Практическая работа № 1

Тема: «Алгоритмы поиска»

Цель работы: изучить алгоритмы поиска

Поиск является одним из наиболее часто встречаемых действий в программировании. Существует множество различных алгоритмов поиска, которые принципиально зависят от способа организации данных. У каждого алгоритма поиска есть свои преимущества и недостатки. Поэтому важно выбрать тот алгоритм, который лучше всего подходит для решения конкретной задачи.

Поставим задачу поиска в линейных структурах. Пусть задано множество данных, которое описывается как массив, состоящий из некоторого

количества элементов. Проверим, входит ли заданный ключ в данный массив.

Если входит, то найдем номер этого элемента массива, то есть, определим первое вхождение заданного ключа (элемента) в исходном массиве.

Таким образом, определим общий алгоритм поиска данных:

Шаг 1. Вычисление элемента, что часто предполагает получение значения элемента, ключа элемента и т.д.

Шаг 2. Сравнение элемента с эталоном или сравнение двух элементов (взависимости от постановки задачи).

Шаг 3. Перебор элементов множества, то есть прохождение по элементам массива.

Основные идеи различных алгоритмов поиска сосредоточены в методах перебора и стратегии поиска.

Рассмотрим основные алгоритмы поиска в линейных структурах более подробно.

Последовательный (линейный) поиск

Последовательный (линейный) поиск – это простейший вид поиска заданного элемента на некотором множестве, осуществляемый путем последовательного сравнения очередного рассматриваемого значения с искомым до тех пор, пока эти значения не совпадут

Идея этого метода заключается в следующем. Множество элементов просматривается последовательно в некотором порядке, гарантирующем, что будут просмотрены все элементы множества (например, слева направо). Если в ходе просмотра множества будет найден искомый элемент, просмотр прекращается с положительным результатом; если же будет просмотрено все множество, а элемент не будет найден, алгоритм должен выдать отрицательный результат.

Алгоритм последовательного поиска

Шаг 1. Полагаем, что значение переменной цикла i=0.

Шаг 2. Если значение элемента массива x[i] равно значению ключа key, то возвращаем значение, равное номеру искомого элемента, и алгоритм завершает работу. В противном случае значение переменной цикла увеличивается на единицу i=i+1.

Шаг 3. Если i<k, где k – число элементов массива x, то выполняется Шаг

2, в противном случае – работа алгоритма завершена и возвращается значение равное -1.

При наличии в массиве нескольких элементов со значением key данный алгоритм находит только первый из них (с наименьшим индексом).

Листинг 1. Реализация линейного поиска на языке Python.

Для списка из n элементов лучшим случаем будет тот, при котором искомое значение равно первому элементу списка и требуется только одно сравнение. Худший случай будет тогда, когда значения в списке нет (или оно находится в самом конце списка), в случае чего необходимо n сравнений.

Если искомое значение входит в список k раз и все вхождения равновероятны, то ожидаемое число сравнений

$$\begin{cases} n & k=0 \\ \frac{n+1}{k+1} & 1 \le k \le n \end{cases}$$

Например, если искомое значение встречается в списке один раз, и все вхождения равновероятны, то среднее число сравнений равно . Однако, если известно, что оно встречается один раз, то достаточно n — 1 сравнений, и среднее число сравнений будет равняться

$$\frac{(n+2)(n-1)}{2n}$$

(для n = 2 это число равняется 1, что соответствует одной конструкции if-then-else).

В любом случае вычислительная сложность алгоритма O(n).

Обычно линейный поиск очень прост в реализации и применим, если список содержит мало элементов, либо в случае одиночного поиска в неупорядоченном списке.

Если предполагается поиск в одном и том же списке большое число раз, то часто имеет смысл предварительной обработки списка, например, сортировки и последующего использования бинарного поиска, либо построения какой-либо эффективной структуры данных для поиска. Частая модификация списка также может оказывать влияние на выбор дальнейших действий, поскольку делает необходимым процесс перестройки структуры.

На рисунке 1 представлена зависимость времени поиска элемента от длины массива. Поиск производился уже в отсортированном массиве

