ИССЛЕДОВАНИЕ ХЭШ-ФУНКЦИЙ

ХЭШ-ФУНКЦИЯ 1

```
unsigned int hash_calc1(HashData data) {
    return 1;
}
```

Всегда возвращает 1

ХЭШ-ФУНКЦИЯ 2

```
unsigned int hash_calc2(HashData data) {
    return strlen(data);
}
```

Возвращает длину строки

```
unsigned int hash_calc3(HashData data) {
    unsigned int sum = 0;
    int i = 0;
    while (data[i])
        sum += data[i++];
    return sum;
}
```

ХЭШ-ФУНКЦИЯ 3

Возвращает сумму ASCII кодов

```
unsigned int hash_calc4(HashData data) {
    int length = strlen(data);
    if (!length)
        return 0;
    return hash_calc3(data) / length;
}
```

ХЭШ-ФУНКЦИЯ 4

Возвращает сумму ASCII кодов, поделенную на длину строки

```
unsigned int hash_calc5(HashData data) {
    unsigned int result = 0;
    int i = 0;
    while(data[i]) {
        result ^= data[i++];
        result = (result<<1 | result>>31);
    }
    return result;
}
```

ХЭШ-ФУНКЦИЯ 5

Изначально хэш равен нулю, потом делается xor со следующим ASCII кодом в строке, затем происходит циклический сдвиг хэша влево на 1 бит.

```
unsigned int hash_calc6(HashData data) {
    unsigned int len = strlen(data);
    unsigned int base = 31;
    unsigned int factor = 1;
    unsigned int result = 0;
    for (unsigned int i = 0; i < len; ++i) {
        result += data[i]*factor;
        factor *= base;
    }
    return result;
}
```

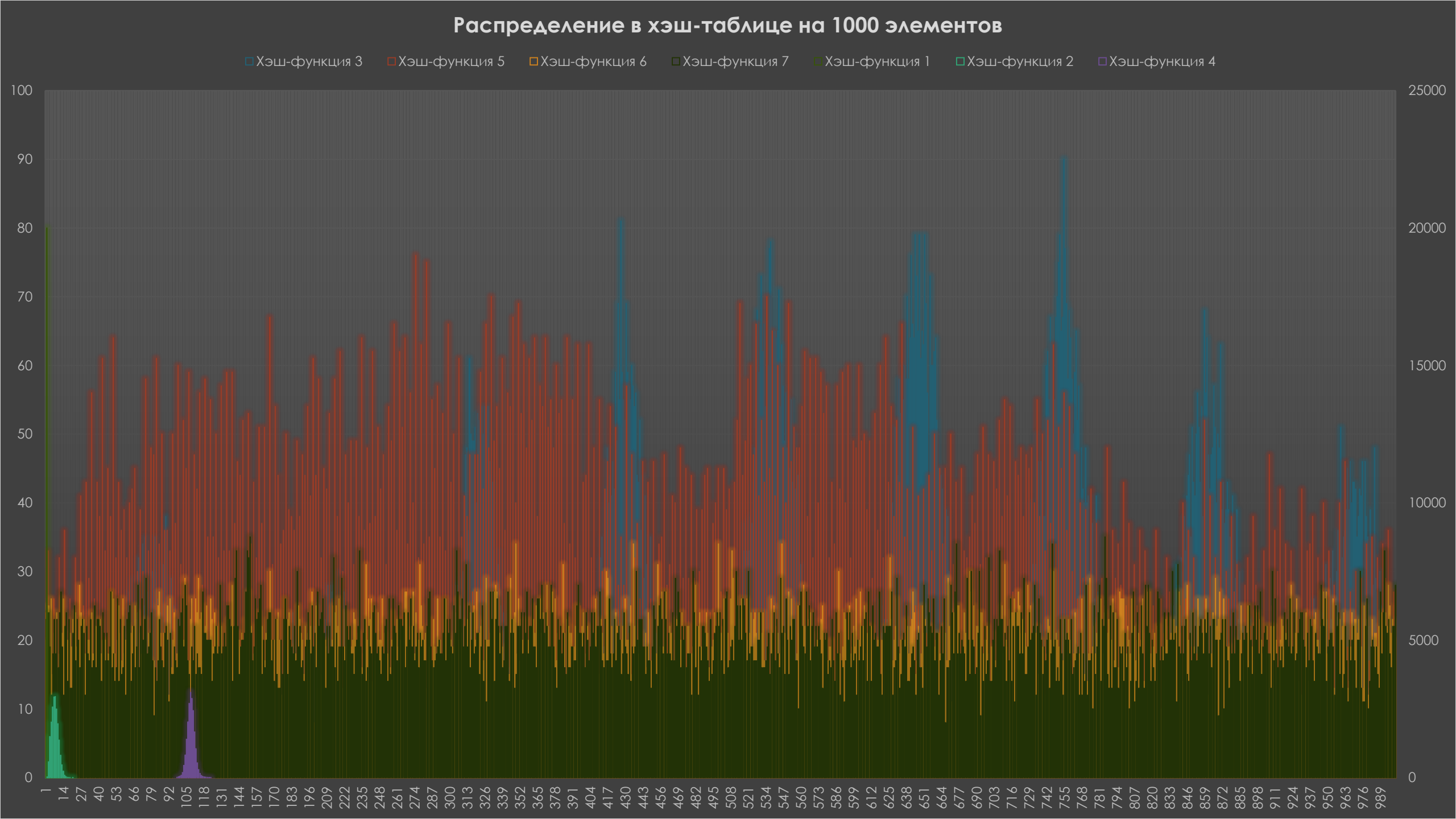
```
unsigned int hash_calc7(HashData data) {
    unsigned int len = strlen(data);
    unsigned int base = 53;
    unsigned int factor = 1;
    unsigned int result = 0;
    for (unsigned int i = 0; i < len; ++i) {
        result += data[i]*factor;
        factor *= base;
    }
    return result;
}
```

