БУ ВО Ханты-Мансийского автономного округа – Югры

«Сургутский государственный университет»

Политехнический институт

Кафедра автоматики и компьютерных систем

ОТЧЁТ

по лабораторной работе №3

по дисциплине: «Алгоритмы и структуры данных»

Выполнил: студент(ка) группы №609-31,

Гаврилов Е.Е.

Принял: старший преподаватель кафедры АиКС

Назаров Е.В.

Сургут

2025г.

**Цель работы**: изучить базовые алгоритмы поиска, исследовать их свойства, закрепить навыки структурного программирования.

**Задание:**

1. Разработать функции, реализующие последовательный поиск, поиск «прыжками», бинарный поиск, поиск Фибоначчи и интерполяционный поиск.

2. Исследовать алгоритмы поиска: построить и сравнить зависимости среднего количества сравнений в случаях успешного и неуспешного поиска для заданных алгоритмов.

3. Найти зависимость оптимальной величины «прыжка» от размера последовательности.

4. Составить отчет, в котором привести графики полученных зависимостей, анализ свойств алгоритмов и выводы по работе.

Ход работы:

1.Разработать функции для Линейного, Бинарного, Интерполяционного поиска и поиска Прыжками (1- и 2- уровневый).

2. Исследовать алгоритмы поиска: построить и сравнить зависимости среднего количества сравнений в случаях успешного и неуспешного поиска для заданных алгоритмов (последовательность для поиска в данном случае – отсортированная неубывающая последовательность целых чисел с постоянным шагом. Длина прыжков для одноуровневого поиска – 1/100 от длины последовательности, для двухуровневого – 1/10 и 1/100 от длины последовательности).

Таблица 1. Зависимость среднего количества сравнений от типа поиска и длинны входной последовательности в случаях успешного поиска.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Длинна | Линейный | Прыжками(1) | Прыжками(2) | Бинарный | Фибоначчи | Интерполяционный |
| 2000 | 1006 | 59 | 21 | 14 | 14 | 2 |
| 4000 | 2007 | 71 | 31 | 15 | 15 | 2 |
| 6000 | 2990 | 80 | 41 | 16 | 16 | 2 |
| 8000 | 3945 | 91 | 50 | 17 | 17 | 2 |
| 10000 | 5196 | 99 | 61 | 17 | 17 | 2 |
| 12000 | 5956 | 109 | 73 | 18 | 18 | 2 |
| 14000 | 7177 | 120 | 80 | 18 | 18 | 2 |
| 16000 | 8018 | 128 | 94 | 19 | 18 | 2 |
| 18000 | 8837 | 139 | 102 | 19 | 19 | 2 |
| 20000 | 9816 | 148 | 110 | 19 | 19 | 2 |

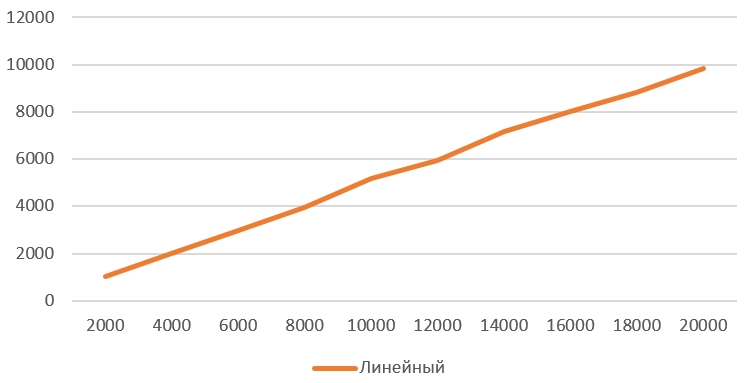


Рисунок 1.Зависимость количества сравнений от длинный последовательности для алгоритма линейного поиска.

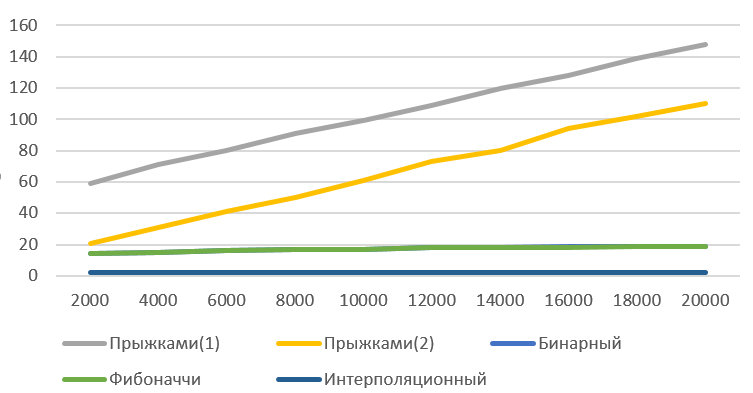


Рисунок 2.Зависимость количества сравнений от длинный последовательности для прочих алгоритмов поиска.

