Лекция №12.

Тема: **Организация поиска данных.**

Поиск данных. Цель поиска. Критерии эффективности алгоритмов поиска. Поиск данных в последовательности.

Ключевые слова: алгоритм поиска; критерий эффективности поиска; линейный поиск; бинарный поиск; прямой поиск подстроки.

Keywords: search algorithms; criteria of search; linear search; binary search algorithm; naïve string search algorithm.

1 Поиск данных

Поиск элемента (ов) с заданным (и) значением (ями) в данных – одна из наиболее часто встречающихся задач в программировании (см. http://en.wikipedia.org/wiki/Search_algorithm). Несмотря на то, что алгоритмы поиска являются исторически одними из первых алгоритмов, созданных человеком, актуальность создания новых алгоритмов поиска только возрастает. В разработку алгоритмов поиска вовлечены не только ученые, но и целые корпорации (например, Google, см. http://en.wikipedia.org/wiki/Google_Search). Современные алгоритмы поиска работают с нечеткими территориально распределенными данными огромной размерности, а результаты такого поиска применяются в различных областях.

2 Цель поиска

Целью алгоритмов поиска, в узком смысле слова, является ответ на вопрос, присутствуют ли в данном наборе данных, некоторые данные, свойства которых нам известны заранее. В широком смысле слова, результат поиска – описание места вхождения искомых данных в наборе данных. Если таких мест вхождения – несколько, то результатом поиска может быть описание всех мест вхождения, или первого места, или последнего места в заданной последовательности данных. Рассматривая ниже алгоритмы поиска, будем предполагать, что нас интересует лишь первое вхождение искомого элемента.

3 Факторы, которые необходимо учитывать в процессе поиска

На процесс поиска оказывают влияние и параметры искомых данных и параметры данных, в которых выполняется поиск. Причем, чем меньше имеется дополнительной информации о данных, тем больше уверенность в том, что искомый элемент должен сравниваться с каждым элементом данных, в которых выполняется поиск. Наличие дополнительной информации позволяет за один шаг поиска уменьшить область поиска больше, чем на один элемент.

Перечислим факторы, влияющие на выбор алгоритма поиска:

1. Структура данных – данные могут быть представлены либо простым, либо сложным типом. Например, может выполняется поиск некоторого целого числа в списке, или - некоторой подстроки в строке, или данных какой-то записи, как элемента «сильно ветвящегося дерева».
2. Объем данных – поиск данных в небольших последовательностях может быть выполнен с помощью простого алгоритма поиска. Для ускорения поиска в больших объемах данных обычно необходима дополнительная информация. Например, если в численном большом массиве необходимо выполнить поиск позиции вхождения конкретного элемента, то предварительно отсортировав массив, можно ускорить проверку наличия такого элемента. Однако

хранение при этом информации об исходных позициях вхождения элементов, потребует и дополнительного времени, и дополнительной памяти. Если же данные изначально хранятся и пополняются в отсортированном виде (частично отсортированном виде), то знание этой дополнительной информации существенно упрощает задачу поиска.

3. Способ хранения данных – данные могут храниться в оперативной или долговременной памяти, причем файлы данных могут быть прямого или последовательного доступа. Обычно способ хранения существенно влияет на выбор алгоритма поиска.

4. Оценки эффективности алгоритма поиска – выбор алгоритма поиска для конкретной прикладной задачи определяется по критерию минимизации затрат по времени и памяти для реализации и выполнения алгоритма поиска.

4 Критерии эффективности алгоритмов поиска

Важными оценками алгоритма поиска являются время поиска и объем дополнительной памяти, необходимой для реализации алгоритма. Анализ элементарных операций при поиске показал, что более эффективным (по времени) алгоритмом поиска является алгоритм, который использует меньшее количество операций сравнения. Операции сравнения в языках высокого уровня выполняются при проверке простых и сложных условий в операторах ветвления, а также в операторах циклов. Программный подсчет количества сравнений позволяет оценить временную сложность алгоритма поиска, а подсчет количества сравнений при разном объеме исходных данных позволяет оценить временную экспоненциальную сложность алгоритма поиска.