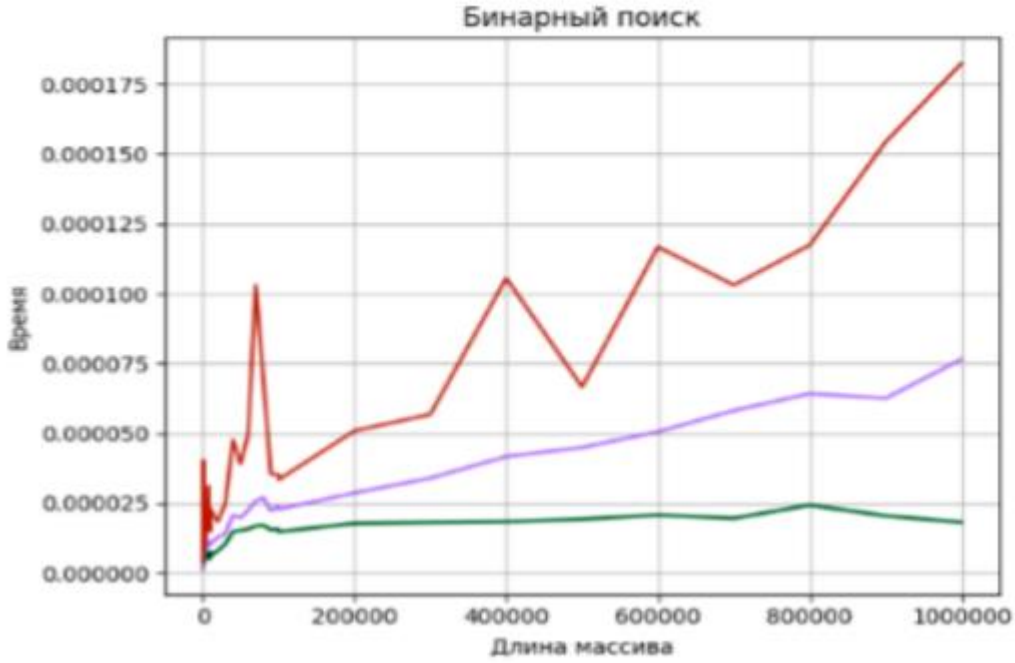


Рисунок 1 - Зависимость времени от размера массива

Нижняя линия на рисунке один это минимальное время поиска за 50 повторов для каждого значения (для каждого повтора был заново сгенерирован массив той же длины), средняя это среднеарифметическое времени поиска по результатам 50 повтороний.

Диаграмма деятельности для линейного поиска представлена на рисунке 2.

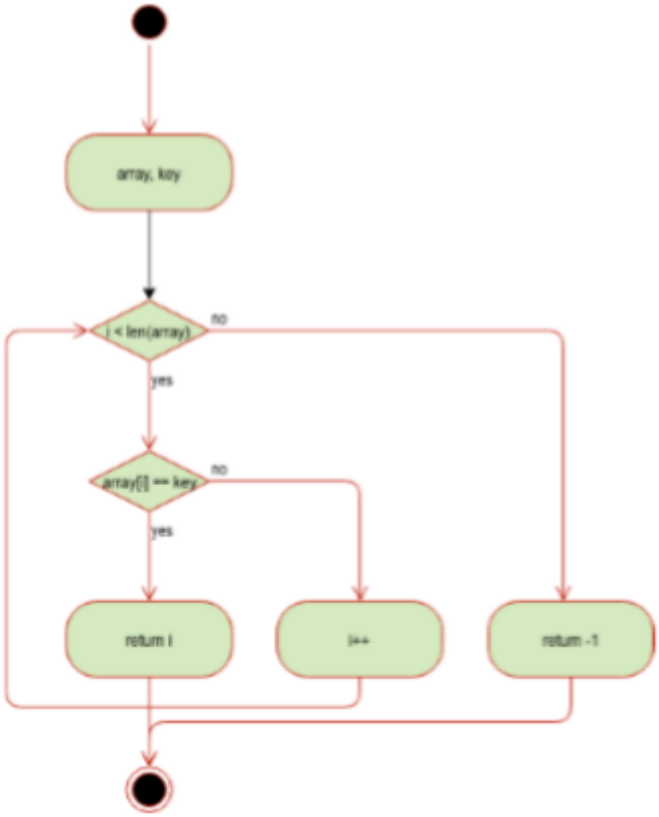


Рисунок 2 - Линейный поиск

Двоичный (бинарный) поиск (также известен как метод деления пополам или дихотомия) — классический алгоритм поиска элемента в отсортированном массиве (векторе), использующий дробление массива на половины. Используется в информатике, вычислительной математике и математическом программировании.

Частным случаем двоичного поиска является метод бисекции, который применяется для поиска корней заданной непрерывной функции на заданном отрезке.

1. Определение значения элемента в середине структуры данных. Полученное значение сравнивается с ключом.

2. Если ключ меньше значения середины, то поиск осуществляется в первой половине элементов, иначе — во второй.

3. Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.

4. Процесс продолжается до тех пор, пока не будет найден элемент со значением ключа или не станет пустым интервал для поиска

Несмотря на то, что код достаточно прост, в нём есть несколько ловушек.

1. Что будет, если first и last по отдельности умещаются в свой тип, а first+last — нет? Если теоретически возможны массивы столь большого размера, приходится идти на ухищрения:

1. Использовать код first + (last - first) / 2, который точно не приведёт к переполнениям (то и другое — неотрицательные целые числа).

1. Если first и last — указатели или итераторы, такой код единственно правильный. Преобразование в uintptr_t и дальнейший расчёт в этом типе нарушает абстракцию, и нет гарантии, что результат останется корректным указателем. Разумеется, чтобы сохранялась сложность алгоритма, нужны быстрые операции «итератор+число → итератор», «итератор−итератор → число».

2. Если first и last — типы со знаком, провести расчёт в беззнаковом типе: ((unsigned)first + (unsigned)last) / 2. В Java соответственно: (first + last) >>> 1.