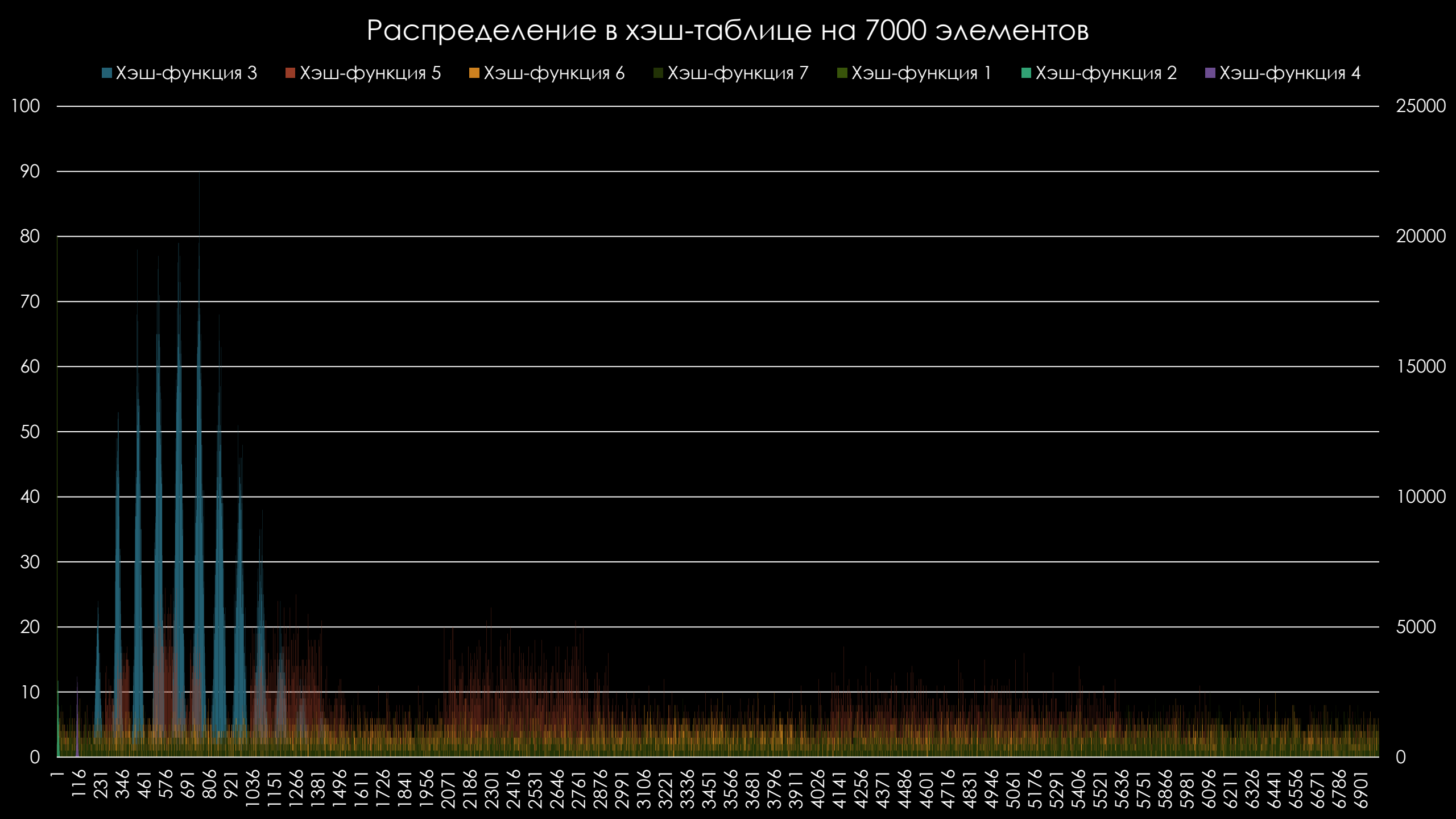
ХЭШ-ФУНКЦИИ 6 И 7

Изначально имеется множитель = 1 и базовый множитель*, хэш = 0. К хэшу прибавляется следующий ASCII код, умноженный на множитель, сам же множитель домножается на базовый множитель

*В хэш-фукнции 6 базовый множитель = 31, а в хэш-функции 7 он равен 53

Всего в хэш-таблицу помещается 20000 слов (все различны). Далее будут графики распределения слов по хэшам для хэш-таблиц размеров 1000, 7000, 10000 (1,2 и 4 хэш-функции отображены на дополнительной оси)


Распределение в хэш-таблице на 1000 элементов
Хэш-функция 3
Хэш-функция 5
Хэш-функция 6
Хэш-функция 7
Хэш-функция 1
Хэш-функция 2
Хэш-функция 4
100
90
80
70
60
50
40
30
20
10
0
25000
20000
15000
10000
5000
0
1
14
27
40
53
66
79
92
105
118
131
144
157
170
183
196
209
222
235
248
261
274
287
300
313
326
339
352
365
378
391
404
417
430
443
456
469
482
495
508
521
534
547
560
573
586
599
612
625
638
651
664
677
690
703
716
729
742
755
768
781
794
807
820
833
846
859
872
885
898
911
924
937
950
963
976
989


Распределение в хэш-таблице на 7000 элементов
