A3. HyperMegaLogLog Pro Max++

Файзуллин Илья Денисович БПИ244

Этап 1. Создание инфраструктуры

Реализовано 2 класса: **HashFuncGen** и **RandomStreamGen**.

RandomStreamGen имеет всего 2 метода и просто генерирует случайный поток из небольших строк, а также может разделить его на равные кусочки

HashFuncGen имеет 3 функции - одна переводит строки в числа fnv методом. И метод, который рушит локальные зависимости. И последняя - объединяет их и переводитв 32 бита

```
# ifndef UNTITLED4_RANDOMSTREAMGEN_H
# define UNTITLED4_RANDOMSTREAMGEN_H
# include <string>
# include <vector>
# include <random>
# include <cmath>
# include <cstdint>

class RandomStreamGen {
private:
    std::mt19937 gen_;
    size_t n_;
    std::string rand_string() {
        static const std::string A = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz-";
        std::uniform_int_distribution<int> lenDist(1, 30);
        std::uniform_int_distribution<int> chDist(0, static_cast<int>(A.size() - 1));

        int L = lenDist(gen_);
        std::string s(L, '\0');
        for (int i = 0; i < L; ++i) s[i] = A[chDist(gen_)];
        return s;
    }
public:
    RandomStreamGen(size_t n, size_t seed =42) : gen_(seed), n_(n) {}
    std::vector<std::string> gen_stream() {
        std::vector<std::string> s;
        s.reserve(n_);
        for (size_t i = 0; i < n_; ++i) {s.push_back(rand_string());}
        return s;
    }

    std::vector<size_t> split_prefixes(double step) {
        std::vector<size_t> splt;

        for (double i = step; i < 1; i+=step) {
            splt.push_back(static_cast<size_t>(std::floor(i * n_)));
        }
        if (splt.empty() || splt.back() != n_) splt.push_back(n_);
        return splt;
    }
};
# endif
```

```
# ifndef UNTITLED4_HASHFUNCGEN_H
# define UNTITLED4_HASHFUNCGEN_H
# include <string>
# include <vector>
# include <random>
```

```
# include <cmath>
# include <cstdint>
class HashFuncGen {
public:

    static uint64_t fnv(const std::string& str)
    {
        uint64_t hash = 14695981039346656037ULL; //константы я загуглил
        const uint64_t prime = 1099511628211ULL;
        for (char c : str) {
            hash ^= static_cast<uint64_t>(c);
            hash *= prime;
        }
        return hash;
    }

    static uint64_t mixing(uint64_t x) { // перемешка, чтобы убрать локальные зависимости, чиселки тоже подсмотрел
        x += 0x9e3779b97f4a7c15ULL;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9ULL;
        x = (x ^ (x >> 27)) * 0x94d049bb133111ebULL;
        return x;

    }

    static uint32_t hash32(const std::string& s) {
        uint64_t h64 = mixing(fnv(s));
        return static_cast<uint32_t>(h64 >> 32);
    }
};

# endif
```

Далее я сгенерировал 1000000 хешей и ниже проверим их равномерность

```
# include <string>
# include <vector>
# include <random>
# include <cmath>
# include <cstdint>
# include <iostream>
# include <fstream>
# include "HashFuncGen.h"
# include "RandomStreamGen.h"



int main() {
    const size_t N = 1'000'000;
    RandomStreamGen gen(N, 1309);
    auto stream = gen.gen_stream();

    std::ofstream out("../hashes.txt");
    for (size_t i = 0; i < stream.size(); ++i) {
        uint64_t h = HashFuncGen::hash32(stream[i]);
        out << h << "\n";
    }
    std::cout << "aboba";
    return 0;
}
```

```
    import numpy as np
```

```
    from google.colab import files
    uploaded = files.upload()
```