Таблица 2.Зависимость среднего количества сравнений от типа поиска и длинны входной последовательности в случаях неуспешного поиска.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Длинна | Линейный | Прыжками(1) | Прыжками(2) | Бинарный | Фибоначчи | Интерполяционный |
| 2000 | 2000 | 61 | 21 | 16 | 17 | 0 |
| 4000 | 4000 | 70 | 31 | 17 | 18 | 1 |
| 6000 | 6000 | 80 | 41 | 18 | 19 | 1 |
| 8000 | 8000 | 92 | 51 | 19 | 20 | 1 |
| 10000 | 10000 | 101 | 61 | 19 | 20 | 1 |
| 12000 | 12000 | 111 | 71 | 20 | 21 | 1 |
| 14000 | 14000 | 119 | 81 | 20 | 21 | 1 |
| 16000 | 16000 | 129 | 89 | 20 | 21 | 0 |
| 18000 | 18000 | 140 | 99 | 21 | 22 | 1 |
| 20000 | 20000 | 148 | 109 | 21 | 22 | 1 |

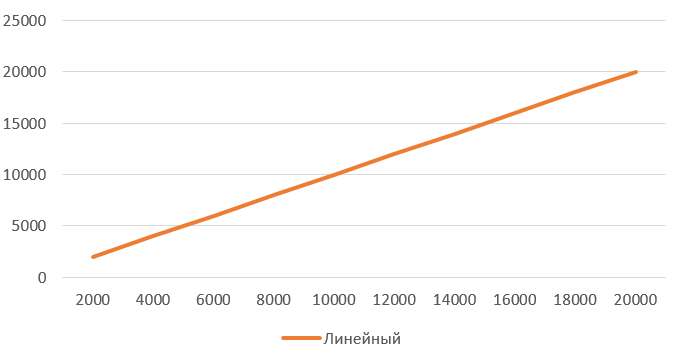


Рисунок 3.Зависимость количества сравнений от длинный последовательности для алгоритма линейного поиска.

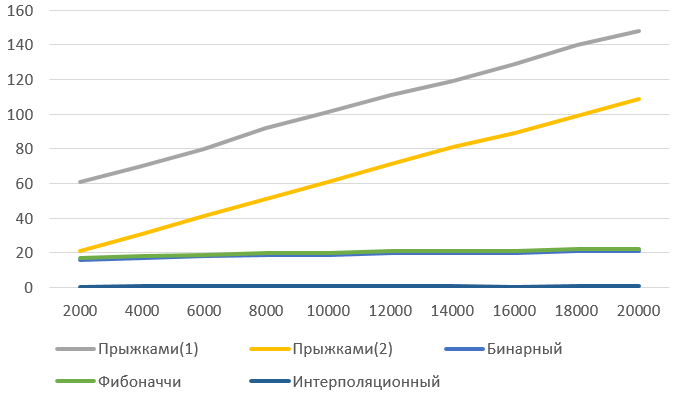


Рисунок 4.Зависимость количества сравнений от длинный последовательности для прочих алгоритмов поиска.

3. Найти зависимость оптимальной величины «прыжка» от размера последовательности.

Использована последовательность из 4096 элементов.

Таблица 3.Зависимость среднего количества сравнений от шага.

|  |  |
| --- | --- |
| Шаг | Количество сравнений |
| 1 | 1990 |
| 2 | 995 |
| 4 | 499 |
| 8 | 253 |
| 16 | 132 |
| 32 | 79 |
| 64 | 63 |
| 128 | 76 |
| 256 | 136 |
| 512 | 252 |
| 1024 | 542 |
| 2048 | 1084 |
| 4096 | 2107 |

Таблица 4.График зависимость среднего количества сравнений от шага.



Самым эффективным для данной последовательности является шаг

4. Анализ полученных данных.

4.1 Линейный поиск является наименее эффективным, в худшем случае количество сравнений равно длине последовательности, однако позволяет проводить поиск в неотсортированной последовательности.

