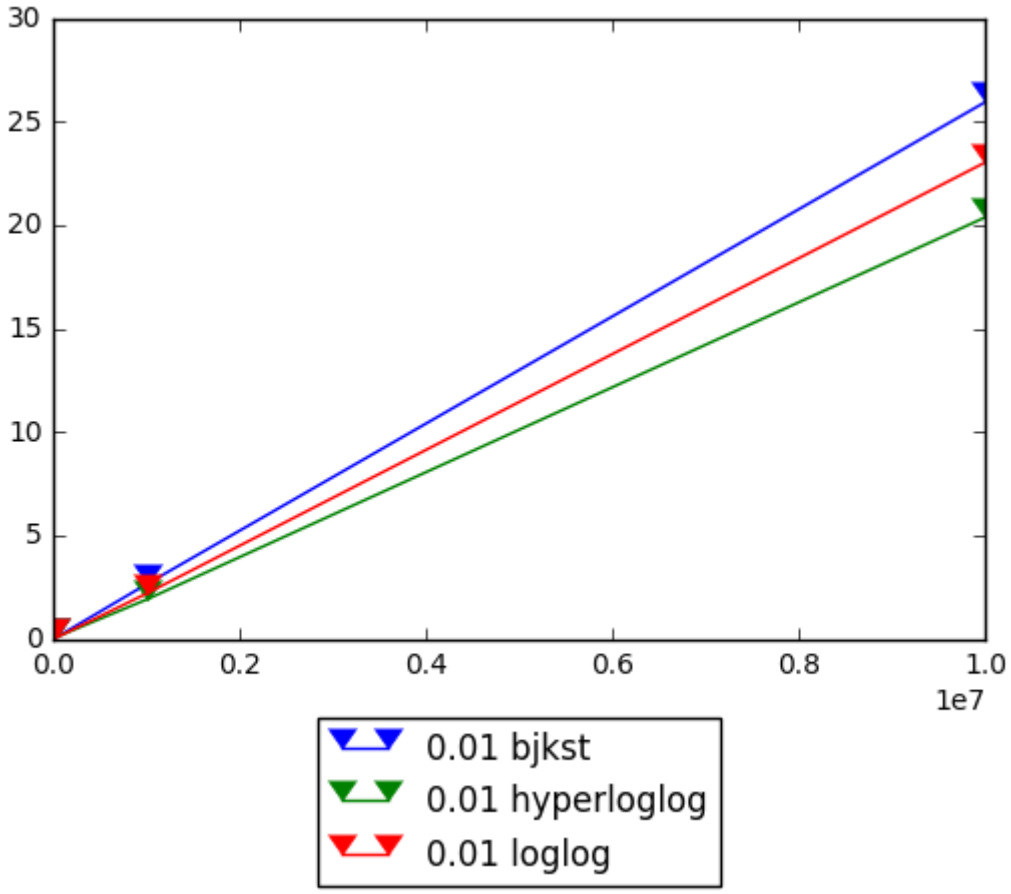
В случае использования murmurhash3 ошибка loglog варьировалась разительно для разного количества различных элементов.