Выбрать файлы raw_B20.csv

raw_B20.csv(text/csv) - 10650 bytes, last modified: 08.02.2026 - 100% done

Saving raw_B20.csv to raw_B20.csv

```
hashes = np.loadtxt("hashes.txt", dtype=np.uint32)
```

```
print(hashes[:100])
```

```
[1786343999 3439707942 3281406200  920051949 2737039149 2493028742
 1651444208 1786943864 2381247689 1076108378  861490856 2772218071
  601710209 2510016012 4149560003 2864013481  942242003 1110429701
 2570667128  449687755 3066606356 3108634907 3240981703  659169115
 1438800023  297790569 2019354434 2904016840 3407367372 2253825611
 1865938621 3108755918 3835939823 1440510635 3844843126 2330108007
 3833361784 2731050190 3258497598 2539717032 3920071278  940731035
  901203497 1204346845 3330864308   98421231 3717433039  274641830
  923682674 1671471034  524002503 2145636775  900606988 2491680913
 1576263344 3096397268 3455335243 3448437440  675734478  188267320
  860906836 1851086231 2433619141 3492525975 4063551937 3915243220
 1025174334 2283409952 4274913586 1531576338 2208400629 1875038337
 3192463231 1814121262 3287322130 1145644326  655368482 2477349996
 1391383613 3448681454 2232369558 1716703849 2570604478 3998614045
  567577275 4263730909  669191758 3998510940 3252516509 1753268679
  216837520 3756024867  720997429 1907852835 2663505985 2794925470
   91618300 2586972017  517161714 3377367771]
```

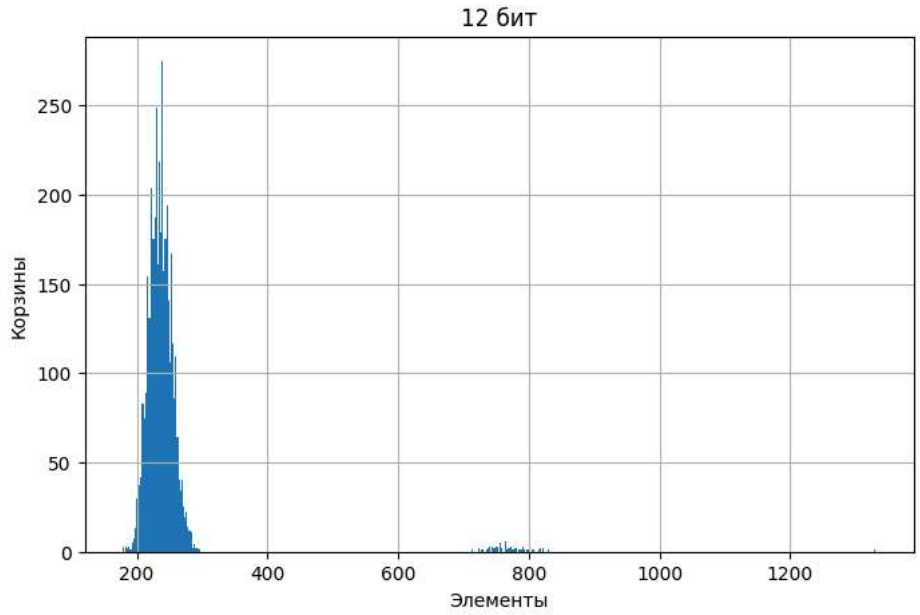
```
print("N =", hashes.size)
print("min hash =", hashes.min(), "max hash =", hashes.max())
```

```
N = 1000000
min hash = 2438 max hash = 4294963573
```

```
B = 12
m = 2**B

bins = (hashes >> (32 - B)).astype(np.int32)
counts = np.bincount(bins, minlength=m)

print("Buckets =", m)
print("min =", counts.min(),
      "max =", counts.max(),
      "mean =", counts.mean())
```

```
Buckets = 4096
min = 178 max = 1331 mean = 244.140625
```