4.2 Поиск прыжками заметно быстрее линейного поиска. Наиболее эффективно брать шаг, равный квадратному корню длинны последовательности.

4.3 Для данных последовательностей бинарный поиск и поиск Фибоначчи показывают очень близкие результаты.

4.4 Для отсортированных последовательностей с распределением значений близким к равномерному, самым эффективным является интерполяционный поиск.

**Вывод:** были разработаны функции реализующие алгоритмы линейного, бинарного, интерполяционного поиска, поиска прыжками и поиска Фибоначчи. Измерена зависимость количества сравнений от длины последовательности для каждого алгоритма. Наименее эффективным для данных последовательностей показал себя линейный поиск, наиболее эффективным – интерполяционный.

Листинг 1. arrgen.h

#ifndef ARRGEN\_H

#define ARRGEN\_H

#include <iostream>

#include <random>

#include <cmath>

#include <fstream>

namespace ArrGen

{

template <typename T>

void IncreasingSequence(T \*array, size\_t len, T min, T max);

template <typename T>

void DecreasingSequence(T \*array, size\_t len, T min, T max);

template <typename T>

void RandomSequence(T \*array, size\_t len, T min, T max);

template <typename T>

void SawtoothSequence(T \*array, size\_t len, T start, T end, size\_t period, size\_t offset);

template <typename T>

void SinSequence(T \*array, size\_t len, T start, T end, double period = 2 \* M\_2\_PI, T offset = 0);

template <typename T>

void StepSequence(T \*array, size\_t len, T start, T end, size\_t period = 10);

template <typename T>

void QuasiOrderedSequence(T \*array, size\_t len, T start, T end, T spread);

template <typename T>

void SawtoothSequence(T \*array, size\_t len, T start, T end);

template <typename T>

void SinSequence(T \*array, size\_t len, T start, T end);

template <typename T>

void StepSequence(T \*array, size\_t len, T start, T end);

template <typename T>

void QuasiOrderedSequence(T \*array, size\_t len, T start, T end);

extern void (\*GenFunArrInt[])(int \*, size\_t, int, int);

extern void (\*GenFunarrDouble[])(double \*, size\_t, double, double);

extern std::string GenFunNames[];

}

#endif

Листинг 2. arrgen.cpp

#include "arrgen.h"

#include <string>

namespace ArrGen

{

template <typename T>

void IncreasingSequence(T \* array, size\_t len, T min, T max)

{

if (nullptr == array || 0 == len)

return;

if (min > max)

{

T temp = min;

min = max;

max = temp;

}

double step = 1.0 \* (max - min) / len;

array[0] = (T)min;

double prev = (T)min;

for (size\_t i = 1; i < len; i++)

{

array[i] = prev + step;

prev += step;

}

}

template <typename T>

void DecreasingSequence(T \* array, size\_t len, T min, T max)

{

if (nullptr == array || 0 == len)

return;

if (min > max)

{

T temp = min;

min = max;

max = temp;

}

double step = 1.0 \* (max - min) / len;

array[0] = max;

double prev = (T)min;

for (size\_t i = 1; i < len; i++)

{

array[i] = prev + step;

prev += step;

}

}

template <typename T>

void RandomSequence(T \* array, size\_t len, T min, T max)

{

if (nullptr == array || 0 == len)

return;

if (min > max)

{

T temp = min;

min = max;

max = temp;

}

std::random\_device random\_device;

std::mt19937 generator(random\_device());

std::uniform\_real\_distribution distribution((double)min, (double)max);

for (size\_t i = 0; i < len; i++)

{

array[i] = distribution(generator);

}

}

template <typename T>

void SawtoothSequence(T \* array, size\_t len, T start, T end, size\_t period, size\_t offset)

{

if (nullptr == array || 0 == len)

return;

double local\_step = 1.0 \* (end - start) / period;

for (size\_t i = 0; i < len; i++)

{

array[i] = start + local\_step \* ((i + offset) % period);

}

}

template <typename T>

void SinSequence(T \* array, size\_t len, T start, T end, double period, T offset)

{

if (nullptr == array || 0 == len)

return;

for (size\_t i = 0; i < len; i++)

{

array[i] = sin(i \* M\_2\_PI \* len / period / period + offset) \* (end - start) / 2 + (end + start) / 2;

}

}

template <typename T>

void StepSequence(T \* array, size\_t len, T start, T end, size\_t period)