3. Написать расчёт на ассемблере, с использованием флага переноса. Что-то наподобие add eax, b; rcr eax, 1. А вот длинные типы использовать нецелесообразно, first + (last - first) / 2 быстрее.

2. В двоичном поиске часты ошибки на единицу. Поэтому важно протестировать такие случаи: пустой массив (n=0), ищем отсутствующее значение (слишком большое, слишком маленькое и гдето в середине), ищем первый и последний элемент. Не выходит ли алгоритм за границы массива? Не зацикливается ли?

3. Иногда требуется, чтобы, если x в цепочке существует в нескольких экземплярах, находило не любой, а обязательно первый (как вариант: последний; либо вообще не x, а следующий за ним элемент).[2] Код на Си в такой ситуации находит первый из равных, более простой код на Си++ — какой попало.

Учёный Йон Бентли утверждает, что 90 % студентов, разрабатывая двоичный поиск, забывают учесть какое-либо из этих требований. И даже в

код, написанный самим Йоном и ходивший из книги в книгу, вкралась ошибка:
код не стоек к переполнениям.

1. Широкое распространение в информатике применительно к поиску в структурах данных. Например, поиск в массивах данных осуществляется по ключу, присвоенному каждому из элементов массива (в простейшем случае сам элемент является ключом).

2. Также его применяют в качестве численного метода для нахождения приближённого решения уравнений (см. Метод бисекции).

3. Метод используется для нахождения экстремума целевой функции и в этом случае является методом условной одномерной оптимизации. Когда функция имеет вещественный аргумент, найти решение с точностью до ε можно за время . Когда аргумент дискретен, и изначально лежит на отрезке длины N, поиск решения займёт 1+log2N времени. Наконец, для поиска экстремума, скажем, для определённости минимума, на очередном шаге отбрасывается тот из концов рассматриваемого отрезка, значение в котором максимально.

Листинг 2. Двоичный (бинарный поиск) на Python с возратом первого вхождения.

```
def binar_search(array, key):
minimum = 0
maximum = len(array) - 1
ret = 0
while minimum <= maximum:
mid = (maximum + minimum) // 2
if key < array[mid]:
maximum = mid - 1
elif key > array[mid]:
minimum = mid + 1
else:
ret = mid
break
while ret > 0 and array[ret - 1] == key:
```

```
ret -= 1
if array[ret] == key:
return ret else:
return -1
```

На рисунке 2 представлена зависимость времени поиска элемента от длины массива. Поиск производился уже в отсортированном массиве.

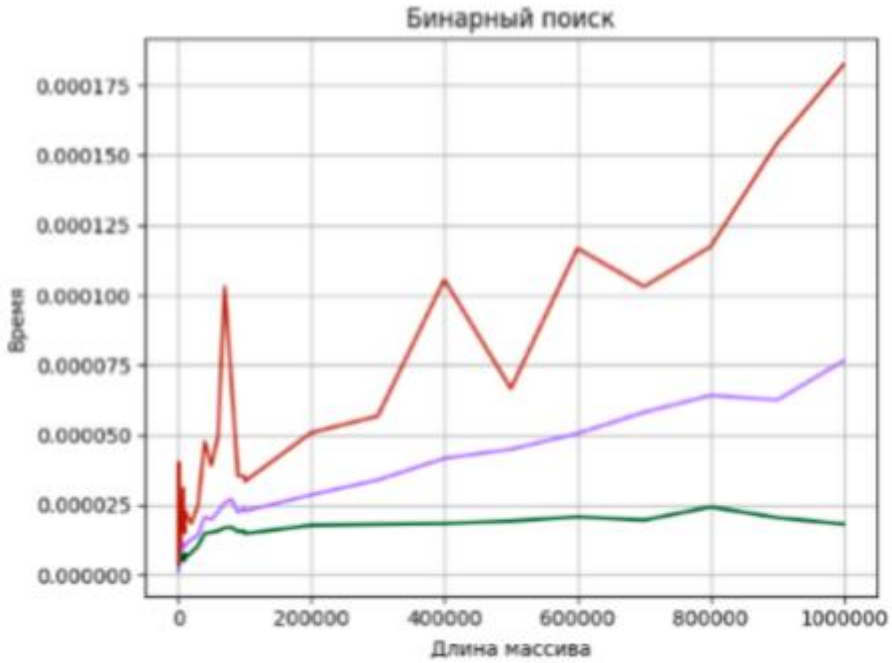


Рисунок 3 - Зависимость времени поиска от длины массива для бинарного поиска

Диаграмма деятельности для бинарного поиска представлена на рисунке 4.