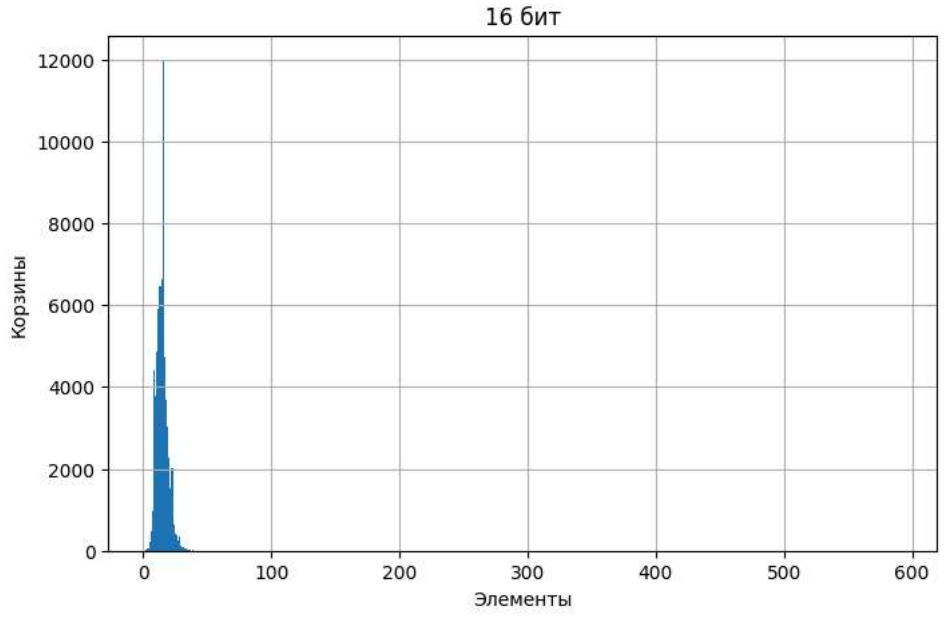
```
import matplotlib.pyplot as plt

plt.figure(figsize=(8,5))
plt.hist(counts, bins=500)
plt.title(f"{B} бит")
plt.xlabel("Элементы")
plt.ylabel("Корзины")
plt.grid(True)
plt.show()
```




```
B = 16
m = 2**B

bins = (hashes >> (32 - B)).astype(np.int32)
counts = np.bincount(bins, minlength=m)

print("Buckets =", m)
print("min =", counts.min(),
      "max =", counts.max(),
      "mean =", counts.mean())
```

```
Buckets = 65536
min = 2 max = 590 mean = 15.2587890625
```

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8,5))
plt.hist(counts, bins=500)
plt.title(f"{B} бит")
plt.xlabel("Элементы")
plt.ylabel("Корзины")
plt.grid(True)
plt.show()
```



Картинка выглядит очень равномерно с небольшими выбросами

Этап 2. Реализация и оценка точности стандартного алгоритма HyperLogLog

Был реализован класс HyperLogLog, который в точности повторяет алгоритм с семинара

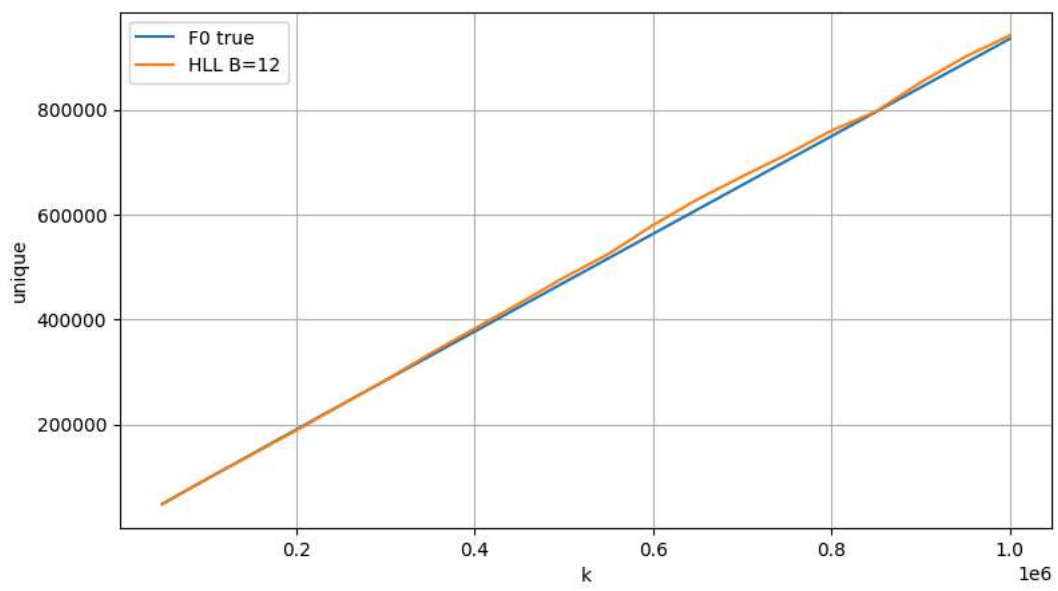
```
# ifndef UNTITLED4_HYPERLOGLOG_H
# define UNTITLED4_HYPERLOGLOG_H