{

if (nullptr == array || 0 == len || 0 == period || len < period)

return;

std::random\_device random\_device;

std::mt19937 generator(random\_device());

std::uniform\_real\_distribution distribution((double)0, (double)(end - start) / (len / period));

for (size\_t i = 0; i < len; i++)

{

double ran = distribution(generator);

array[i] = (i / period) \* (end - start) / (len / period) + ran;

}

}

template <typename T>

void QuasiOrderedSequence(T \* array, size\_t len, T start, T end, T spread)

{

if (nullptr == array || 0 == len)

return;

double step = (end - start) / len;

std::random\_device random\_device;

std::mt19937 generator(random\_device());

T max = start > end ? start : end;

T min = start < end ? start : end;

double direct = start;

for (size\_t i = 0; i < len; i++, direct += step)

{

direct = start + 1.0 \* i \* (end - start) / len;

double up\_limit = max > direct + spread ? direct + spread : max;

double low\_limit = min < direct - spread ? direct - spread : min;

std::uniform\_real\_distribution distribution(low\_limit, up\_limit);

double ran = distribution(generator);

array[i] = ran;

}

}

template <typename T>

void SawtoothSequence(T \* array, size\_t len, T start, T end)

{

SawtoothSequence(array, len, start, end, 10, 0);

}

template <typename T>

void SinSequence(T \* array, size\_t len, T start, T end)

{

SinSequence<T>(array, len, start, end, 10 \* M\_2\_PI, 0);

}

template <typename T>

void StepSequence(T \* array, size\_t len, T start, T end)

{

StepSequence<T>(array, len, start, end, 10);

}

template <typename T>

void QuasiOrderedSequence(T \* array, size\_t len, T start, T end)

{

QuasiOrderedSequence<T>(array, len, start, end, 10);

}

}

Листинг 3. lab03.cpp

#include "arrgen.h"

#include <chrono>

#include <iostream>

#include <fstream>

#include <cstdlib>

void PrintArray(int \*array, size\_t size)

{

for (size\_t i = 0; i < size; i++)

std::cout << array[i] << " ";

std::cout << std::endl;

}

template <typename T>

T \*LineSearch(T \*array, size\_t size, T val, size\_t \*counter)

{

T \*result = nullptr;

\*counter = 0;

for (size\_t i = 0; i < size && result == nullptr; i++)

{

(\*counter)++;

if (array[i] == val)

result = array + i;

}

return result;

}

template <typename T>

T \*JumpSearchLVL1(T \*array, size\_t size, T val, size\_t jump, size\_t \*counter)

{

\*counter = 0;

T \*result = nullptr;

size\_t pos;

size\_t pre\_pos;

for (pos = 0, pre\_pos = 0; pos < size && array[pos] <= val && result == nullptr; pos += jump)

{

(\*counter)++;

if (array[pos] == val)

result = array + pos;

else

pre\_pos = pos;

}

if (pos >= size)

pos = size - 1;

for (; pos > pre\_pos && array[pos] >= val && result == nullptr; pos--)

{

(\*counter)++;

if (array[pos] == val)

result = array + pos;

}

return result;

}

template <typename T>

T \*JumpSearchLVL2(T \*array, size\_t size, T val, size\_t jump1, size\_t jump2, size\_t \*counter)

{

T \*result = nullptr;

size\_t pos;

size\_t pre\_pos;

size\_t post\_pos;

\*counter = 0;

for (pos = 0, pre\_pos = 0; pos < size && array[pos] <= val && result == nullptr; pos += jump1)

{

(\*counter)++;

if (array[pos] == val)

result = array + pos;

else

pre\_pos = pos;

}

if (pos >= size)

pos = size - 1;

for (; pos > pre\_pos && pos - jump2 < pos && array[pos] >= val && result == nullptr; pos -= jump2)

{

(\*counter)++;

if (array[pos] == val)

result = array + pos;

else

post\_pos = pos;

}

if (pos - jump2 > pos)

pos = 0;

if (post\_pos == 0)

post\_pos = size - 1;

for (; pos <= post\_pos && array[pos] <= val && result == nullptr; pos++)

{

(\*counter)++;

if (array[pos] == val)

result = array + pos;

}

return result;

}

template <typename T>

T \*BinarySearch(T \*array, size\_t size, T val, size\_t \*counter)

{

T \*result = nullptr;

size\_t low = 0, high = size - 1, pos = size / 2;