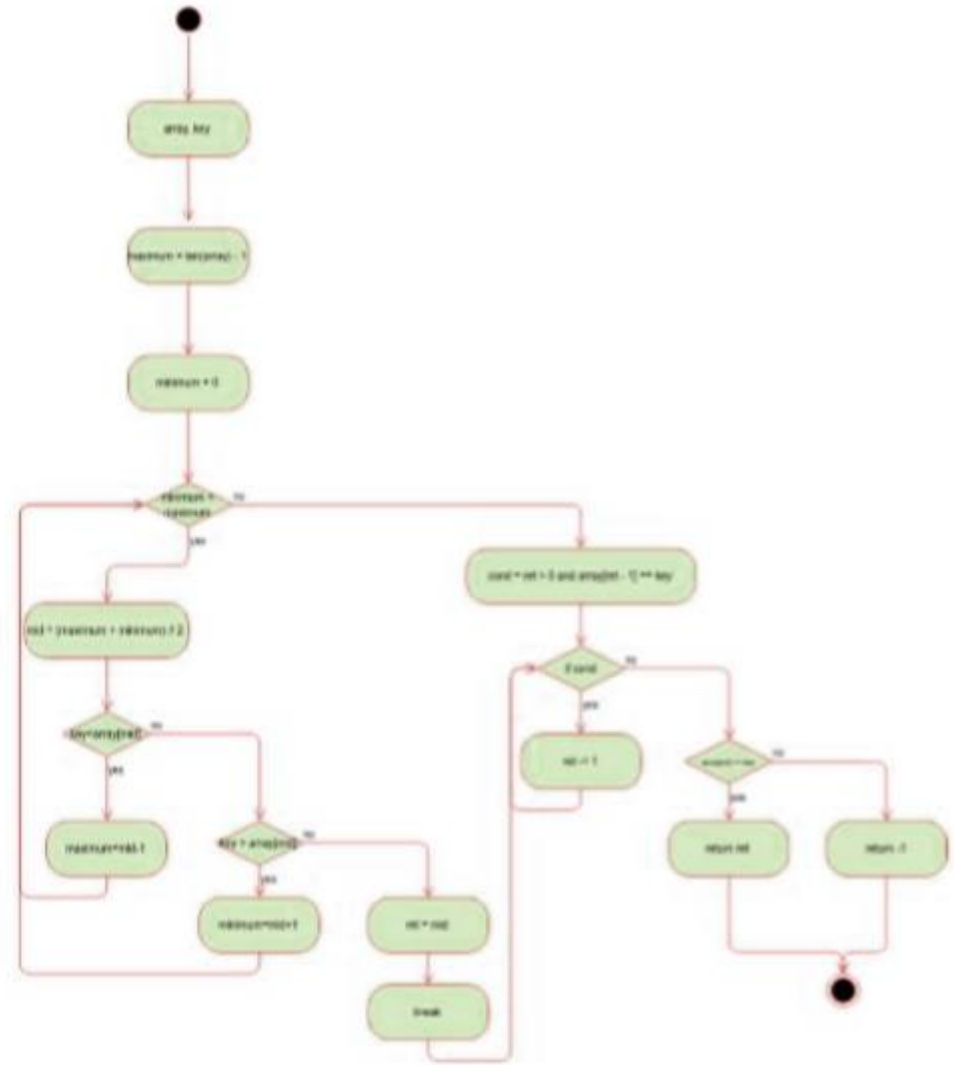


Рисунок 4 - Бинарный поиск

Интерполяционный поиск (интерполирующий поиск) основан на принципе поиска в телефонной книге или, например, в словаре. Вместо сравнения каждого элемента с искомым, как при линейном поиске, данный алгоритм производит предсказание местонахождения элемента: поиск происходит подобно двоичному поиску, но вместо деления области поиска на две части, интерполирующий поиск производит оценку новой области поиска

по расстоянию между ключом и текущим значением элемента. Другими словами, бинарный поиск учитывает лишь знак разности между ключом и текущим значением, а интерполирующий ещё учитывает и модуль этой разности и по данному значению производит предсказание позиции

следующего элемента для проверки. В среднем интерполирующий поиск производит log(log(N)) операций, где N есть число элементов, среди которых производится поиск. Число необходимых операций зависит от равномерности
распределения значений среди элементов. В плохом случае (например, когда значения элементов экспоненциально возрастают) интерполяционный поиск может потребовать до O(N) операций.

На практике интерполяционный поиск часто быстрее бинарного, так как с вычислительной стороны их отличают лишь применяемые арифметические операции: интерполирование — в интерполирующем поиске и деление на два — в двоичном, а скорость их вычисления отличается незначительно, с другой стороны интерполирующий поиск использует такое принципиальное свойство
данных, как однородность распределения значений. Ключом может быть не только номер, число, но и, например, текстовая строка, тогда становится понятна аналогия с телефонной книгой: если мы ищем имя в телефонной книге, начинающееся на «А», следовательно, нужно искать его в начале, но никак не в середине. В принципе, ключом может быть всё что угодно, так как те же строки, например, запросто кодируются посимвольно, в простейшем случае символ можно закодировать значением от 1 до 33 (только русские символы) или, например, от 1 до 26 (только латинский алфавит) и т. д.

Интерполяция может производиться на основе функции, аппроксимирующей распределение значений, либо набора кривых, выполняющих аппроксимацию на отдельных участках. В этом случае поиск может завершиться за несколько проверок. Преимущества этого метода состоят в уменьшении запросов на чтение медленной памяти (такой, как, например, жесткий диск), если запросы происходят часто. Такой подход становится похожим на частный случай поиска с использованием хештаблицы.