# include <vector>
# include <cstdint>
# include <cmath>
# include <algorithm>

class HyperLogLog {
private:
    int B_;
    uint32_t m_;
    std::vector<uint8_t> M_;

    static uint8_t rho(uint32_t tail, int tailBits) {
        if (tail == 0) return static_cast<uint8_t>(tailBits + 1);
        uint8_t cnt = 1;
        for (int i = tailBits - 1; i >= 0; --i) {
            if ((tail >> i) & 1u) break;
            cnt++;
        }
        return cnt;
    }

    double alpha() const {
        return 0.7213 / (1.0 + 1.079 / static_cast<double>(m_));
    }

public:
    HyperLogLog(int B) : B_(B), m_(1u << B), M_(m_, 0) {}

    void add(uint32_t h) {
        uint32_t j = h >> (32 - B_);
        int tailBits = 32 - B_;
        uint32_t tailMask = (tailBits == 32) ? 0xFFFFFFFFu : ((1u << tailBits) - 1u);
        uint32_t tail = h & tailMask;

        uint8_t r = rho(tail, tailBits);
        if (r > M_[j]) M_[j] = r;
    }

    double estimate() const {
        double Z = 0.0;
        double V = 0;

        for (uint8_t v : M_) {
            Z += std::ldexp(1.0, -v);
            if (v == 0) V++;
        }

        double E = alpha() * static_cast<double>(m_) * static_cast<double>(m_) / Z;

        if (E <= 2.5 * static_cast<double>(m_) && V > 0) {
            E = (static_cast<double>(m_) * std::log(static_cast<double>(m_) / V));
        }
        return E;
    }
};


# endif //UNTITLED4_HYPERLOGLOG_H
```

А также был было сгенерировано 1000000 элементов и поток разбили на 20 частей и сохранили в файл, на основе которого будет построен график

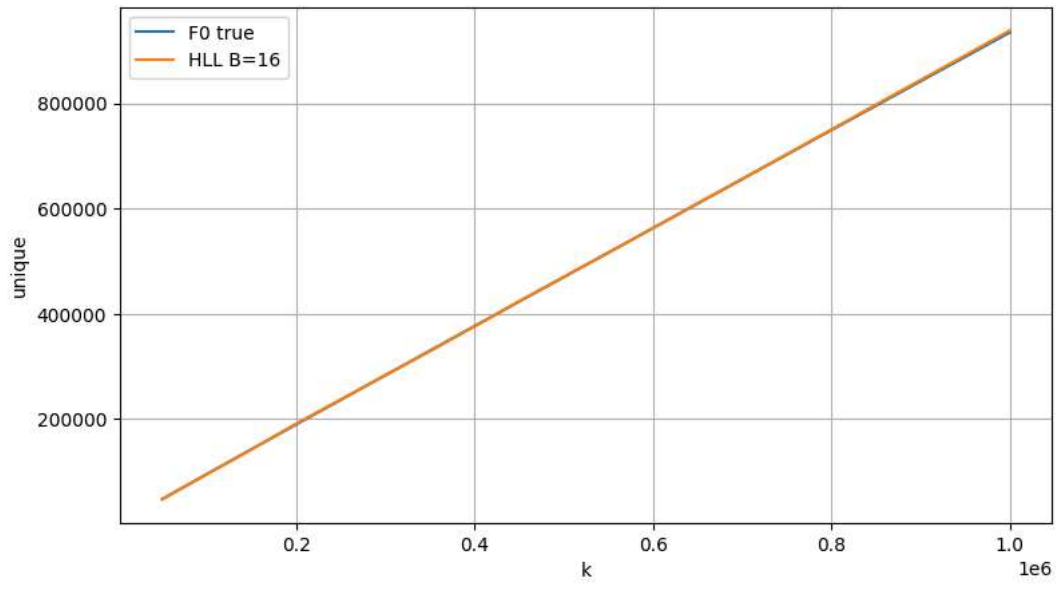
```
# include <iostream>
# include <fstream>
# include <vector>
# include <string>
# include <unordered_set>
# include <cstdint>
# include <cmath>