В языке Python для оценки времени выполнения кода может быть использован модуль timeit. Для оценки временных затрат алгоритма поиска необходимо вызвать функцию timeit.timeit:

```
timeit(stmt='pass', setup='pass',
       timer=<built-in function perf_counter>,
       number=1000000)
```

где stmt – фрагмент кода в виде строки, для которого вычисляется время выполнения,

setup – фрагмент кода в виде строки, который должен выполниться перед вычислением времени выполнения stmt,

timer – функция таймера,

number – количество повторений выполнения stmt (используется для увеличения точности оценки времени выполнения stmt); по умолчанию number = 1000000.

Функция возвращает время в секундах (тип данных float), которое требуется для выполнения фрагмента кода (stmt) number раз.

Пример использования функции timeit.timeit будет приведен при рассмотрении алгоритмов поиска.

5 Поиск данных в последовательности

В данной лекции рассматриваются алгоритмы, наиболее часто применяемые при поиске в одномерных массивах. Поиск символа в последовательности символов может быть расширен до поиска подстроки в строке.

Классификация алгоритмов поиска по объему данных для обработки позволяет разделить все алгоритмы на две группы – простые и сложные алгоритмы поиска.

Для обработки значительного количества данных привлекается дополнительная информация, как отмечалось ранее. Учет дополнительной информации обычно усложняет сам алгоритм поиска, но применение таких «усложненных» алгоритмов к большим объемам данных приводит к получению эффективного по времени алгоритма. Дополнительная память при этом используется, но объемы такой памяти незначительны по сравнению с объемами памяти для хранения данных, в которых выполняется поиск.

Рассмотрим простые алгоритмы поиска.

5.1 Линейный алгоритм поиска

Линейный алгоритм поиска – это естественный для человека алгоритм поиска, который сравнивает значение искомого элемента с каждым элементом массива. На каждом шаге такого поиска мы уменьшаем область данных, в которой выполняется поиск, только на один элемент.

```
    # Объявление и инициализация массива A.

    ...

    # Определение значения (x), для которого выполняется поиск.

    ...

    N = len(A)
    i = 0
    while i < N and A[i] != x:
        i += 1
    # Если после завершения цикла i == N, то элемента со значением x
в массиве нет.
    # Если после завершения цикла i != N, то в позиции i найдено
первое вхождение элемента со значением x в массиве.
    if i == N:
        print('Элемент не найден.')
    else:
        print('Элемент', x, 'найден в позиции', i)
```

Пример использования функции timeit.timeit для оценки временных затрат на выполнение линейного алгоритма поиска

```
    setup = '''
    import numpy as np
    import random as rnd
    N = int(input('Количество элементов в массиве = '))
    A = np.array([rnd.randint(0, 100) for i in range(N)])
```

```
x = float(input('Искомый элемент x = '))
i = 0
'''
stmt = 'while i < N and A[i] != x: i += 1'
timeit.timeit(stmt, setup)
timeit.timeit(stmt, setup, number = 1)
```

На примере данного алгоритма рассмотрим прием программирования – «использование барьерного элемента». «Барьерный элемент» – это элемент, заранее размещенный в некоторых данных в определенной позиции. Если выполняется поиск в этих данных, то совпадение искомого элемента со значением в позиции «барьерного элемента» означает отсутствие искомого элемента в данных.

В приведенном выше алгоритме последовательного поиска, упростим условие продолжения цикла (уменьшим количество сравнений), заранее выделив под массив на один элемент больше

```
# N - количество элементов в массиве.
# Объявление массива, в котором выполняется поиск.
A = np.zeros(N + 1)
# Определение значения (x), для которого выполняется поиск.
...
i = 0
# Установка барьера для поиска
A[N] = x
while A[i] != x: i += 1
if i == N:
     print('Элемент не найден.')
else:
     print('Элемент', x, 'найден в позиции', i)
```

Алгоритм линейного (последовательного) поиска эффективен при небольшом количестве данных. Требует в лучшем случае одного шага (первый элемент равен искомому). При размерности массива в N элементов алгоритм линейного поиска требует в худшем случае – N шагов, в среднем – (N+1) // 2 шагов.

5.2 Алгоритм бинарного поиска

Упорядоченность элементов массива позволяет сделать поиск более эффективным. Пусть для элементов массива выполняется условие

$$A[i - 1] \le A[i], \text{ для } 1 \le i \le (N - 1),$$

где N – количество элементов в массиве.

