Data Structures and Algorithms

Алгоритмы.
Мемоизация. Реализация на Python и Java.

Мемоизация

Мемоизация — сохранение результатов выполнения функций для предотвращения повторных вычислений. Применяется для увеличения скорости выполнения программ. Основной механизм реализации - перед вызовом функции проверяется, вызывалась ли функция ранее:

- если не вызывалась, то функция вызывается, и результат её выполнения сохраняется;
- если вызывалась, то используется сохранённый результат.

Мемоизацию можно отнести к разновидности кеширования данных. При работе с рекурсивными функциями может повышать производительность.

Общие идее при реализации мемоизации

Нужно создать элемент (в дальнейшем хранилище) который способен хранить пары значений с последующим быстрым извлечением. Для этого идеально подойдут ассоциативные массивы (словарь, карта) в качестве ключа использовать значение параметра функции, в качестве значения результат ее работы. В таком случае при вызове функции сначала проверяем нет ли таких параметров в хранилище (проверяем наличие такого ключа) и если они там есть, то возвращаем значение не выполняя при этом тело функции, если же таких параметров в хранилище нет, то выполняем тело функции после чего записываем в хранилище пару ключ которой равен параметрам, а значение вычисленному результату. После чего возвращаем вычисленное значение.

Замечания по использованию мемоизации

Общие рекомендации при применении мемоизации:

- Для того, чтобы функцию можно было подвергнуть мемоизации, она должна быть чистой:
    - детерминированной (т.е. при одном и том же наборе параметров функции должна возвращать одинаковое значение)
    - без побочных эффектов (т.е. не должна влиять на состояние системы).
- Мемоизация — это компромисс между производительностью и потреблением памяти. Мемоизация хороша для функций, имеющих сравнительно небольшой диапазон входных значений, что позволяет достаточно часто, при повторных вызовах функций, задействовать значения, найденные ранее, не тратя на хранение данных слишком много памяти.
- Функции с мемоизацией хорошо показывают себя там, где выполняются сложные, ресурсоёмкие вычисления. Здесь данная техника может значительно повысить производительность решения.

Реализация на Python

Python

Простая реализация мемоизации

```
mem = dict()

def factorial(number):
    if number <= 1:
        return 1
    elif number in mem:
        return mem[number]
    else:
        fact = number * factorial(number - 1)
        mem[number] = fact
        return fact
```

Хранилище созданное на основе словаря

Проверка есть ли такой параметр в хранилище и его возврат

Вычисление значения и занесение его в хранилище

В примере показан пример простой реализации мемоизации. В качестве хранилища используется внешний словарь. В функции выполняется проверка (есть ли такой ключ в словаре) и если да, то возвращается значение с ним связанное. В противном случае вычисляется новое значение и после вычисления заносится в хранилище.

Python

Реализация с помощью замыкания

```
def get_memoization_function(fun):
    mem = dict()
    
    def mem_fun(*arg):
        if arg in mem:
            return mem[arg]
        else:
            value = fun(*arg)
            mem[arg] = value
            return value
    return mem_fun
```

Хранилище созданное на основе словаря

Если параметры есть берем из хранилища

Вычисление значение добавление его в хранилище и возврат

В примере показан пример реализации мемоизации с помощью замыканий. Данный прием можно применить к более широкому спектру функций. Теперь если вам нужно на основе функции получить функцию с мемоизацией достаточно передать эту функцию в качестве параметра.

Пример использования

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n*factorial(n-1)


mem_factorial = get_memoization_function(factorial)
```

В этом примере продемонстрирована получение функции с мемоизацией на основе обычной функции. А так, как функция get_memoization_function принимает один параметр, то ее также можно использовать как декоратор, что делает ее применение еще более удобным.

Пример применения как декоратор

```
def get_memoization_function(fun):
    mem = dict()

    def mem_fun(*arg):
        if arg in mem:
            return mem[arg]
        else:
            value = fun(*arg)
            mem[arg] = value
            return value
    return mem_fun


@get_memoization_function
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n*factorial(n-1)
```

Применяем функцию оболочку как декоратор

Использование декоратора functools.lru_cache

Для использования этого декоратора нужно импортировать модуль funtools

У данного декоратора существует два параметра:

maxsize — сколько результатов вызова хранить. Для достижения максимальной производительности рекомендуется использовать два в целой степени.

typed — по-разному хранить параметры разных типов. Т.е. integer и float хранятся по разному

Внимание! Параметры функции должны быть хешируемого типа.

Python

Пример применения