# include "RandomStreamGen.h"
# include "HashFuncGen.h"
# include "HyperLogLog.h"

int main() {
    const size_t N = 1000000;
    const double step = 0.05;
    const int B = 12;

    RandomStreamGen gen(N, 228);
    auto stream = gen.gen_stream();
    auto splt = gen.split_prefixes(step);

    HyperLogLog hll(B);
    std::unordered_set<std::string> seen;
    seen.reserve(N * 2);

    std::ofstream out("../data_curve.csv");
    out << "k,F0,N_est\n";

    size_t next_idx = 0;
    size_t next_k = splt[next_idx];

    for (size_t i = 0; i < stream.size(); ++i) {
        seen.insert(stream[i]);

        uint32_t h32 = HashFuncGen::hash32(stream[i]);
        hll.add(h32);

        if (i + 1 == next_k) {
            auto F0 = seen.size();
            double N_est = hll.estimate();
            out << i + 1 << "," << F0 << "," << N_est << "\n";

            next_idx++;
            if (next_idx >= splt.size()) break;
            next_k = splt[next_idx];
        }
    }

    std::cout << "aboba";
    return 0;
}
```

```
    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt

    df12 = pd.read_csv("data_curve12.csv")
    df16 = pd.read_csv("data_curve16.csv")
    df20 = pd.read_csv("data_curve20.csv")

    assert (df12["k"].values == df16["k"].values).all()
    assert (df12["k"].values == df20["k"].values).all()
    k = df12["k"].values
    F0 = df12["F0"].values.astype(float)

    plt.figure(figsize=(9,5))
```

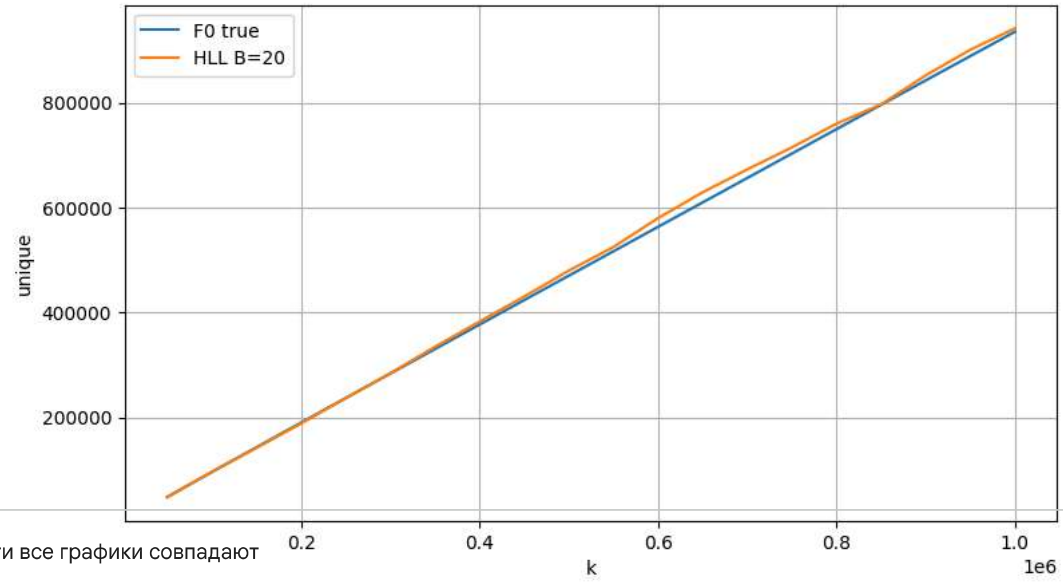
```
plt.plot(k, F0, label="F0 true")
plt.plot(k, df12["N_est"], label="HLL B=12")
plt.xlabel("k")
plt.ylabel("unique")
plt.grid(True)
plt.legend()
plt.show()
```




