Санкт-Петербургский Государственный Университет

Факультет Прикладной математики – процессов управления

Лабораторная работа по курсу «Алгоритмы и анализ сложности»

Алгоритм интерполяционного поиска

Выполнил Тихонков Сергей

332 группа ФИИТ

Содержание:

Введение и краткое описание алгоритма:

Исследуемый алгоритм невольно используется нами в повседневной жизни. Представим такую ситуацию - мы хотим найти в документации к языку программирования описание определенного метода *addElem( )*. Известно, что методы в документации отсортированы в алфавитном порядке. Как мы будем действовать? Очевидно, что описание этого метода мы будем искать где-то в начале документации, а не в середине и точно не в конце.

В этот момент мы сильно сокращаем область поиска используя тот факт, что понятия *сильно меньше* и *немного меньше* различны. Этого не делают ни бинарный поиск, ни поиск Фибоначчи, что ставит интерполяционный поиск в выгодную позицию, по сравнению с ними. Но все ли так просто? Давайте разбираться математически.

Итак, пусть имеем отсортированный, без потери общности, по неубыванию, массив **a** размера **n**, то есть **a =** $[\boldsymbol{a_1} \leq \boldsymbol{a_2} \leq \ldots \leq \boldsymbol{a_n}]$.

Требуется найти позицию, на которой стоит элемент **key**, либо понять, что его нет в массиве. Поиск происходит подобно двоичному поиску, но вместо деления области поиска на две примерно равные части, интерполирующий поиск производит оценку новой области поиска по расстоянию между ключом и текущим значением элемента. Если известно, что **key** лежит между $\boldsymbol{a_l}$ и $\boldsymbol{a_r}$, то следующая проверка выполняется примерно на расстоянии $\frac{\boldsymbol{key-a_l}}{\boldsymbol{a_r-a_l}}$ $(\boldsymbol{r-l})$ от **l**. Откуда формула для разделительного элемента $\boldsymbol{m = l + \frac{key-a_l}{a_r-a_l}(r-l)}$. Если значения в массиве распределены равномерно случайным образом, то за один шаг алгоритм уменьшает количество проверяемых элементов с **n** до $\sqrt{\boldsymbol{n}}$. Следовательно, асимптотика в среднем $\boldsymbol{O(\log\log n)}$. Строго математически этот факт доказывается в пункте «*Математический анализ алгоритма*».

Можно считать, что алгоритм интерполяционного поиска в неявном виде имеет народное происхождение, а возник он именно в тот момент, когда люди научились составлять упорядоченные списки каких-либо объектов. Но строгое описание метода впервые было предложено **В.В.Петерсоном (W.W.Peterson) [IBM J.Res & Devel. 1 (1957), 131-132]**.

Сейчас алгоритм интерполяционного поиска активно применяют в случаях взаимодействия с внешней памятью, например, при поиске в документах, которые не помещаются в оперативную память. Так как обращение к внешней памяти сравнительно времязатратная операция, то сначала с помощью интерполяционного поиска сужают промежуток поиска, далее загружают фрагмент документа в оперативную память и применяют бинарный поиск.

Математический анализ алгоритма:

Лучший случай – искомый ключ найден сразу же, $\boldsymbol{O(1)}$

Худший случай – экспоненциальное возрастание элементов, $\boldsymbol{O(n)}$.

Средний случай – при условии, что значения элементов распределены равномерно, $\boldsymbol{O(\log\log n)}$.

Доказательство этих утверждений (через случайные величины) было приведено в статье **Yehoshua Perl, Alon Itai** и **Haim Avni** в журнале **Communications of the ACM** за **Июль 1978г**.

(Ссылка)

Характеристики входных данных:

На вход алгоритму подается отсортированный по неубыванию массив целых чисел **a** и искомое число **key.**

Единица измерения трудоемкости во время эксперимента – количество операций сравнения.

Входные данные: массив с размерами $\boldsymbol{n = 10, 10^2, 10^3, 5 \cdot 10^5, 5 \cdot 10^6, 10^7}$. Элементы массива и искомое число – случайные целые числа из диапазона $[\boldsymbol{0; n}]$. Для достоверности измерений для каждого **n** несколько раз генерируются массивы, а для каждого сгенерированного массива несколько раз запускается интерполяционный поиск случайного элемента. Для каждого **n** берется среднее арифметическое количество итераций по всем запускам исследуемого алгоритма.

Реализация генератора:

```
import numpy as np
def generate_sort_array( size=20, min_elem=0, max_elem=100):
    """
    param size: требуемый размер массива
    param min_elem: нижняя граница рандома
    param max_elem: верхняя граница рандома
    return: отсортированный массив размера size с целочисленными значениями от
min_elem до max_elem
    """
    return sorted( np.random.randint(min_elem, max_elem, size, dtype='int64'))
```

Используемая функция randint() случайно заполняет массив целыми значениями в заданном диапазоне, после чего массив сортируется методом sorted().