\*counter = 0;

for (; (low <= high) && (result == nullptr) && low >= 0 && high <= size - 1; pos = (low + high) / 2)

{

(\*counter)++;

if (array[pos] == val)

result = array + pos;

else if (val < array[pos])

{

(\*counter)++;

high = pos - 1;

}

else

low = pos + 1;

}

return result;

}

template <typename T>

T \*FibonacciSearch(T \*array, size\_t size, T val, size\_t \*counter)

{

int result;

int f1 = 1, f2 = 1, temp;

\*counter = 0;

for (; f1 <= size; f1 += f2, f2 = f1 - f2)

;

result = f2 - 1;

f1 -= f2;

f2 -= f1;

while (f1 + f2 > 0 && f1 >= 0 && f2 >= 0)

{

(\*counter)++;

if (val < array[result])

{

result -= f2;

temp = f2;

f2 = f1 - f2;

f1 = temp;

}

else if (val > array[result])

{

(\*counter)++;

result += f2;

result = size - 1 > result ? result : size - 1;

f1 -= f2;

f2 -= f1;

}

else

return array + result;

}

return nullptr;

}

template <typename T>

T \*InterpolationSearch(T \*array, size\_t size, T val, size\_t \*counter)

{

T \*result = nullptr;

\*counter = 0;

int left = 0, right = size - 1;

int pos = left + val \* (right - left) / (array[right] - array[left]);

while (array[left] <= val && array[right] >= val && result == nullptr)

{

(\*counter)++;

pos = (array[right] \* left - array[left] \* right + val \* (right - left)) / (array[right] - array[left]);

if (array[pos] < val)

left = pos + 1;

else

{

(\*counter)++;

if (array[pos] > val)

right = pos - 1;

else

result = array + pos;

}

}

return result;

}

using namespace std;

using namespace ArrGen;

using namespace std::chrono;

int Comp(int a, int b)

{

return b - a;

}

int main()

{

srand(time(0));

ofstream outstream\_find("out3\_find.txt");

ofstream outstream\_none("out3\_none.txt");

ofstream outstream\_jump("out3\_jump.txt");

ofstream log("log.txt");

outstream\_find << "Search | Len | Compares | X | Status\n";

size\_t len;

int \*a;

size\_t counter;

size\_t m\_counter = 0;

size\_t n\_counter = 0;

int times = 1000;

for (int i = 2; i <= 20; i += 2)

{

len = i \* 1'000;

a = (int \*)malloc(len \* sizeof(int));

ArrGen::IncreasingSequence<int>(a, len, -len, len);

// line

m\_counter = 0;

n\_counter = 0;

for (int k = 0; k < times; k++)

{

int x = (rand() % (len - 1 - (-len + 1) + 1) - (len - 1)) / 2 \* 2;

int \*f = LineSearch<int>(a, len, x, &counter);

m\_counter += counter;

log << "LineSearch " << len << " " << counter << " " << x << " " << (\*f == x ? "SUCCES" : "FAIL") << endl;

f = LineSearch<int>(a, len, x + 1, &counter);

n\_counter += counter;

log << "LineSearch " << len << " " << counter << " " << x + 1 << " " << (f == nullptr ? "SUCCES" : "FAIL") << endl;

}

outstream\_find << "LineSearch " << len << " " << m\_counter / times << " - -" << endl;

outstream\_none << "LineSearch " << len << " " << n\_counter / times << " - -" << endl;

// jump 1

m\_counter = 0;

n\_counter = 0;

for (int k = 0; k < times; k++)

{

int x = (rand() % (len - 1 - (-len + 1) + 1) - (len - 1)) / 2 \* 2;

int \*f = JumpSearchLVL1<int>(a, len, x, i \* 10, &counter);

m\_counter += counter;

log << "JumpSearchLVL1 " << len << " " << counter << " " << x << " " << (\*f == x ? "SUCCES" : "FAIL") << endl;

f = JumpSearchLVL1<int>(a, len, x + 1, i \* 10, &counter);

n\_counter += counter;

log << "JumpSearchLVL1 " << len << " " << counter << " " << x + 1 << " " << (f == nullptr ? "SUCCES" : "FAIL") << endl;

}

outstream\_find << "JumpSearchLVL1 " << len << " " << m\_counter / times << " - -" << endl;

outstream\_none << "JumpSearchLVL1 " << len << " " << n\_counter / times << " - -" << endl;

// jump 2

m\_counter = 0;

n\_counter = 0;

for (int k = 0; k < times; k++)