```
plt.figure(figsize=(9,5))
plt.plot(k, F0, label="F0 true")
plt.plot(k, df16["N_est"], label="HLL B=16")
plt.xlabel("k")
plt.ylabel("unique")
plt.grid(True)
plt.legend()
plt.show()
```

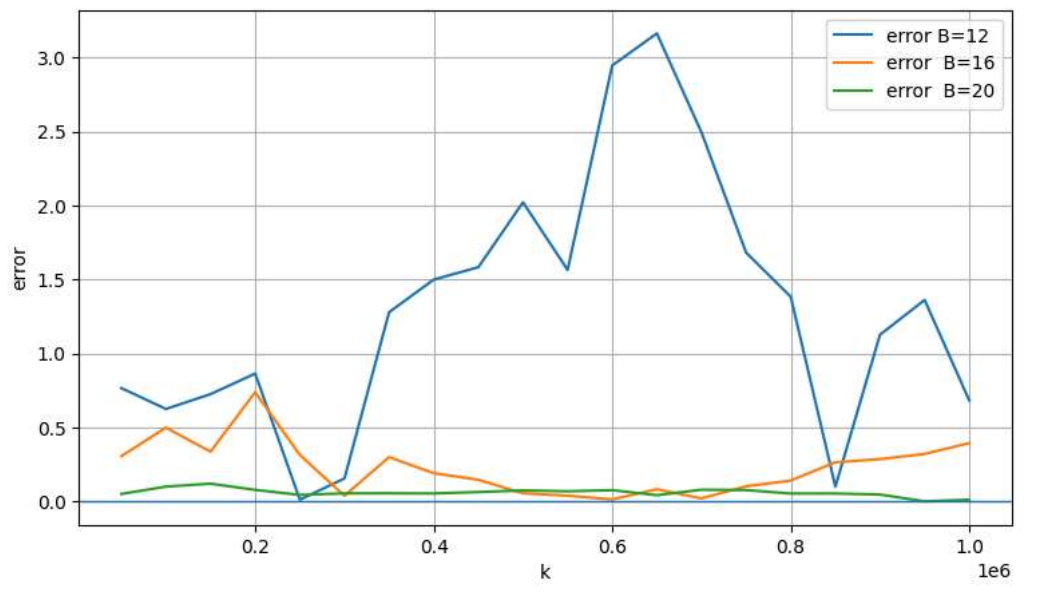



```
plt.figure(figsize=(9,5))
plt.plot(k, F0, label="F0 true")
plt.plot(k, df12["N_est"], label="HLL B=20")
plt.xlabel("k")
plt.ylabel("unique")
plt.grid(True)
plt.legend()
plt.show()
```



Почти все графики совпадают

```
def rel_err(df):
    return abs((df["N_est"].values - F0) / F0 *100)

e12 = rel_err(df12)
e16 = rel_err(df16)
e20 = rel_err(df20)

