Обзор алгоритмов

Составитель: Рощупкин Александр

Обзор алгоритмов

Цель алгоритмов

Вычислительные ресурсы современных компьютеров конечны, вычислительные потребности современных приложений всегда растут

Основные ресурсы компьютеров:

* Процессорное время
* Оперативная память

Основная цель алгоритмов: **как решить задачу быстрее и потратить на её решение меньше памяти**

Виды алгоритмов

* Комбинаторные алгоритмы
* Алгоритмы на графах
* Алгоритмы поиска
* Алгоритмы на строках
* Алгоритмы сжатия данных
* Алгоритмы сортировки
* Алгоритмы компьютерной графики
* Криптографические алгоритмы
* Алгоритмы грамматического разбора

  И другие…

Оценка алгоритма

* **Временная эффективность** – количество операций, которое выполняет алгоритм
* **Пространственная эффективность** – количество памяти, требуемое для выполнения алгоритма

В большинстве алгоритмов количество выполняемых операций напрямую зависит от размера входных данных

* Поиск максимального элемента массива зависит от длинны массива
* Чем больше входных данных, тем дольше работает алгоритм

RAM - машина

* Для универсального подсчёта числа операций, выполняемых алгоритмом, необходимо формально описать систему команд абстрактного исполнителя, измеряемого в операциях или тактах процессора
* Однопроцессорная машина с произвольным доступом к памяти
* Основные правила:
  * Арифметические и логические операции (+,-,*,/,%) – 1 такт
  * Обращение к ячейке памяти – 1 такт
  * Условный переход, условие (if) – 1 такт
  * Цикл, условие плюс количество тактов в теле – 1 + k тактов

Пример суммы элементов

* Вычислим количество операций алгоритма вычисления суммы n элементов в примере на **псевдокоде**

1. **function** sumArray($a[1..n]$)
2. $sum = 0$
3. **for** $i = 1$ **to** n **do**
4. $sum = sum + a[i]$
5. **end for**
6. **return** sum
7. **end function**

В строке 2 выполняется одна операция записи в память.

Далее, перед выполнением каждой из n итераций цикла происходят проверка условия его окончания $i = n$ и переход на строку 4 или 6.

На каждой итерации в строке 4 выполняется четыре операции: чтение из памяти значений sum и $a[i]$, их сложение и запись результата в память. В конце алгоритма выполняется возврат результирующего значения – одна операция

Количество операций **T(n) = 4n + 2**

Усреднённое значение

* Обычно точный анализ числа операций алгоритма во многих случаях не требуется.
* Достаточно ограничиться подсчетом лишь тех операций, суммарное количество которых зависит от размера входных данных.
* Так, в алгоритме SUMARRAY строки 2 и 6 не имеют значимого влияния на итоговое время выполнения, **которое фактически определяется только операциями в строке 4**.
* При анализе вычислительной сложности алгоритмов мы будем игнорировать операции, связанные с проверкой условия окончания цикла for и автоматическим увеличением его счетчика.

Фактически количество операций **T(n) = 4n**

Худший случай работы алгоритма

* Существует большое количество алгоритмов, время выполнения которых зависит не только от размера входных данных, но и от их значений.
* В качестве примера рассмотрим алгоритм LINEARSEARCH линейного поиска заданного значения x в массиве из n элементов.

```
function LINEARSEARCH(a[1..n], x)
    for i = 1 to n do
        if a[i] = x then
        return i
        end if
    end for
    return –1
end function
```

Худший случай работы алгоритма

* **Лучший случай (best case)** – экземпляр задачи (набор входных данных), на котором алгоритм выполняет наименьшее число операций. В нашем примере **a[1,2,3]** - входной массив, первый элемент которого содержит искомое значение $x = 1$. В этой ситуации требуется выполнить

$$T_{best}(n) = 3$$

операции: проверка условия окончания цикла, условие в цикле (строка 3) и возврат найденного значения (строка 4). Таким образом, время работы алгоритма в лучшем случае – **теоретическая нижняя граница времени его работы.**

* Худший случай (worst case) – экземпляр задачи, на котором алгоритм выполняет наибольшее число операций. Для рассматриваемого алгоритма – массив, в котором отсутствует искомый элемент или он расположен в последней ячейке. В этой ситуации требуется выполнить

$$T_{worst}(n) = 2n + 1$$

операции: n раз проверить условие окончания цикла и условие в нем, за- тем вернуть значение –1. **Время работы алгоритма в худшем случае – теоретическая верхняя граница времени его работы.**

