**Бинарный поиск в массиве**

*1. Общая постановка задачи*

Необходимо создать новый компонент для калькулятора, который должен реализовывать новый интерфейс с методом определения элемента в массиве методом бинарного поиска. Интерфейс должен содержать методы с разными типами данных (int, float, double, long double).

*2. Реализуемый алгоритм бинарного поиска.*

Бинарный поиск — это очень быстрый и эффективный алгоритм для нахождения элемента в отсортированном массиве (или списке).

Ключевое требование, без которого он не работает — данные должны быть упорядочены.

Как он работает по шагам?

1. Начало: Алгоритм определяет границы поиска: в самом начале это весь массив (от первого до последнего элемента).
2. Поиск середины: Он находит средний элемент в текущих границах поиска.
3. Сравнение: Значение этого среднего элемента сравнивается с тем значением, которое мы ищем.

* Нашли! Если средний элемент и есть искомый, поиск успешно завершен.
* Искомый элемент меньше. Если искомый элемент меньше среднего, значит, он может находиться только в левой половине массива. Алгоритм отбрасывает правую половину и повторяет поиск, но уже в новых, суженных границах (от начала до элемента перед серединой).
* Искомый элемент больше. Если искомый элемент больше среднего, он может быть только в правой половине. Алгоритм отбрасывает левую и продолжает поиск в правой части (от элемента после середины и до конца).

1. Повторение: Шаги 2 и 3 повторяются до тех пор, пока элемент не будет найден или пока границы поиска не "схлопнутся" (левая граница станет больше правой). Если границы сошлись, а элемент так и не найден, значит, его в массиве нет.

Почему он так эффективен?

Главное преимущество бинарного поиска в том, что на каждом шаге он отбрасывает половину оставшихся данных. Из-за этого количество проверок растет очень медленно по сравнению с ростом размера массива. Например, в массиве из миллиона элементов он найдет нужный не более чем за 20 сравнений, в то время как обычный перебор (линейный поиск) в худшем случае потребовал бы миллион сравнений.

Асимптотическая сложность: O(log n).

Количество дополнительной памяти: O(1).

*3. Реализация*

typedef struct IEcoLab1\* IEcoLab1Ptr\_t;

typedef struct IEcoLab1VTbl {

/\* IEcoUnknown \*/

int16\_t (ECOCALLMETHOD \*QueryInterface)(/\* in \*/ IEcoLab1Ptr\_t me, /\* in \*/ const UGUID\* riid, /\* out \*/ voidptr\_t\* ppv);

uint32\_t (ECOCALLMETHOD \*AddRef)(/\* in \*/ IEcoLab1Ptr\_t me);

uint32\_t (ECOCALLMETHOD \*Release)(/\* in \*/ IEcoLab1Ptr\_t me);

/\* IEcoLab1 \*/

int16\_t (ECOCALLMETHOD \*Bsearchi)(/\* in \*/ IEcoLab1Ptr\_t me, /\* in \*/ const int\* arr, /\* in \*/ const int n, /\* in \*/ const int target, /\* out \*/ int\* index);

int16\_t (ECOCALLMETHOD \*Bsearchl)(/\* in \*/ IEcoLab1Ptr\_t me, /\* in \*/ const long long\* arr, /\* in \*/ const int n, /\* in \*/ const long long target, /\* out \*/ int\* index);

int16\_t (ECOCALLMETHOD \*Bsearchf)(/\* in \*/ IEcoLab1Ptr\_t me, /\* in \*/ const float\* arr, /\* in \*/ const int n, /\* in \*/ const float target, /\* out \*/ int\* index);

int16\_t (ECOCALLMETHOD \*Bsearchd)(/\* in \*/ IEcoLab1Ptr\_t me, /\* in \*/ const double\* arr, /\* in \*/ const int n, /\* in \*/ const double target, /\* out \*/ int\* index);

int16\_t (ECOCALLMETHOD \*Bsearchld)(/\* in \*/ IEcoLab1Ptr\_t me, /\* in \*/ const long double\* arr, /\* in \*/ const int n, /\* in \*/ const long double target, /\* out \*/ int\* index);

} IEcoLab1VTbl, \*IEcoLab1VTblPtr;

interface IEcoLab1 {

struct IEcoLab1VTbl \*pVTbl;

} IEcoLab1;

*4. Тесты*

Реализация

Чтобы проверить эффективность написанных методов, были написаны функции, генерирующие массивы от 1000 до 10000000 элементов каждого представленного типа данных. При этом каждый элемент массива равен его индексу.

**Пример генерации целочисленного массива:**

int\* GenerateIntArray(IEcoMemoryAllocator1\* pIMem, int size) {

size\_t i;

int\* arr = (int\*)pIMem->pVTbl->Alloc(pIMem, size \* sizeof(int));

for (i = 0; i < size; i++) {

arr[i] = i;

}

return arr;

}

Далее с помощью многочисленных запусков вычисляется среднее время отработки метода. Для замеров использовался QueryPerformanceCounter из windows.h, так как он позволяет вычислить наиболее точное время в наносекундах.