Часто анализ и построение аппроксимирующих кривых не требуется,

показательный случай здесь — когда все элементы отсортированы по возрастанию. В таком списке минимальное значение будет по индексу 1, а максимальное по индексу N. В этом случае аппроксимирующую кривую можно принять за прямую и применять линейную интерполяцию. Листинг 3. Интерполяционный поиск с возратом первого совпадающего элемента в отсортированном массиве на Python.

```
minimum = 0
maximum = len(array) - 1
ret = 0
while array[minimum] < key < array[maximum]: mid = int(minimum + (maximum - minimum) * (key -
array[minimum]) / (array[maximum] - array[minimum]))
if array[mid] == key:
ret = mid
break
elif array[mid] > key:
maximum = mid - 1
else:
minimum = mid + 1
if array[minimum] == key:
ret = minimum
if array[maximum] == key:
ret = maximum
while ret > 0 and array[ret - 1] == key:
ret -= 1
if array[ret] == key:
return ret
else:
return -1
```

На рисунке 3 представлена зависимость времени поиска элемента от длины массива. Поиск производился уже в отсортированном массиве. На рисунке 4 представлена диаграмма деятельности для интерполяционного поиска.

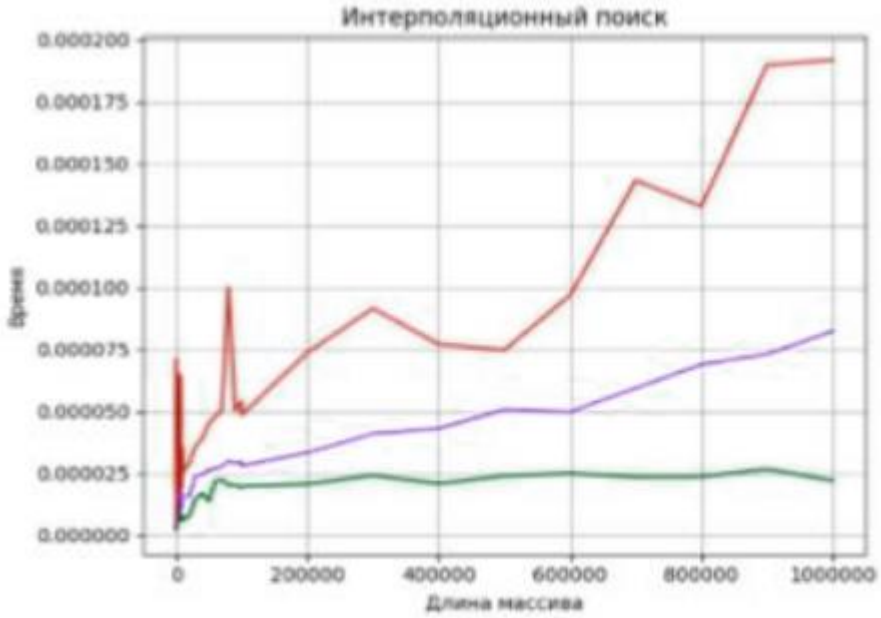


Рисунок 5 - Зависимость времени поиска от длины массива для интерполяционного поиска

Рисунок 6 - Интерполяционный поиск

Как и двоичный поиск, Jump Search - это алгоритм поиска отсортированных массивов. Основная идея заключается в том, чтобы проверить меньшее количество элементов (чем линейный поиск), прыгнув вперед фиксированными шагами или пропустив некоторые элементы вместо поиска всех элементов.

Например, предположим, что у нас есть массив arr[] размера n и блок (для прыжка) размера m. Затем мы ищем по индексам arr[0], arr[m], arr[2m]....arr[km] и так далее. Найдя интервал (arr[km] < x < arr[(k+1)m]), выполняем линейную операцию поиска из индекса km, чтобы найти элемент x.

Рассмотрим следующий массив: (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610). Длина массива 16. Поиск прыжка найдет значение 55 со следующими шагами, предполагая, что размер перепрыгиваемого блока 4.

ШАГ 1: Перепрыгнуть с индекса 0 на индекс 4;

ШАГ 2: Перепрыгнуть с индекса 4 на индекс 8;

ШАГ 3: Перепрыгнуть с индекса 8 на индекс 12;

ШАГ 4: Так как элемент в индексе 12 больше 55, мы перепрыгнем назад на шаг, чтобы прийти к индексу 8.

ШАГ 5: Выполним линейный поиск с индекса 8, чтобы получить элемент 55.

Каков оптимальный размер блока, который нужно пропустить?

В худшем случае, мы должны сделать nm прыжков, и если последнее проверенное значение больше, чем элемент, который нужно искать, мы выполним m-1 сравнения больше для линейного поиска. Поэтому общее число

сравнений в наихудшем случае будет ((nm) + m-1). Значение функции ((nm) + m-1) будет минимальным, когда m = √n. Поэтому наилучший размер шага - m = √n.