plt.figure(figsize=(9,5))
plt.plot(k, e12, label="error B=12")
plt.plot(k, e16, label="error  B=16")
plt.plot(k, e20, label="error  B=20")
plt.axhline(0, linewidth=1)
plt.xlabel("k")
plt.ylabel("error")
plt.grid(True)
plt.legend()
plt.show()
```

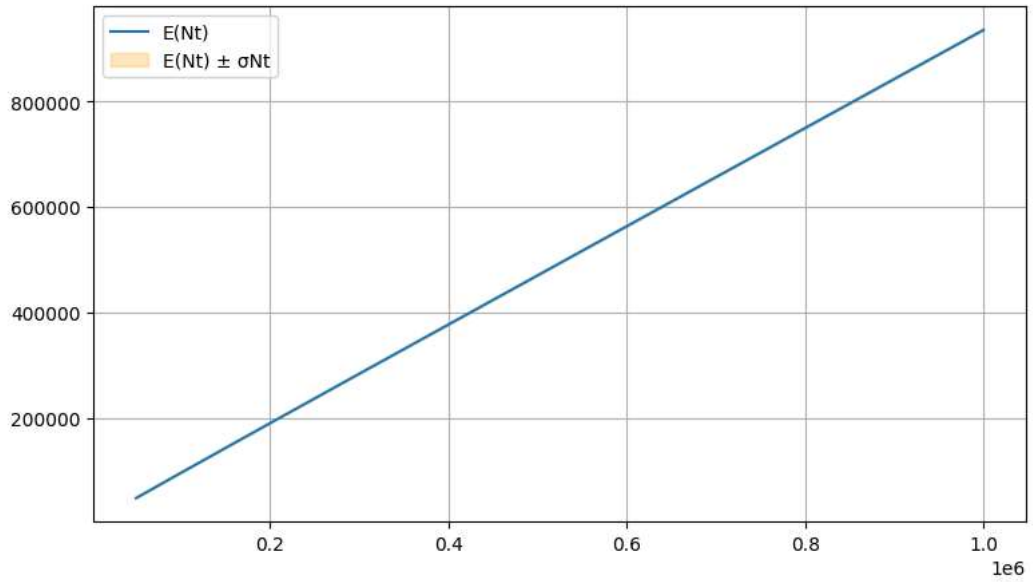


Ожидаемо, что B=20 даёт наилучшую оценку. Поэтому возьмём его для подсчёта матожидания и отклонения

```
# include <iostream>
# include <fstream>
# include <vector>
# include <string>

# include "RandomStreamGen.h"
# include "HashFuncGen.h"
# include "HyperLogLog.h"
```

```
int main() {
    const size_t N = 1'000'000;
    const double step = 0.05;
    const int B = 20;
    const int R = 30;

    RandomStreamGen tmp(N, 1);
    auto ks = tmp.split_prefixes(step);

    std::ofstream out("../raw_B20.csv");
    out << "k,run,Nt\n";

    for (int r = 0; r < R; ++r) {
        RandomStreamGen gen(N, 1000 + r);
        auto stream = gen.gen_stream();

        HyperLogLog hll(B);

        size_t idx = 0;
        size_t next_k = ks[idx];

        for (size_t i = 0; i < stream.size(); ++i) {
            uint32_t h = HashFuncGen::hash32(stream[i]);
            hll.add(h);

            size_t processed = i + 1;
            if (processed == next_k) {
                out << processed << "," << r << "," << hll.estimate() << "\n";
                idx++;
                if (idx >= ks.size()) break;
                next_k = ks[idx];
            }
        }
    }

    return 0;
}
```

```
    df = pd.read_csv("raw_B20.csv")

    grp = df.groupby("k")["Nt"]

    mean = grp.mean()
    std = grp.std()

    k = mean.index.values
    mu = mean.values
    sd = std.values

    plt.figure(figsize=(9,5))
    plt.plot(k, mu, label="E(Nt)")
    plt.fill_between(k, mu - sd, mu + sd, color ='orange', alpha=0.25, label="E(Nt) ± σNt")
    plt.grid(True)
    plt.legend()
    plt.show()
    print(std, mean)
```