**Пример тестирования бинарного поиска для типа int:**

void TestBsearchInt(IEcoLab1\* pIEcoLab1, IEcoMemoryAllocator1\* pIMem) {

int\* arr;

int size\_i;

int launch\_i;

int size;

int index;

LARGE\_INTEGER start, end;

long long elapsed\_ticks;

double avg\_cpu\_time\_ns;

int result; // void\* result; (для stdlib bsearch)

printf("Binary search INT tests started\n");

for (size\_i = 0; size\_i < SIZES\_COUNT; size\_i++) {

size = g\_testSizes[size\_i];

arr = GenerateIntArray(pIMem, size);

QueryPerformanceCounter(&start);

for (launch\_i = 0; launch\_i < LAUNCH\_COUNT; launch\_i++) {

result = pIEcoLab1->pVTbl->Bsearchi(pIEcoLab1, arr, size, -1, &index);

if (result != ERR\_ECO\_SUCCESES) { // Данной проверки нет для stdlib bsearch

printf("Binary search INT tests failed\n");

pIMem->pVTbl->Free(pIMem, arr);

return;

}

}

QueryPerformanceCounter(&end);

assert(index == -1); // assert(index == NULL); (для stdlib bsearch)

elapsed\_ticks = end.QuadPart - start.QuadPart;

avg\_cpu\_time\_ns = ((double)elapsed\_ticks \* 1000000000.0 / g\_performanceFrequency.QuadPart) / LAUNCH\_COUNT;

printf("test %d elements: %.2f ns.\n", size, avg\_cpu\_time\_ns);

pIMem->pVTbl->Free(pIMem, arr);

}

printf("Binary search INT tests completed\n\n");

}

Стоит заметить, что в массиве все элементы неотрицательны, но в тестах в качестве target выступает -1. Это сделано для того, чтобы основной цикл в бинарном поиске отработал полностью и оценка производительности была более справедливой.

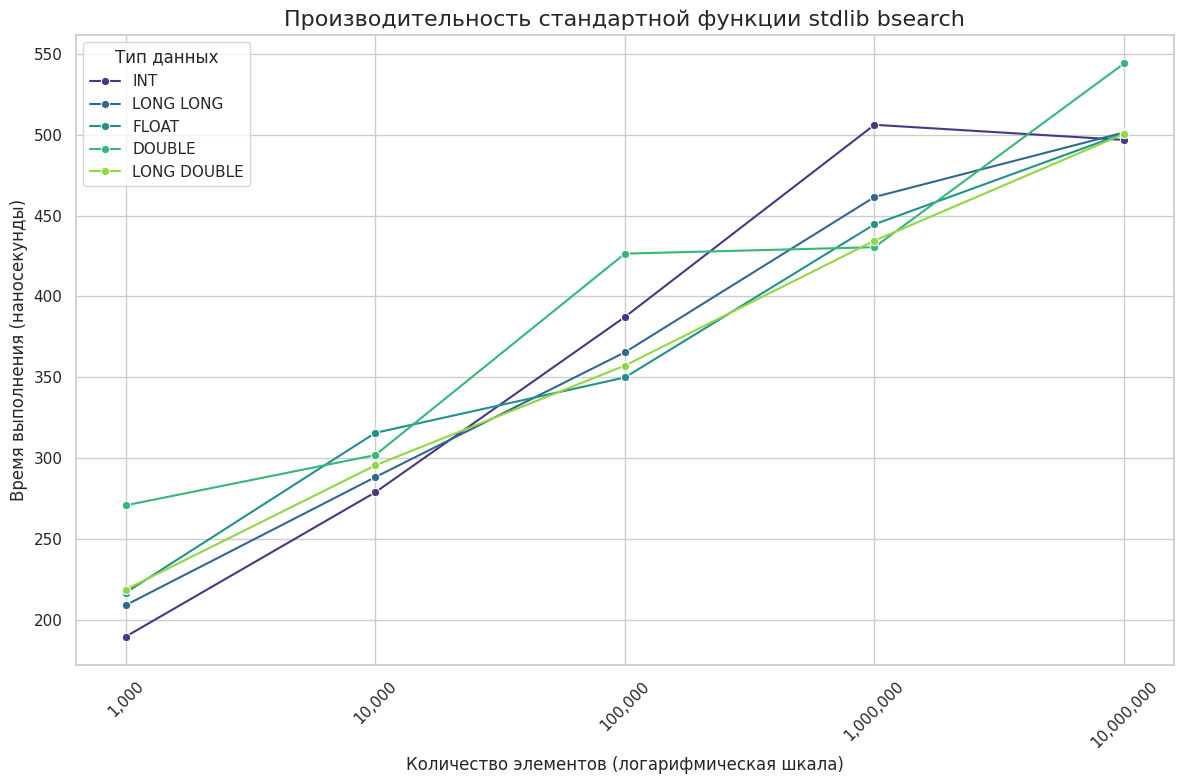
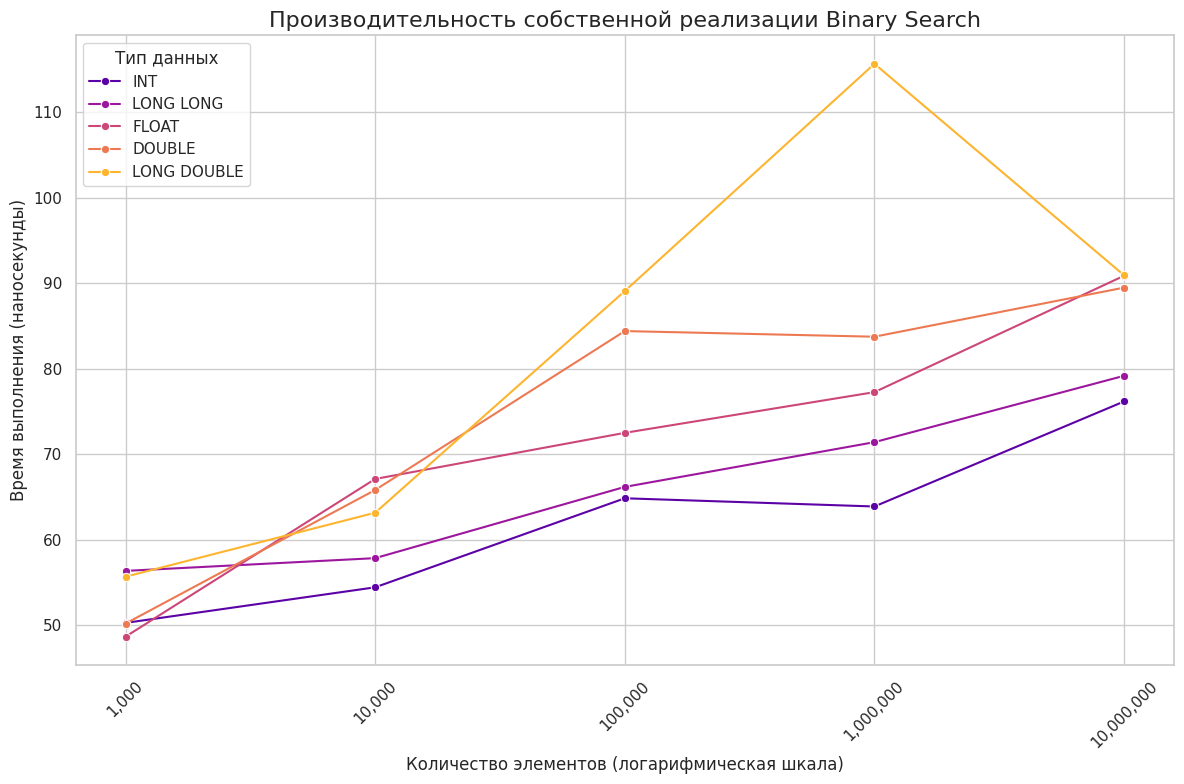
Результаты