Реализация алгоритма:

```
def interpolation_search(array, key):
    """
    param array: массив, в котором производится поиск. Должен быть отсортирован.
    param key: значение элемента, который ищется в массиве array
    return: tuple(pos, count), где
        pos: позиция элемента key в массиве array или -1 если key не содержится в array
        count: количество операций сравнения
    """
    left = 0  # левая граница поиска (будем считать, что элементы массива нумеруются с нуля)
    right = len(array) - 1  # правая граница поиска
    count = 0 # счетчик операций сравнения

    while array[left] < key and key < array[right]:
        count += 2
        # индекс элемента, с которым будем проводить сравнение
        mid = left + (key - array[left]) * (right - left) // (array[right] - array[left])
        if array[mid] < key:
            count += 1
            left = mid + 1
        elif array[mid] > key:
            count += 1
            right = mid - 1
        else:
            return mid, count

    if array[left] == key:
        return left, count + 1
    elif array[right] == key:
        return right, count + 2
    else:
        return -1, count+2 # если такого элемента в массиве нет
```

Итог вычислительного эксперимента:

По результатам вычислительного эксперимента наглядно видно, что доказанная асимптотика $\boldsymbol{O(\log\log n)}$ подтверждается. Ниже представлены таблица и график результатов эксперимента. Исходный код вычислительного эксперимента приведен (среди прочего) в сопроводительном файле ***interpolation_search.py***

Размер массива **n**	Среднее количество операций сравнения	Реальное значение функции $f(n) = 3 * \log_2 \log_2 n$
10	2.7	5.1
100	5.5	8.1
1000	8.06	9.9
500000	10.875	12.7
5000000	11.62	13.4
10000000	11.8	13.6

Таблица 1. Результаты вычислительного эксперимента.

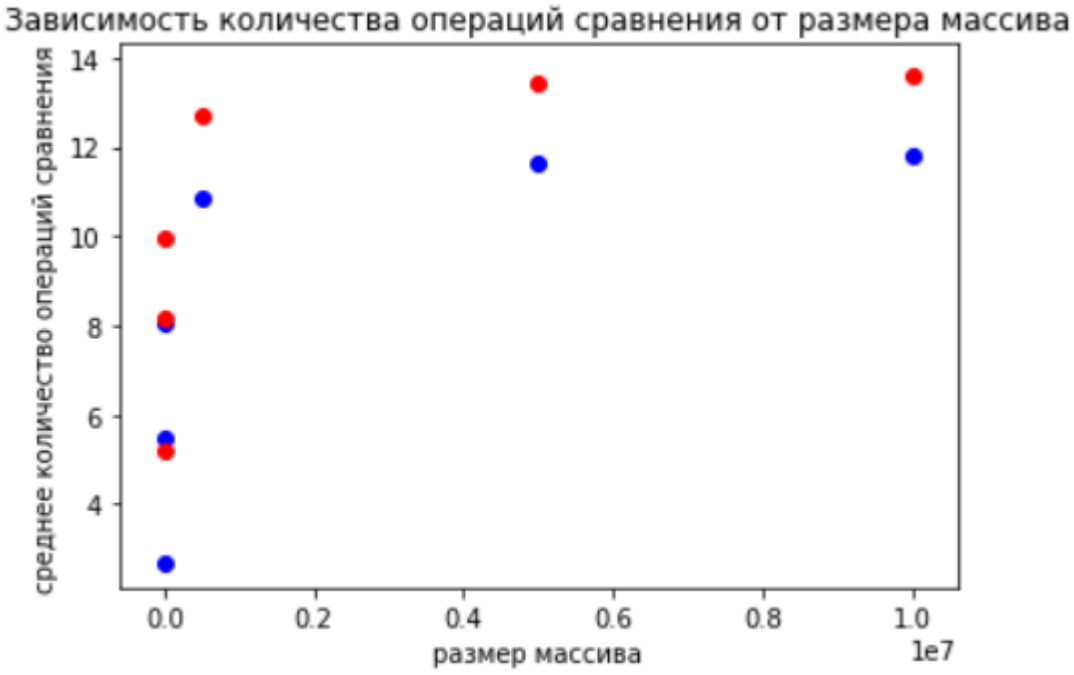

Рисунок 1. График зависимости количества операций сравнения от размера входных данных

Красные точки – реальное значение функции, **синие** – экспериментальные оценки.

Рассмотрим отношение трудоемкостей при увеличении входных данных в два раза

n	$T(n)$	$T(2n)$	$\frac{T(2n)}{T(n)}$
100	5.5	6.5	1,18
5000000	11.62	11.8	1,015

Таблица 2. Отношение трудоемкостей при увеличении входных данных в два раза

Причем, $\lim\limits_{n\to\infty} \frac{T(2n)}{T(n)} = \lim\limits_{n\to\infty} \frac{3\log\log 2n}{3\log\log n} = 1$. Полученные результаты еще раз доказывают правильность асимптотической оценки $\mathrm{O}(\log\log n)$.

Характеристики вычислительной среды:

Процессор: Intel Core i5-6300HQ CPU 2.30GHz

Оперативная память: 8,00 ГБ

Тип системы: 64-разрядная операционная система

IDE: Jupyter Notebook

Список литературы:

Основной источник информации об алгоритме книга **Donald E. Knuth "The Art of Computer Programming" Volume 3 – Sorting and Searching.**

Математическое доказательство асимптотики исследуемого алгоритма - статья **Yehoshua Perl, Alon Itai** и **Haim Avni** в журнале **"Communications of the ACM"** за **Июль 1978г** .(Ссылка)