{

int x = (rand() % (len - 1 - (-len + 1) + 1) - (len - 1)) / 2 \* 2;

int \*f = JumpSearchLVL2<int>(a, len, x, i \* 100, i \* 10, &counter);

m\_counter += counter;

log << "JumpSearchLVL2 " << len << " " << counter << " " << x << " " << (\*f == x ? "SUCCES" : "FAIL") << endl;

f = JumpSearchLVL2<int>(a, len, x + 1, i \* 100, i \* 10, &counter);

n\_counter += counter;

log << "JumpSearchLVL2 " << len << " " << counter << " " << x + 1 << " " << (f == nullptr ? "SUCCES" : "FAIL") << endl;

}

outstream\_find << "JumpSearchLVL2 " << len << " " << m\_counter / times << " - -" << endl;

outstream\_none << "JumpSearchLVL2 " << len << " " << n\_counter / times << " - -" << endl;

// bin

m\_counter = 0;

n\_counter = 0;

for (int k = 0; k < times; k++)

{

int x = (rand() % (len - 1 - (-len + 1) + 1) - (len - 1)) / 2 \* 2;

int \*f = BinarySearch<int>(a, len, x, &counter);

m\_counter += counter;

log << "BinarySearch " << len << " " << counter << " " << x << " " << (\*f == x ? "SUCCES" : "FAIL") << endl;

f = BinarySearch<int>(a, len, x + 1, &counter);

n\_counter += counter;

log << "BinarySearch " << len << " " << counter << " " << x + 1 << " " << (f == nullptr ? "SUCCES" : "FAIL") << endl;

}

outstream\_find << "BinarySearch " << len << " " << m\_counter / times << " - -" << endl;

outstream\_none << "BinarySearch " << len << " " << n\_counter / times << " - -" << endl;

// interpol

m\_counter = 0;

n\_counter = 0;

for (int k = 0; k < times; k++)

{

int x = (rand() % (len - 1 - (-len + 1) + 1) - (len - 1)) / 2 \* 2;

int \*f = InterpolationSearch<int>(a, len, x, &counter);

m\_counter += counter;

log << "InterpolationSearch " << len << " " << counter << " " << x << " " << (\*f == x ? "SUCCES" : "FAIL") << endl;

f = InterpolationSearch<int>(a, len, x + 1, &counter);

n\_counter += counter;

log << "InterpolationSearch " << len << " " << counter << " " << x + 1 << " " << (f == nullptr ? "SUCCES" : "FAIL") << endl;

}

outstream\_find << "InterpolationSearch " << len << " " << m\_counter / times << " - -" << endl;

outstream\_none << "InterpolationSearch " << len << " " << n\_counter / times << " - -" << endl;

// fibonacci

m\_counter = 0;

n\_counter = 0;

for (int k = 0; k < times; k++)

{

int x = (rand() % (len - 1 - (-len + 1) + 1) - (len - 1)) / 2 \* 2;

int \*f = FibonacciSearch<int>(a, len, x, &counter);

m\_counter += counter;

log << "FibonacciSearch " << len << " " << counter << " " << x << " " << (\*f == x ? "SUCCES" : "FAIL") << endl;

f = FibonacciSearch<int>(a, len, x + 1, &counter);

n\_counter += counter;

log << "FibonacciSearch " << len << " " << counter << " " << x + 1 << " " << (f == nullptr ? "SUCCES" : "FAIL") << endl;

}

outstream\_find << "FibonacciSearch " << len << " " << m\_counter / times << " - -" << endl;

outstream\_none << "FibonacciSearch " << len << " " << n\_counter / times << " - -" << endl;

free(a);

}

len = 4'096;

int jumparr[13] = {0};

a = (int \*)malloc(len \* sizeof(int));

ArrGen::IncreasingSequence<int>(a, len, 0, len);

times = 100;

for (int i = 0; i < times; i++)

{

int x = (rand() % len) / 2 \* 2;

for (int j = len; j; j /= 2)

{

int \*f = JumpSearchLVL1<int>(a, len, x, j, &counter);

n\_counter += counter;

log << "JUMP" << i << " " << j << " " << counter << endl;

jumparr[(int)log2(j)] += counter;

}

}

for (int i = 0; i < 13; i++)

outstream\_jump << "JUMP " << (1 << i) << " " << jumparr[i] / times << endl;

free(a);

outstream\_find.close();

log.close();

return 0;

}

#include "arrgen.cpp"