Важные моменты:

1. Работают только отсортированные массивы.Оптимальный размер

перепрыгиваемого блока (√ n). Это усложняет поиск по времени прыжка O(√ n).

2. Временная сложность Прыжкового поиска находится между линейным поиском ((O(n)) и двоичным поиском (O (Log n)).

3. Двоичный поиск лучше, чем Прыжковый поиск, но у Прыжкового поиска есть преимущество, заключающееся в том, что мы возвращаемся назад

только один раз (Двоичный поиск может потребовать до прыжков O(Log n), рассмотрим ситуацию, когда элемент, подлежащий поиску, является наименьшим элементом или меньшим, чем наименьший). Поэтому в системе, где двоичный поиск является дорогостоящим, мы используем Jump Search.

Листинг 4. Jump Search на Python.

```
def jump_search(array, key):
length = len(array)
jump_step = int(math.sqrt(length))
previous: int = 0
while array[min(jump_step, length) - 1] < key:
previous = jump_step
jump_step += int(math.sqrt(length))
if previous >= length:
return -1
while array[previous] < key:
previous += 1
if previous == min(jump_step, length):
return -1 if array[previous] == key:
return previous
return -1
```

На рисунке 4 представлена зависимость времени поиска элемента от длины массива. Поиск производился уже в отсортированном массиве. На рисунке 5 представлена диаграмма деятельности.

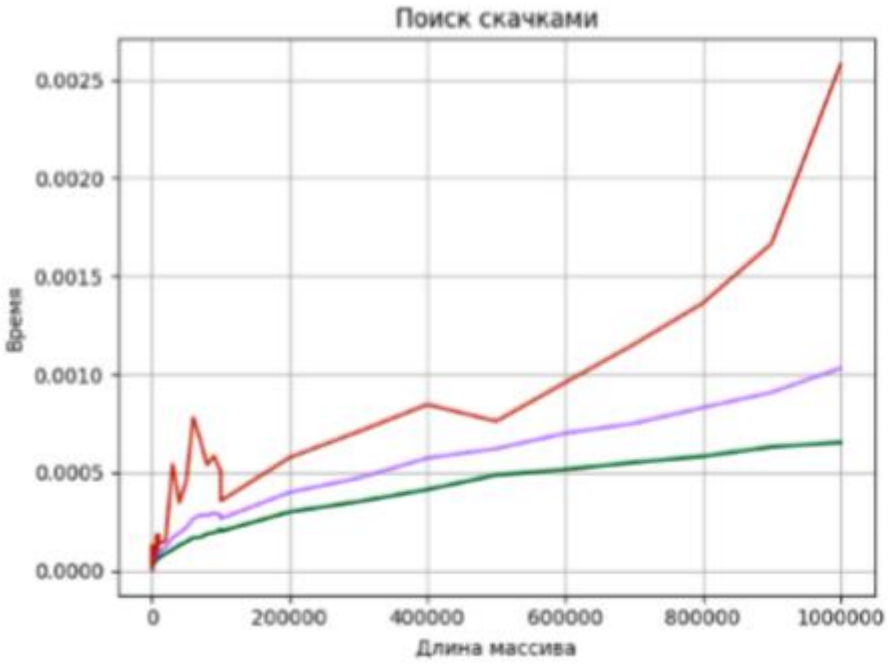


Рисунок 5 - Зависимость времени поиска от длины массива для интерполяционного поиска