```
k
50000       57.681400
100000      93.341292
150000     132.059065
200000     171.899225
250000     195.643060
300000     219.840812
350000     269.267600
399999     280.289319
449999     327.068315
499999     377.378946
549999     410.747505
600000     442.239963
650000     513.163571
700000     546.910735
750000     620.721568
800000     601.579206
850000     694.104935
900000     692.168839
950000     681.487748
1000000    718.987111
Name: Nt, dtype: float64 k
50000       48082.453333
100000      95629.703333
150000     142825.300000
200000     189854.966667
250000     236680.000000
300000     283460.500000
350000     330126.833333
399999     376756.366667
449999     423347.800000
499999     469936.400000
549999     516515.266667
600000     563102.566667
650000     609678.666667
700000     656182.400000
750000     702621.100000
800000     749125.866667
850000     795765.833333
900000     842222.233333
950000     888677.633333
1000000    935125.500000
Name: Nt, dtype: float64
```

Я попытался нарисовать, но они почти не видны

Этап 3. Анализ результатов работы стандартного алгоритма HyperLogLog

Посчитаем ошибки для B = 12, 16 и 20.

B = 12

1.3/64 = 2.03%

1.04/64 = 1.63%

Видно, что почти всегда за исключением выброса на 0.6 оценка меньше 2.03% и кроме 0.5 - ниже 1.63

B = 16

1.3/256 = 0.51%

1.04/256 = 0.41%

В самом начале обе границы не соблюдены, но с увеличением элементов всё становится хорошо

B = 32

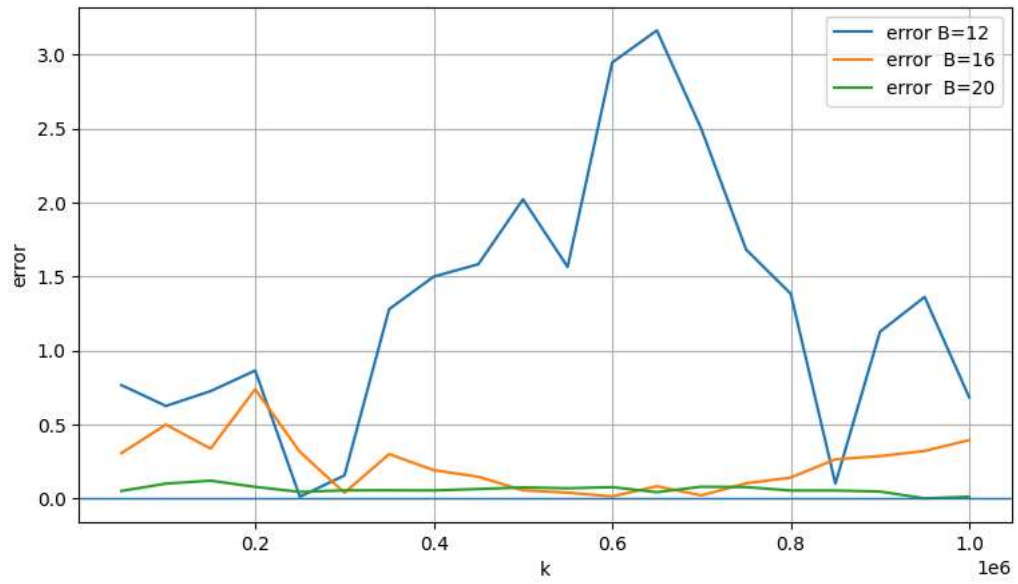
1.3/1024 = 0.13%

1.04/1024 = 0.1%

Тут вообще все границы соблюдены

```
def rel_err(df):
    return abs((df["N_est"].values - F0) / F0 *100)
```

```
e12 = rel_err(df12)
e16 = rel_err(df16)
e20 = rel_err(df20)

plt.figure(figsize=(9,5))
plt.plot(k, e12, label="error B=12")
plt.plot(k, e16, label="error  B=16")
plt.plot(k, e20, label="error  B=20")
plt.axhline(0, linewidth=1)
plt.xlabel("k")
plt.ylabel("error")
plt.grid(True)
plt.legend()
plt.show()
```




```
std
```

k	Nt
50000	57.681400
100000	93.341292
150000	132.059065
200000	171.899225
250000	195.643060
300000	219.840812
350000	269.267600
399999	280.289319
449999	327.068315
499999	377.378946
549999	410.747505
600000	442.239963
650000	513.163571
700000	546.910735
750000	620.721568
800000	601.579206
850000	694.104935
900000	692.168839
950000	681.487748
1000000	718.987111

dtype: float64