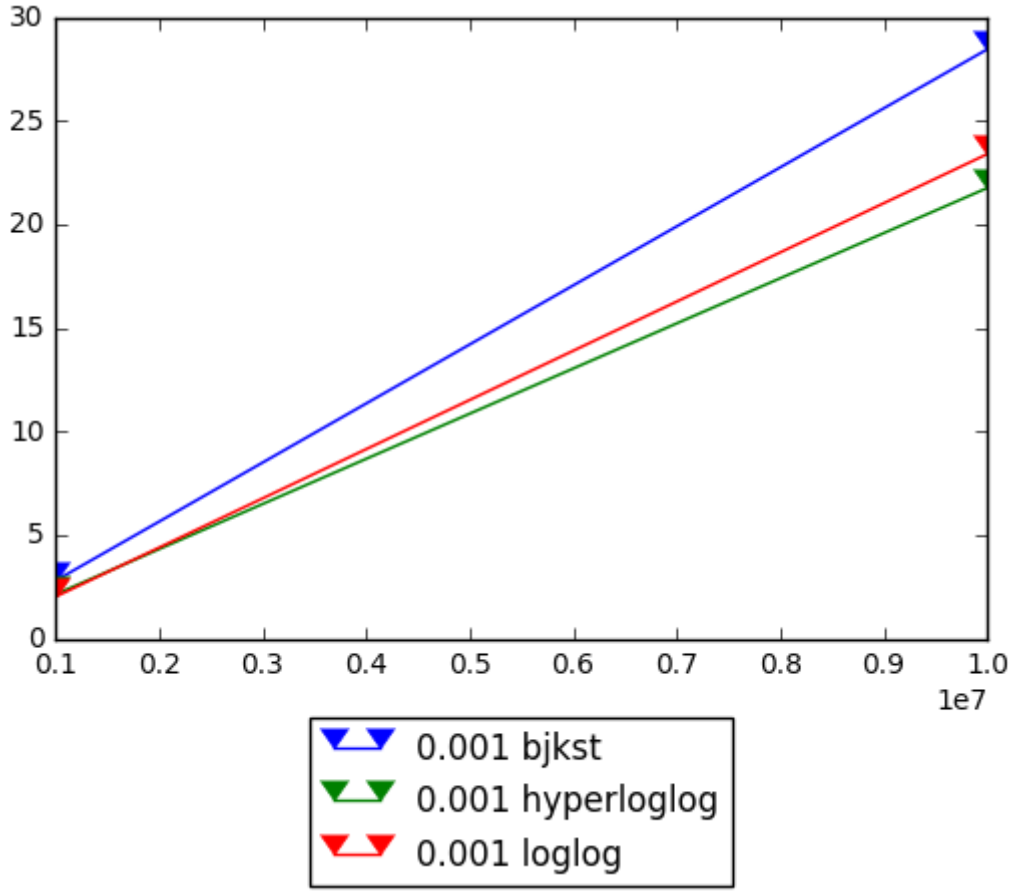
В силу скорости работы алгоритма murmurhash3 он даёт самые быстрые результаты (можно также попробовать использовать CityHash128, который должен быть ещё быстрее).
Сравнение скоростей хэшей можно увидеть в следующей таблице (датасет размера 216553)

Hash	Lowercase	Random UUID	Numbers
Murmur	145 ns	259 ns	92 ns
	6 collis	5 collis	0 collis
FNV-1a	152 ns	504 ns	86 ns
	4 collis	4 collis	0 collis
FNV-1	184 ns	730 ns	92 ns
	1 collis	5 collis	0 collis▪
DBJ2a	158 ns	443 ns	91 ns
	5 collis	6 collis	0 collis▪▪▪
DJB2	156 ns	437 ns	93 ns
	7 collis	6 collis	0 collis▪▪▪
SDBM	148 ns	484 ns	90 ns
	4 collis	6 collis	0 collis**
SuperFastHash	164 ns	344 ns	118 ns
	85 collis	4 collis	18742 collis
CRC32	250 ns	946 ns	130 ns
	2 collis	0 collis	0 collis

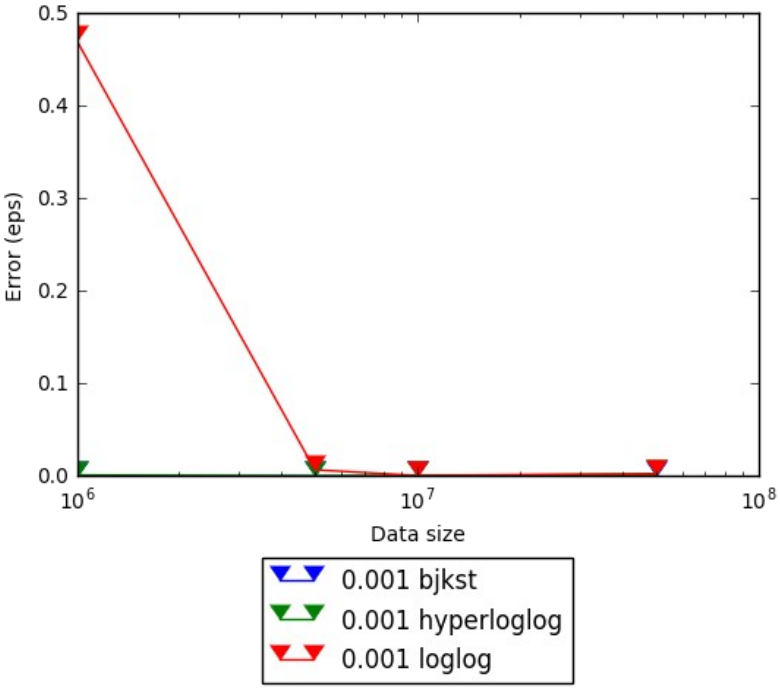
Время работы:

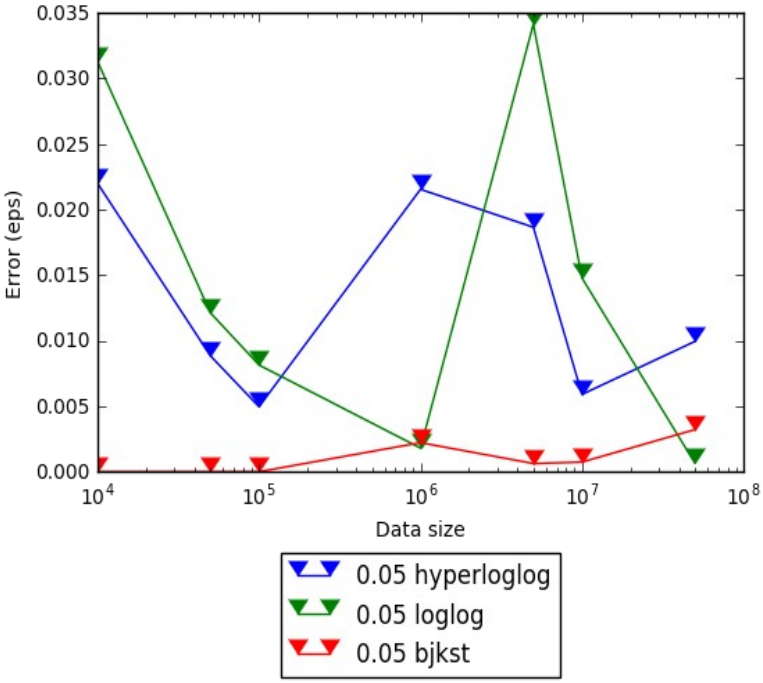
Наибольшее время работы при одинаковых хэш-функциях у меня показал *bjkst* на больших объёмах данных. (ось Y – секунды, X – количество объектов)