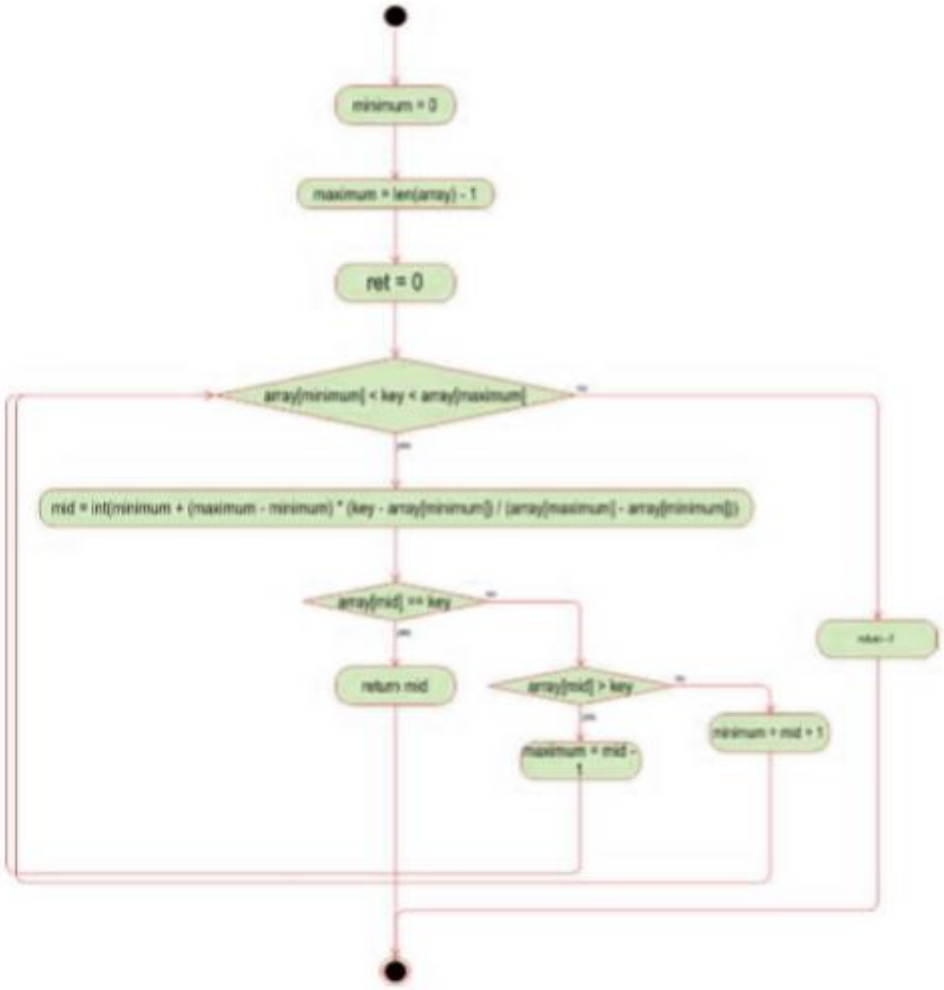


Рисунок 6 - Интерполяционный поиск

Как и двоичный поиск, Jump Search - это алгоритм поиска отсортированных массивов. Основная идея заключается в том, чтобы проверить меньшее количество элементов (чем линейный поиск), прыгнув вперед фиксированными шагами или пропустив некоторые элементы вместо поиска всех элементов.

Например, предположим, что у нас есть массив arr[] размера n и блок (для прыжка) размера m. Затем мы ищем по индексам arr[0], arr[m], arr[2m]....arr[km] и так далее. Найдя интервал (arr[km] < x < arr[(k+1)m]), выполняем линейную операцию поиска из индекса km, чтобы найти элемент x.

Рассмотрим следующий массив: (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610). Длина массива 16. Поиск прыжка найдет значение 55 со следующими шагами, предполагая, что размер перепрыгиваемого блока 4.

ШАГ 1: Перепрыгнуть с индекса 0 на индекс 4;

ШАГ 2: Перепрыгнуть с индекса 4 на индекс 8;

ШАГ 3: Перепрыгнуть с индекса 8 на индекс 12;

ШАГ 4: Так как элемент в индексе 12 больше 55, мы перепрыгнем назад на шаг, чтобы прийти к индексу 8.

ШАГ 5: Выполним линейный поиск с индекса 8, чтобы получить элемент 55.

Каков оптимальный размер блока, который нужно пропустить?

В худшем случае, мы должны сделать nm прыжков, и если последнее проверенное значение больше, чем элемент, который нужно искать, мы выполним m-1 сравнения больше для линейного поиска. Поэтому общее число

сравнений в наихудшем случае будет ((nm) + m-1). Значение функции ((nm) + m-1) будет минимальным, когда m = √n. Поэтому наилучший размер шага - m = √n.

Важные моменты

1. Работают только отсортированные массивы.Оптимальный размер перепрыгиваемого блока (√ n). Это усложняет поиск по времени прыжка O(√ n).

2. Временная сложность Прыжкового поиска находится между линейным поиском ((O(n)) и двоичным поиском (O (Log n)).

3. Двоичный поиск лучше, чем Прыжковый поиск, но у Прыжкового поиска есть преимущество, заключающееся в том, что мы возвращаемся назад

только один раз (Двоичный поиск может потребовать до прыжков O(Log n), рассмотрим ситуацию, когда элемент, подлежащий поиску, является наименьшим элементом или меньшим, чем наименьший). Поэтому в системе, где двоичный поиск является дорогостоящим, мы используем Jump Search.

Листинг 4. Jump Search на Python.

```
def jump_search(array, key):
length = len(array)
jump_step = int(math.sqrt(length))
previous: int = 0
while array[min(jump_step, length) - 1] < key:
previous = jump_step
jump_step += int(math.sqrt(length))
if previous >= length:
return -1
while array[previous] < key:
previous += 1
if previous == min(jump_step, length):
return -1 if array[previous] == key:
return previous
return -1
```

На рисунке 4 представлена зависимость времени поиска элемента от длины массива. Поиск производился уже в отсортированном массиве. На

рисунке 5 представлена диаграмма деятельности.