В таком случае можно выбрать некоторый k-й элемент массива (средний, случайный, по золотому сечению), сравнить с искомым элементом, и если выбранный элемент больше искомого, то исключить из рассмотрения ту часть массива, у элементов которой значения индексов больше, чем у выбранного элемента. Например, если k – средний (по положению в массиве) элемент массива

и A[k] > x, то все элементы с индексом, большим k или равным k, для поиска не нужны, а искать нужно среди элементов с индексами от 0 до k-1. Аналогично для случая a[k] < x.

Для построения алгоритма бинарного поиска используется еще один прием программирования – «использование флажка». «Флажок» – это переменная булевского типа, которая в случае нахождения нужного элемента меняет свое значение с False на True, что сигнализирует об окончании циклического процесса поиска.

В алгоритме бинарного поиска дополнительно вводятся две переменные L и R для описания самого младшего и самого старшего индексов элементов в выбранной зоне массива. Рассмотрим алгоритм поиска на языке Python.

```
# Объявление и инициализация массива A.

...

# Определение значения (x), для которого выполняется поиск.

...

# Флажок.
flag = False
L = 0
R = len(A) - 1
k = 0
while L <= R and not flag:
    k = (L + R) // 2
    if A[k] == x:
        flag = True
    elif A[k] < x:
        L = k + 1
    else:
        R = k - 1

if not flag:
    print('Элемент не найден.')
else:
    print('Элемент', x, 'найден в позиции', k)
```

Условие A[k] == x должно выполниться один раз, поэтому исключим использование флажка и ускорим выполнение цикла

```
# Объявление и инициализация массива A.

...

# Определение значения (x), для которого выполняется поиск.

...

i = -1
L = 0
```

```
R = len(A) - 1
while L <= R:
    mid = (L + R) // 2
    if A[mid] > x:
        R = mid - 1
    elif A[mid] < x:
        L = mid + 1
    else:
        i = mid
        break
if i == -1:
    print('Элемент не найден.')
else:
    print('Элемент', x, 'найден в позиции', i)
```

Примечание:

1. Разделение массива «по золотому сечению» (см. https://en.wikipedia.org/wiki/Golden_ratio) означает выбор в качестве элемента массива для сравнения, элемент, индекс которого делит массив в пропорции 8:5 (иррациональное число «фи» - $\phi \approx 1.61803\ldots$), то есть для целого индекса примерное значение

```
k = (L + R) * 5 // 8
```

2. Разделение массива случайным образом означает выбор в качестве элемента массива для сравнения, элемент, индекс которого вычисляется как случайное число в диапазоне изменения индексов массива

```
import random as rnd
k = rnd.randint(L, R)
```

3. Разделение массива бинарным способом (дихотомия) означает выбор в качестве элемента массива для сравнения, элемент, индекс которого делит массив в пропорции почти 1:1, то есть

```
k = (L + R) // 2
```

Так как на каждом шаге желательно исключить как можно больше элементов, то не имея никакой дополнительной информации, желательно использовать алгоритм деления пополам.

5.3 Алгоритм прямого поиска подстроки

Для поиска подстроки в исходной строке (тексте) язык Python предоставляет функцию str.find(sub[, start[, end]]), которая работает со строковым типом данных. Результатом поиска может быть индекс i, который указывает на первое с начала строки совпадение с подстрокой или значение, равное -1, которое говорит об отсутствии вхождения.

Если исходную строку и подстроку рассматривать как последовательности символов (массивы символов), то алгоритм прямого поиска подстроки на языке Python может быть представлен следующим образом

```
    # Объявление и инициализация исходной строки text и подстроки
pattern

    ...

    i = -1
    j = 0
    while (j < len(pattern)) and i < (len(text) - len(pattern)):
        j = 0
        i += 1
        while j < len(pattern) and pattern[j] == text[i + j]:
            j += 1
    if (j == len(pattern)):
        print ('Подстрока найдена в позиции', i)
    else:
        print('Элемент не найден.')
```

Этот алгоритм работает достаточно эффективно, если несовпадение пары символов происходит только после нескольких сравнений во внутреннем цикле. Однако в худшем случае, если совпадение в конце строки, требуется len(text) * len(pattern) сравнений.