Хэш-функция 3
Хэш-функция 5
Хэш-функция 6
Хэш-функция 7
Хэш-функция 1
Хэш-функция 2
Хэш-функция 4
100
90
80
70
60
50
40
30
20
10
0
25000
20000
15000
10000
5000
0
1
116
231
346
461
576
691
806
921
1036
1151
1266
1381
1496
1611
1726
1841
1956
2071
2186
2301
2416
2531
2646
2761
2876
2991
3106
3221
3336
3451
3566
3681
3796
3911
4026
4141
4256
4371
4486
4601
4716
4831
4946
5061
5176
5291
5406
5521
5636
5751
5866
5981
6096
6211
6326
6441
6556
6671
6786
6901


Распределение в хэш-таблице на 10000 элементов
Хэш-функция 3
Хэш-функция 5
Хэш-функция 6
Хэш-функция 7
Хэш-функция 1
Хэш-функция 2
Хэш-функция 4
100
90
80
70
60
50
40
30
20
10
0
25000
20000
15000
10000
5000
0
1
122
243
364
485
606
727
848
969
1090
1211
1332
1453
1574
1695
1816
1937
2058
2179
2300
2421
2542
2663
2784
2905
3026
3147
3268
3389
3510
3631
3752
3873
3994
4115
4236
4357
4478
4599
4720
4841
4962
5083
5204
5325
5446
5567
5688
5809
5930
6051
6172
6293
6414
6535
6656
6777
6898
7019
7140
7261
7382
7503
7624
7745
7866
7987
8108
8229
8350
8471
8592
8713
8834
8955
9076
9197
9318
9439
9560
9681
9802
9923