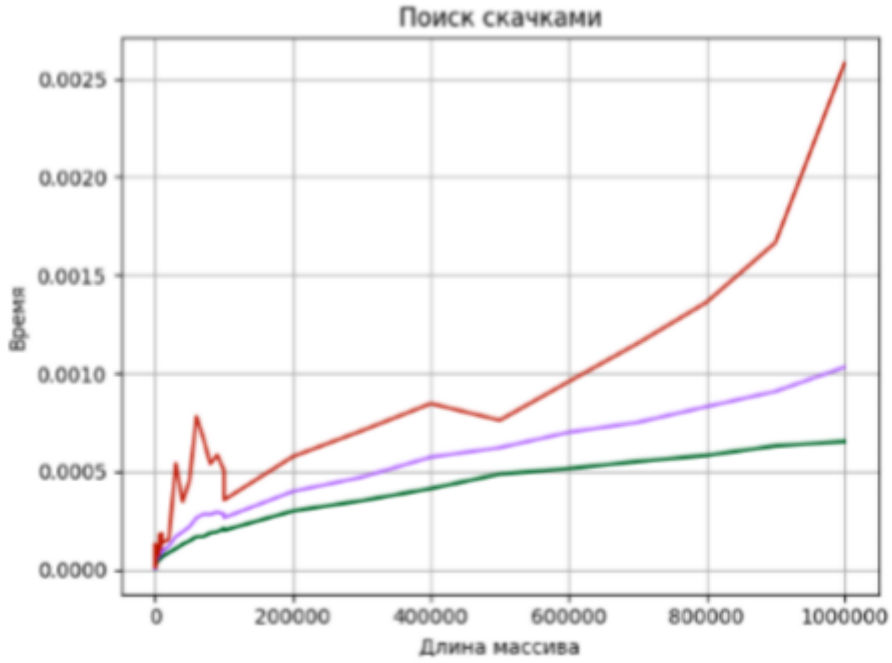


Рисунок 7 - Зависимость времени поиска от длины массива для Jump Search

Рисунок 8 - Поиск скачками

Поиск Фибоначчи — это метод сравнения, использующий числа Фибоначчи для поиска элемента в отсортированном массиве.

Сходства с бинарным поиском:

1. Работает для отсортированных массивов
2. Алгоритм разделяй и властвуй
3. Имеет лог n времени сложности.

Отличия от бинарного поиска :

1. Поиск Фибоначчи делит данный массив на неравные части
2. Бинарный поиск использует оператор деления для разделения диапазона. Поиск Фибоначчи не использует /, но использует + и -. Оператор деления может быть дорогостоящим на некоторых процессорах.
3. Поиск Фибоначчи исследует относительно более близкие элементы в последующих шагах. Поэтому, когда входной массив большой, который не помещается в кэш процессора или даже в оперативную память, поиск по Фибоначчи может быть полезен

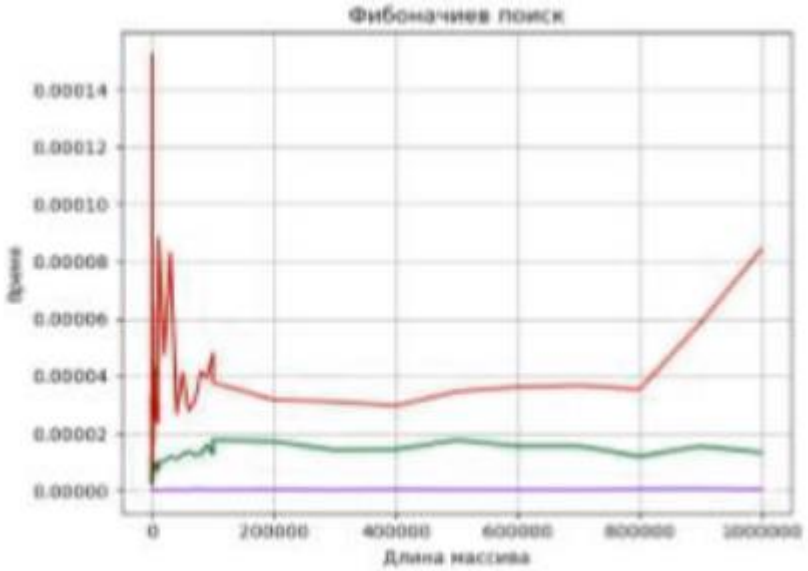
Код реализации приведен в Листинге 5, диаграмма деятельности в рисунке 9 и график работы в рисунке 10.

Листинг 5. Реализация на Python

```
def fibMonaccianSearch(arr, x, n):
 fibMMm2 = 0 # (m-2)'th Fibonacci No.
 fibMMm1 = 1 # (m-1)'th Fibonacci No.
 fibM = fibMMm2 + fibMMm1 # m'th Fibonacci
 while (fibM < n):
 fibMMm2 = fibMMm1
 fibMMm1 = fibM
 fibM = fibMMm2 + fibMMm1
 offset = -1;
 while (fibM > 1):
 i = min(offset+fibMMm2, n-1)
 if (arr[i] < x):
```

```
 fibM = fibMMm1
 fibMMm1 = fibMMm2
fibMMm2 = fibM - fibMMm1
 offset = i
 elif (arr[i] > x):
 fibM = fibMMm2
 fibMMm1 = fibMMm1 - fibMMm2
 fibMMm2 = fibM - fibMMm1
 else :
 return i
 if(fibMMm1 and arr[offset+1] == x):
 return offset+1;
return -1
```

Рисунок 9-Фибоначиев поиск



Вывод: в ходе выполнения практической работы были изучены алгоритмы поиск