```
import functools
import math



@functools.lru_cache(1024)
def is_prime(number):
    for n in range(2, int(math.sqrt(number)+1)):
        if number % n == 0:
            return False
    return True
```

Использование встроенного декоратора

Пример применения стандартного декоратора для реализации мемоизации. Теперь мемоизации применена к функции проверки числа на простоту.

Реализация на Java

Реализация мемоизации в Java

```
public class Sample1 {

    public static Map<Integer, BigInteger> mem = new HashMap<>();

    public static void main(String[] args) {

        BigInteger fac = factorial(5);
        System.out.println(fac);
    }

    public static BigInteger factorial(int number) {
        BigInteger fact = mem.get(number);
        if (fact != null) {
            return fact;
        } else {
            fact = BigInteger.ONE;
            for (int i = 1; i <= number; i++) {
                fact = fact.multiply(BigInteger.valueOf(i));
            }
            mem.put(number, fact);
            return fact;
        }
    }
}
```

Хранилище созданное на основе map

Проверка на наличие параметра в хранилище

Вычисление с занесением в хранилище в случае отсутствия

Описание реализации на предыдущего примера

В предыдущем примере была продемонстрирована реализация мемоизации с помощью хранилища в виде статического поля класса (Map<Integer, BigInteger> mem). Каждый статический метод этого класса имеет доступ к этому полю. В методе вычисляющем факториал сначала выполняется проверка на наличие такого ключа в карте, и если такое значение есть то возвращается значение с ним связанное. Если нет, то сначала вычисляется факториал (циклическим способом) и после вычисления в карту заносится как число, так и вычисленный факториал. Таким образом при следующем вызове, вычисления уже не будут выполнятся и результат будет взят из карты.

Java

Обобщенная реализация

```java
public class Memoization {

public static <T, R> Function<T, R> getMemoizationFunction(Function<T, R> fun) {
    class MemCache implements Function<T, R> {
        private ConcurrentMap<T, R> cache = new ConcurrentHashMap<>();
        @Override
        public R apply(T t) {
            R result = cache.get(t);
            if (result != null) {
                return result;
            }
            result = fun.apply(t);
            cache.put(t, result);
            return result;
        }
    }
    return new MemCache();
}
}
```

Пример использования

```
public class Main {

    public static void main(String[] args) {

        Function<Integer, BigInteger> fact = Main::factorial;

        fact = Memoization.getMemoizationFunction(fact);

        System.out.println(fact.apply(5));
        System.out.println(fact.apply(5));

    }

    public static BigInteger factorial(int number) {
        BigInteger fact = BigInteger.ONE;
        for (int i = 1; i <= number; i++) {
            fact = fact.multiply(BigInteger.valueOf(i));
        }
        return fact;
    }
}
```

Интересная задача где можно применить мемоизацию

Для демонстрации интересной задачи с мемоизацией исследуем гипотезу Коллатца.

Гипотеза Коллатца (3n+1 дилемма, сиракузская проблема) — одна из нерешённых проблем математики. На основании любого натурального числа генерируется следующее натуральное число по следующему правилу:

- Число четное, тогда результат получается делением на 2
- Число не четное, тогда результат 3n+1, где n — число

Гипотеза заключается в том что для любого натурального числа n эта последовательность вернет число 1. И последовательность перейдет в режим цикла.

Пример сиракузской последовательности

Например, для числа 3 получаем:

3 — нечётное, 3×3 + 1 = 10
10 — чётное, 10:2 = 5
5 — нечётное, 5×3 + 1 = 16
16 — чётное, 16:2 = 8
8 — чётное, 8:2 = 4
4 — чётное, 4:2 = 2
2 — чётное, 2:2 = 1
1 — нечётное, 1×3 + 1 = 4

Далее, начиная с 1, начинают циклически повторяться числа 1, 4, 2.

Суть исследования

Для ряда натуральных чисел вычислим длину последовательности до попадания ее в цикл. Например для числа 3 длинна последовательности равна 7 (по сути сколько членов было создано до достижения 1). Задание функции рекурсивно по своей природе, но применение мемоизации поможет ускорить процесс.

Java

Реализация метода для вычисления длинны последовательности