ВЫВОДЫ

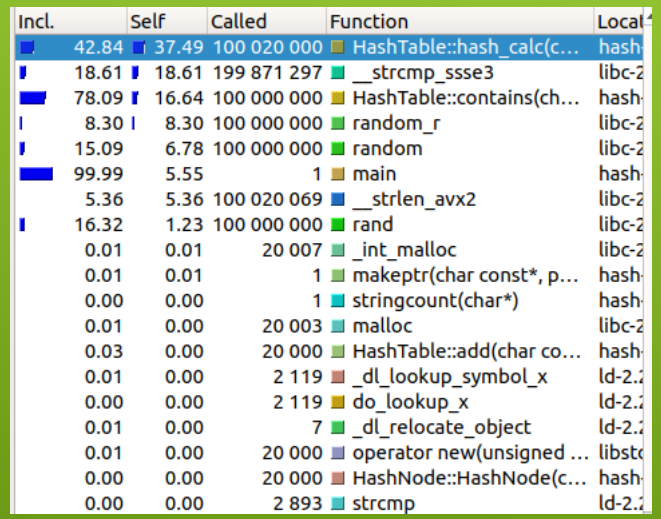
Хуже всех оказался первый хэш (оно и очевидно), лучшие хэши – 6 и 7.
Размер хэш-таблицы 1000 оказался маловат для 20 тысяч слов – на графике видно много коллизий.
Разница между хэш-таблицами размеров 7000 и 10000 не особо чувствуется – во втором варианте лишь немного меньше коллизий.

ОПТИМИЗАЦИЯ ХЕШ-ТАБЛИЦЫ

- В качестве основной хэш-функции я выбрал седьмую

```
unsigned int hash_calc(HashData data) {
    unsigned int len = strlen(data);
    unsigned int base = 53;
    unsigned int factor = 1;
    unsigned int result = 0;
    for (unsigned int i = 0; i < len; ++i) {
        result += data[i]*factor;
        factor *= base;
    }
    return result;
}
```

- Проанализировал вызовы функций в моей программе, используя callgrind. Манускрипты callgrind’а я прочитал с помощью kcachegrind.



Incl.	Self	Called	Function	Local
42.84	37.49	100 020 000	HashTable::hash_calc(c…	hash
18.61	18.61	199 871 297	__strcmp_ssse3	libc-2
78.09	16.64	100 000 000	HashTable::contains(ch…	hash
8.30	8.30	100 000 000	random_r	libc-2
15.09	6.78	100 000 000	random	libc-2
99.99	5.55	1	main	hash
5.36	5.36	100 020 069	__strlen_avx2	libc-2
16.32	1.23	100 000 000	rand	libc-2
0.01	0.01	20 007	_int_malloc	libc-2
0.01	0.01	1	makeptr(char const*, p…	hash
0.00	0.00	1	stringcount(char*)	hash
0.01	0.00	20 003	malloc	libc-2
0.03	0.00	20 000	HashTable::add(char co…	hash
0.01	0.00	2 119	_dl_lookup_symbol_x	ld-2.2
0.00	0.00	2 119	do_lookup_x	ld-2.2
0.01	0.00	7	_dl_relocate_object	ld-2.2
0.01	0.00	20 000	operator new(unsigned …	libst
0.00	0.00	20 000	HashNode::HashNode(c…	hash
0.00	0.00	2 893	strcmp	ld-2.2

Как бы прискорбно это не звучало, большую часть заняло исполнение функции подсчёта хэша. (37,49%). Именно поэтому я принял решение о реализации этой функции на ассемблере.

```
unsigned int hash_calc(HashData data) {
    unsigned int len = strlen(data);
    unsigned int result = 0;

    /*unsigned int factor = 1;

    for (unsigned int i = 0; i < len; ++i) {
        result += data[i]*factor;
        factor *= base;
    }*/

    asm (".intel_syntax noprefix\n"
         "xor edx, edx\n"
         "xor edi, edi\n inc edi\n"
         "loop_note:\n"
         "xor eax, eax\n"
         "lodsb\n"
         "mov r8, rdx\n"
         "mul edi\n"
         "mov rdx, r8\n"
         "add edx, eax\n"
         "mov eax, edi\n"
         "mul bl\n"
         "mov edi, eax\n"
         "loop loop_note\n"
         ".att_syntax prefix\n"
         :"=d"(result), "=c"(len)
         :"b" (HASH_CALC_BASE), "c"(len), "S"(data)
         :"%eax", "%edi");
    return result;
}
```

ВОТ ТАКАЯ ВОТ АССЕМБЛЕРНАЯ ВСТАВКА

```
const unsigned int HASH_TABLE_SIZE = 10000;

const unsigned int SEARCH_NUMBER = 100000000;
```

Теперь работаем с хэш-таблицей размера 10000
и обрабатываем 100 млн запросов поиска

ВРЕМЯ РАБОТЫ	
ДО ASM ОПТИМИЗАЦИИ	ПОСЛЕ ASM ОПТИМИЗАЦИИ
9,86 секунд	9,31 секунд

Результат налицо!
Коэффициент ускорения (©Дед): ((9,86/9,31) / 13) * 1000 = 81,46

СПАСИБО ЗА ВНИМАНИЕ!