Скорость роста функций

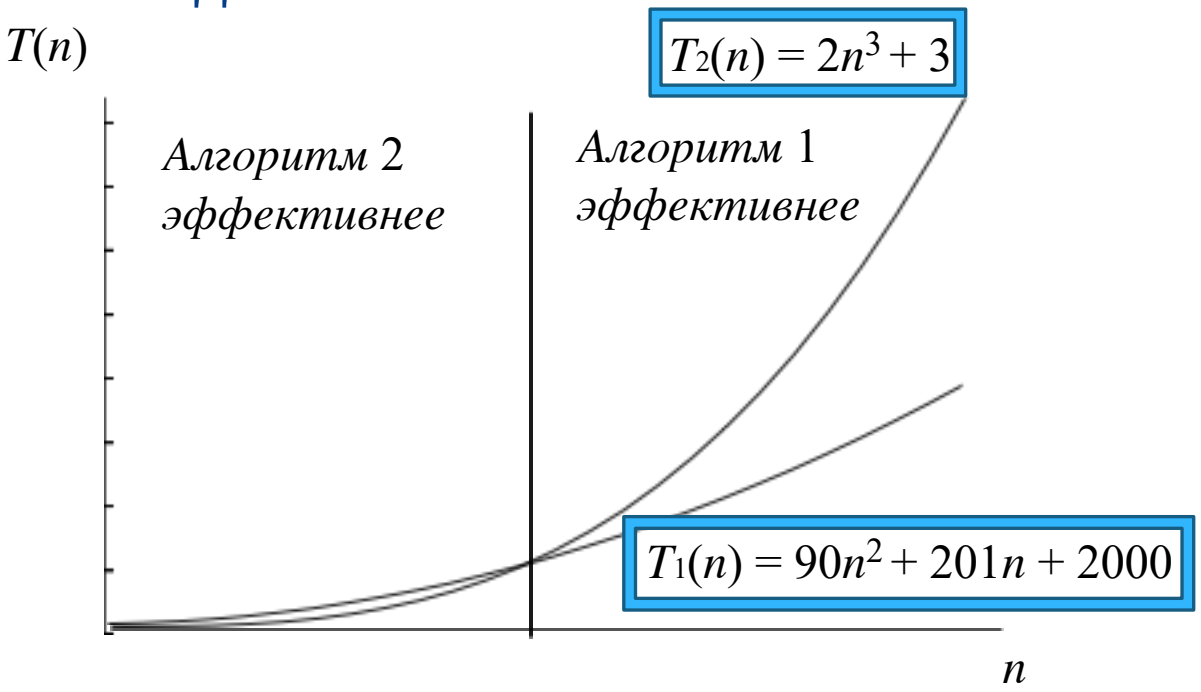
* Допустим что у нас есть два алгоритма решения одной и той же задачи. И у нас есть функции $T1(n)$ и $T2(n)$ зависимости числа операций алгоритмов от размера их входных данных для худшего случая. Определимся, что

$$T_1(n) = 90n^2 + 201n + 2000,$$
$$T_2(n) = 2n^3 + 3$$

* Возникает вопрос: какой из алгоритмов предпочтительнее использовать на практике?

Скорость роста функции

* Сравним графики функций **T1(n)** и **T2(n)**
* Обычно алгоритмы сравниваются на больших размерах данных



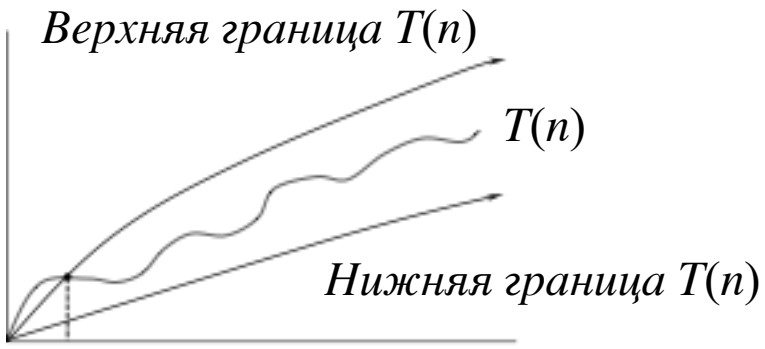
Скорость роста основных функций

* C – константная скорость (нет роста)
* log n – логарифмическая скорость
* n – линейная скорость
* n log n – линейно-логарифмическая скорость
* n^2 – квадратичная скорость
* n^3 – кубическая скорость
* 2^n – экспоненциальная скорость
* n! – факториальная скорость

Вывод: чем медленнее растёт количество операций алгоритма в зависимости от размера данных, тем **эффективнее** алгоритм

Асимптотические обозначения

* Как правило функция T(n) имеет локальные экстремумы – неровный график с выпуклостями и впадинами
* Проще работать с верхними и нижними границами



Асимптотические обозначения

Для указания границ функций $T(n)$ в теории вычислительной сложности алгоритмов (computational complexity theory) используют асимптотические обозначения:

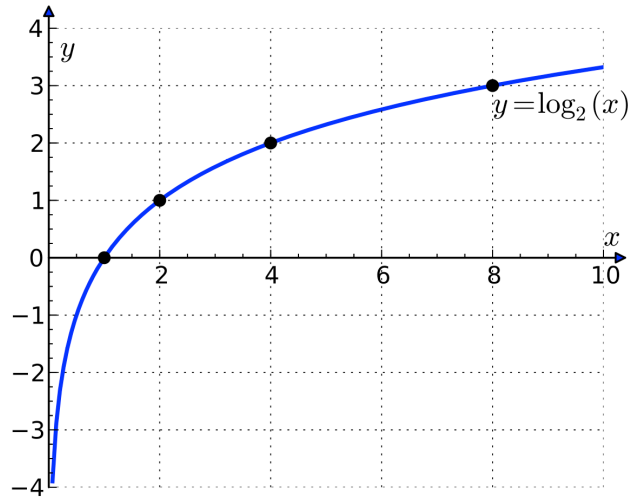
* O (о большое) – асимптотическая верхняя граница
* Ω (омега большое) – асимптотическая нижняя граница
* Θ (тета большое) – асимптотически точная граница
* o (о малое)
* ω (омега малое)

Классы сложности алгоритма в О-нотации

Обозначение класса	Название класса	Пример
$O(C)$	Константная сложность	Алгоритм определения четности целого числа. Время выполнения таких алгоритмов не зависит от размера входных данных
$O(\log n)$	Логарифмическая сложность	Алгоритм бинарного поиска в упорядоченном массиве. Такие алгоритмы на каждом шаге обрабатывают лишь часть входного набора данных
$O(n)$	Линейная сложность	Алгоритм поиска минимального элемента в неупорядоченном массиве. Просмотр всего набора входных данных
$O(n \log n)$	Линейно-логарифмическая сложность	Алгоритм сортировки слиянием. Такая сложность характерна для алгоритмов, разработанных по методике («разделяй и властвуй»)
$O(n^2)$	Квадратичная сложность	Алгоритм сортировка выбором
$O(n^3)$	Кубическая сложность	Алгоритм умножения квадратных матриц по определению
$O(2^n)$	Экспоненциальная сложность	Алгоритмы, обрабатывающие все подмножества некоторого множества из n элементов

Логарифмическая сложность

* Логарифм числа b по основанию a определяется[2] как показатель степени, в которую надо возвести основание a, чтобы получить число b
* Нахождение $x = \log_a b$ равносильно решению уравнения $a^x = b$
* Например $\log_2 8 = 3$ потому что $2^3 = 8$



Бинарный поиск

* Двоичный поиск – классический алгоритм поиска элемента в отсортированном массиве, использующий дробление массива на половины
* Метод деления на половины (метод дихотомии) позволяет найти нужный элемент не перебирая каждый элемент, а перебирая меньше чем N элементов исходной коллекции

Дополнительный материал (поиск в подстроке)

Алгоритмы поиска подстроки

* Алгоритмы представлены со среднем временем поиска и худшим

* Простой алгоритм – O(2n)(n*m)
* Алгоритм Рабина – Карпа – O(n+m)(n*m)
* Алгоритм Бойера – Мура – O(n+m)(n*m)
* Алгоритм Кнута – Морриса – Пратта – O(n+m)(n+m)

Пример простого поиска

Найти в стоке A=”*AAAAB*” подстроку X=“*AAAB*”

* Идти по проверяемой строке **A** и искать в ней вхождение первого символа искомой строки **X**. Когда находим, делаем гипотезу, что это и есть то самое искомое вхождение.
* Затем остается проверять по очереди все последующие символы шаблона на совпадение с соответствующими символами строки **A**. Если они все совпали — значит вот оно, прямо перед нами.
* Но если какой-то из символов не совпал, то ничего не остается, как признать нашу гипотезу неверной, что возвращает нас к символу, следующему за вхождением первого символа из **X**

Алгоритм КМП

Алгоритм поиска Knuth-Morris-Pratt String или алгоритм КМР ищет появление «шаблона» в основном «тексте», используя наблюдение, что, когда происходит несоответствие, само слово содержит достаточную информацию, чтобы определить, где может начаться следующее соответствие, тем самым минуя повторное рассмотрение ранее согласованных символов

Основные определения:

* Префикс — символы с начала строки
* Суффикс — символы с конца строки
* Собственные префикс и суффикс — это значит что они не совпадают со всей строкой
* Префикс функция для i-ой позиции — это длина максимального префикса строки, который короче i и который совпадает с суффиксом префикса длины i

Визуализация алгоритма КМП

* http://jovilab.sinaapp.com/visualization/algorithms/strings/kmp
* https://people.ok.ubc.ca/ylucet/DS/KnuthMorrisPratt.html

Домашнее задание

* Сравнить рост функций, используемых для сравнения сложности алгоритма. Упорядочить рост функций по возрастанию. Каждую функцию представить классом с методом вычисления функции
* Написать метод выполняющий двоичный поиск элемента в списке
* Написать метод выполняющий поиск подстроки