```
public static Map<BigInteger, BigInteger> mem = new HashMap<>();

public static BigInteger getSequinceLength(BigInteger number) {
     BigInteger result = mem.get(number);
     if (result != null) {
          return result;
     }
     if (number.equals(BigInteger.ONE)) {
          mem.put(number, BigInteger.ZERO);
     } else {
          BigInteger newNumber = null;
          if (number.remainder(BigInteger.TWO).equals(BigInteger.ZERO)) {
               newNumber = number.divide(BigInteger.TWO);
          } else {
               newNumber = number.multiply(BigInteger.valueOf(3)).add(BigInteger.ONE);
          }
     result = BigInteger.ONE.add(getSequinceLength(newNumber));
     mem.put(number, result);
     }
return mem.get(number);
}
```

Хранилище созданное на основе map

Проверка на наличие параметра в хранилище

Вычисление с занесением в хранилище в случае отсутствия

Java

Результат

Sequince length

200 180 160 140 120 100 80 60 40 20 0

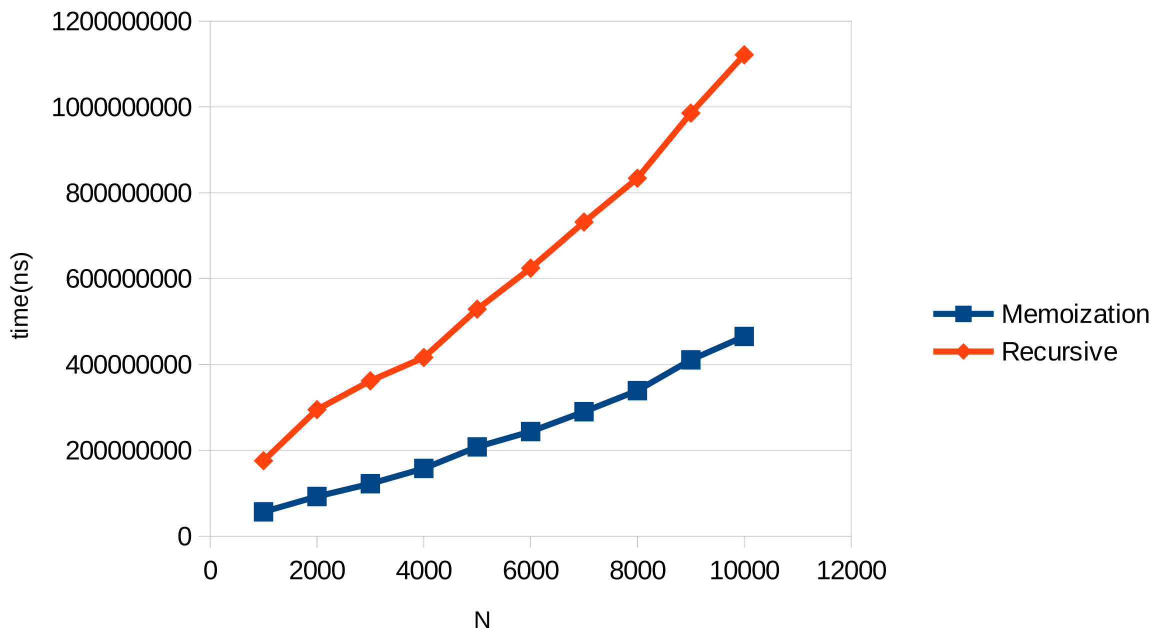
0 200 400 600 800 1000 1200 1400 1600 1800 2000

Number

Интересным результатом исследования является факт, что при увеличения значения числа максимальная длинна последовательности растет незначительно. Правда это касается только указанного диапазона.

Вычислительный эксперимент

Для проверки увеличения производительности при использовании мемоизации был проведен вычислительный эксперимент. Были вычислены длинны всех последовательностей для чисел от 1 до N и замерено время потраченное на это. Был использован просто рекурсивный метод и метод с использованием мемоизации. Результаты приведены на графике ниже.



Как можно видеть из графика применение мемоизации способно увеличить производительность решения.

Список литературы

1) Гипотеза Коллатца
2) Ананий Левитин. Алгоритмы: введение в разработку и анализ. : Пер. с англ. — М. :Издательский дом "Вильямс", 2006. — 576 с. : ил. — Парал. тит. Англ. ISBN 5-8459-0987-2.