Таблица 1: Собственная реализация Binary Search

| Тип данных | 1000 элементов (ns) | 10000 элементов (ns) | 100000 элементов (ns) | 1000000 элементов (ns) | 10000000 элементов (ns) |
| --- | --- | --- | --- | --- | --- |
| int | 50.27 | 54.43 | 64.84 | 63.88 | 76.16 |
| long long | 56.35 | 57.84 | 66.17 | 71.40 | 79.16 |
| float | 48.66 | 67.10 | 72.49 | 77.25 | 90.86 |
| double | 50.19 | 65.81 | 84.39 | 83.73 | 89.47 |
| long double | 55.67 | 63.15 | 89.07 | 115.64 | 90.91 |

Таблица 2: Стандартная функция stdlib bsearch

| Тип данных | 1000 элементов (ns) | 10000 элементов (ns) | 100000 элементов (ns) | 1000000 элементов (ns) | 10000000 элементов (ns) |
| --- | --- | --- | --- | --- | --- |
| int | 189.63 | 278.87 | 387.32 | 506.17 | 496.81 |
| long long | 209.10 | 288.27 | 365.48 | 461.50 | 501.44 |
| float | 216.84 | 315.64 | 349.90 | 444.61 | 500.84 |
| double | 270.83 | 301.92 | 426.50 | 430.48 | 544.09 |
| long double | 218.74 | 295.55 | 357.26 | 434.51 | 500.57 |



Собственная реализация бинарного поиска даже на огромных размерах массивов отрабатывает очень быстро - менее 100 наносекунд. При этом можно заметить, что хоть и немного, но методы, обрабатывающие массивы с нецелыми числами работают дольше. Stdlib реализация со всеми типами работает примерно одинаково (по крайней мере с теми, что рассматривались в данной работе). Время выполнения значительно дольше, но это связано с тем, что стандартный вариант работает с указателем на void, что приводит к многочисленным приведениям типов, но это делает функцию универсальной. Собственная реализация же может работать только с пятью типами данных, при этом для каждого типа - свой метод, что очень ограничивает в использовании.

**Вывод:** Собственная реализация работает быстрее, чем стандартная, но ограничена в использовании.