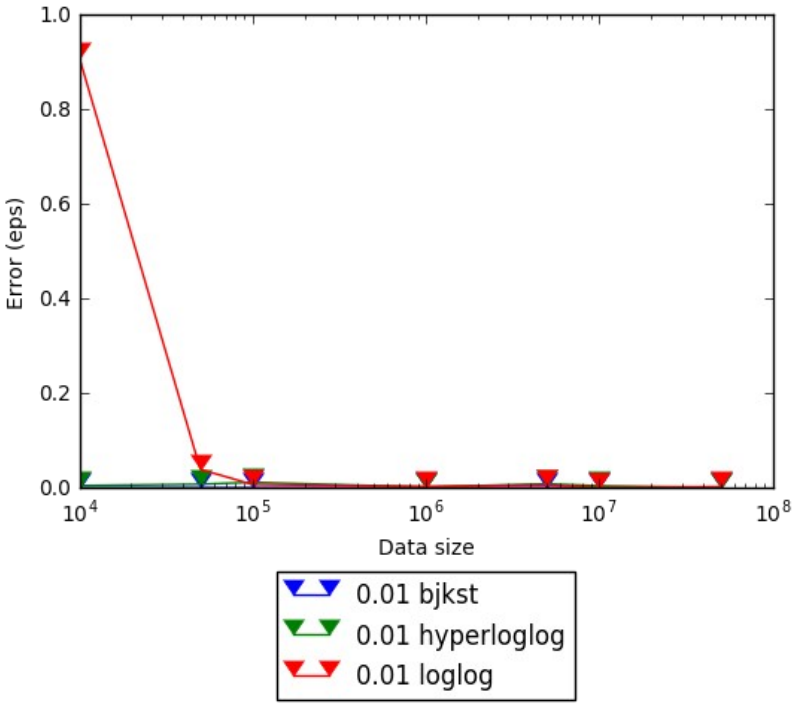




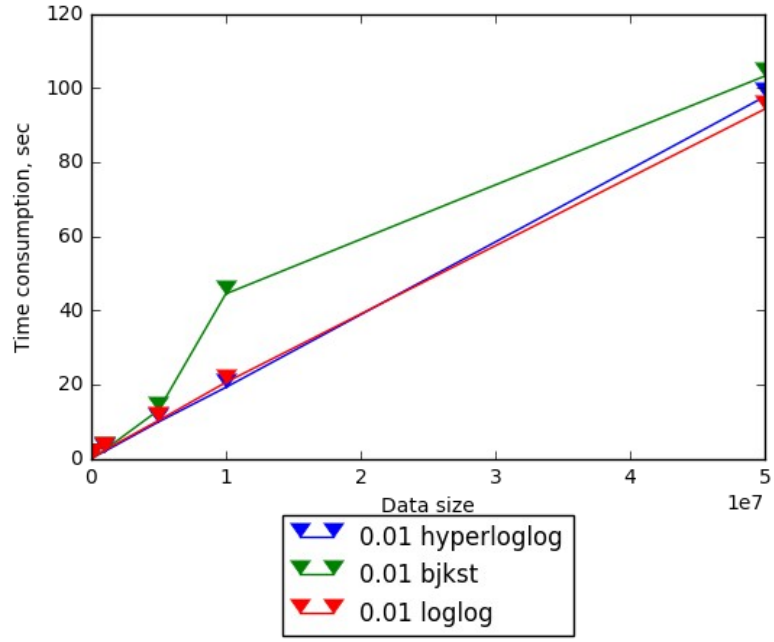
Также я добавил *median trick* для минимизации ошибки (реакция на параметр errorProbability (delta)), который уменьшает выдаваемую ошибку в $O(e^{log(1/d)})$ раз за счёт увеличения количества счётчиков (и кратного роста использования памяти), а затем из K счётчиков берётся медианный, тем самым и уменьшается погрешность, результаты работы при delta=0.01:

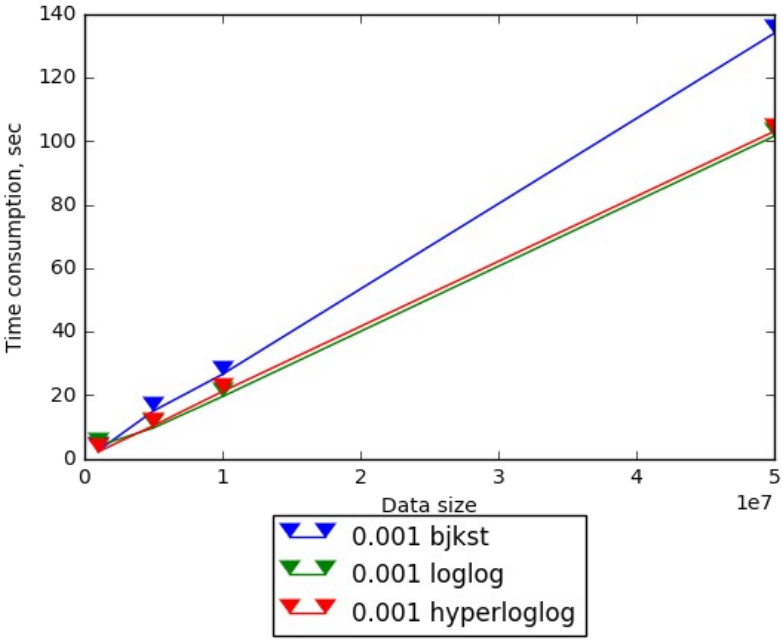


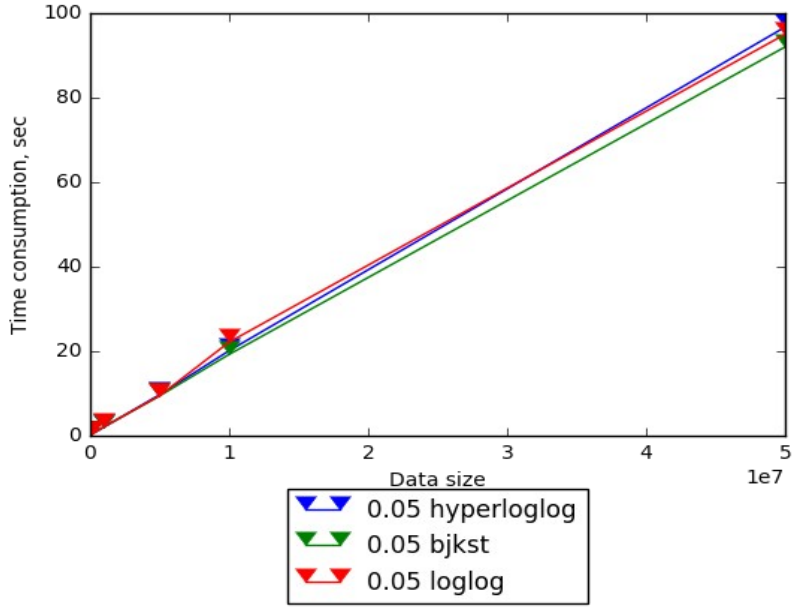




Время с использованием *median trick* (виден кратный рост по времени):







Также наблюдается кратный рост использования памяти:

без увеличения количества счётчиков: *628384*

с увеличением количества счётчиков: *2926796*