2. Lesson 23.02.23

Big-O обозначение

- Асимптотический анализ
- Порядок роста
- Константный - O(1)
- Логарифмический- O(log n)
- Корень - Sqrt(n)
- Линейный - O(n)
- Линейно-логарифмический - O(n log n)
- Квадратичный - O(n^2)
- Показательный - 2^n
- Факториальный - O(n!)

Асимптотический анализ

- В асимптотическом анализе мы оцениваем производительность алгоритма с точки зрения размера входных данных, другими словами мы подразумеваем анализ времени, которое потребуется для обработки очень большого набора данных.
- Имея два алгоритма для задачи, как мы узнаем, какой из них лучше?
- Мы рассчитываем, как время (time complexity) или пространство (space complexity), занимаемое алгоритмом, увеличивается с размером входных данных.

Рассмотрим задачу поиска (поиск заданного элемента) в отсортированном массиве.

- линейный поиск (порядок роста — линейный O(n))
- бинарный поиск (порядок роста — логарифмический O(log n))

Компьютер А – константное время 0.2 сек Компьютер В – константное время 1000 сек

Время выполнения линейного поиска в сек : для A = 0,2 * n для B : 1000*log(n)

n	time on A	time on B
10	2 sec	~ 1 h
100	20 sec	~ 1.8 h
10^6	~ 55.5 h	~ 5.5 h
10^9	~ 6.3 years	~ 8.3 h

Причина в том, что порядок роста бинарного поиска по размеру входных данных является логарифмическим, а порядок роста линейного поиска — линейным.

Порядок роста

Порядок роста описывает то, как сложность алгоритма растет с увеличением размера входных данных. Порядок роста представляется в виде O-нотации: O(f(x)), где f(x) — формула, выражающая сложность алгоритма.

O(1) – **Константный**

Порядок роста O(1) означает, что вычислительная сложность алгоритма не зависит от размера входных данных.

```
public int getSize(int[] arr) {
    return arr.length;
}
```

O(n) – линейный

Порядок роста O(n) означает, что сложность алгоритма линейно растет с увеличением входного массива.

Если линейный алгоритм обрабатывает один элемент 1 секунду, то сто элементов обработается за сто секунд.

```
public long getSum(int[] arr) {
     long sum = 0;
     for (int i = 0; i < arr.length; i++) {
         sum += i; }
    return sum;
 }
```

O(log n) **– логарифмический**

Порядок роста O(log n) означает, что время выполнения алгоритма растет логарифмически с увеличением размера входного массива.

Большинство алгоритмов, работающих по принципу «деления пополам», имеют логарифмическую сложность.

Пример: Алгоритм двоичного поиска

O(n log n) **– линейно-логарифмический**

Некоторые алгоритмы типа «разделяй и властвуй» попадают в эту категорию.

Пример: Сортировка слиянием и быстрая сортировка

O(n2) **– квадратичный**

Время работы алгоритма O(n^2) зависит от квадрата размера входного массива.

Квадратичная сложность — повод задуматься и переписать алгоритм.

Массив из 100 элементов потребует 1 0000 операций

Массив из миллиона элементов потребует 1 000 000 000 000 (триллион) операций.

Если 1 операция занимает миллисекунду для выполнения, квадратичный алгоритм будет обрабатывать миллион элементов 32 года. Даже если он будет в сто раз быстрее, работа займет 84 дня.

Пример: Алгоритм пузырьковая сортировка

O(n!) **– факториальный**

Очень медленный алгоритм.

Пример: Задача коммивояжёра

Найти оптимальный маршрут - для 15 городов существует 43 миллиарда маршрутов, а для 18 городов уже 177 триллионов.
Если бы существовало устройство, находящее решение для 30 городов за час, то для двух дополнительных городов требуется в тысячу раз больше времени;
то есть, более чем 40 суток!

Такая задача из класса NP - задача с ответом «да» или «нет»

Наилучший, средний и наихудший случаи

Обычно имеется в виду наихудший случай, за исключением тех случаев, когда наихудший и средний сильно отличаются. (оценка сверху)

Например: array.add()

В среднем имеет порядок роста O(1), но иногда может иметь O(n).

В этом случае мы будем указывать, что алгоритм работает в среднем за константное время, и объяснять случаи, когда сложность возрастает.

Самое важное здесь то, что O(n) означает, что алгоритм потребует не более n шагов!

Что мы в итоге измеряем и всегда ли это работает?

При измерении сложности алгоритмов и структур данных мы обычно говорим о двух вещах: кол-во операций, требуемых для завершения работы (вычислительная сложность), и **объем ресурсов**, в частности, **памяти**, который необходим алгоритму (пространственная сложность).

Алгоритм, который выполняется в 10 раз быстрее, но использует в 10 раз больше места, может вполне подходить для серверной машины с большим объемом памяти. Но на встроенных системах, где кол-во памяти ограничено, такой алгоритм использовать нельзя.

Асимптотический анализ не идеален, но это лучший доступный способ анализа алгоритмов.

Два алгоритма сортировки, которые занимают на машине 1000 nLogn и 2 nLogn.

Мы не можем судить, какой из них лучше, поскольку мы игнорируем константы.

Таким образом, вы можете в конечном итоге выбрать алгоритм, который асимптотически медленнее, но быстрее для вашего программного обеспечения.

Важно:

Скорость алгоритма измеряется не в секундах, а в приросте количества операций.

Насколько быстро возрастает время работы алгоритма в зависимости от увеличения объема входящих данных.

Время работы алгоритма выражается при помощи нотации большого «О».

Алгоритм со скоростью O(log n) быстрее, чем со скоростью O(n), но он становится намного быстрее по мере увеличения списка элементов.

Пять столпов асимптотической оценки сложности простых алгоритмов

Есть базовые правила того, как подходить к оценке сложности используя нотацию О-большое.

Существуют **пять основных правил** для расчета асимптотической сложности алгоритма:

$\boxed{1}$ Если для некоторой математической функции f алгоритму необходимо выполнить $f(N)$ действий, то это означает, что алгоритму потребуется сделать $O(f(N))$ шагов.

```
def search_max_item(li: list) -> int:
  index: int = 0
  max_item: int = li[index]
  size: int = len(li)

  while index < size:
    if li[index] > max_item:
      max_item = li[index]
    index += 1

  return max_item
```

Заметка: алгоритм проверяет каждый из N элементов набора чисел всего один раз, поэтому потребуется $O(N)$ шагов.

$\boxed{2}$ Если алгоритм выполняет одно действие, состоящее из $O(f(N))$ шагов, а затем вторую, включающую $O(g(N))$ шагов, то для функций f и g потребуется $O(f(N) + g(N))$ шагов.

```
def search_max_item(li: list) -> int:
  index: int = 0 # O(1)
  max_item: int = li[index] # O(1)
  n: int = len(li) # O(1)

  while index < n: # O(N)
    if li[index] > max_item: #
      max_item = li[index]   #
    index += 1               #

  return max_item  # O(1)
```

Заметка: алгоритм выполняет 3 шага перед циклом и ещё 1 после него. Каждый из них имеет производительность $O(1)$ (договоримся, считать это однократным действием), поэтому общее количество шагов составит $O(3 + N + 1)$, то есть $O(4 + N)$.

$\boxed{3}$ Если алгоритму необходимо сделать $O(f(N) + g(N))$ шагов, и область допустимых значений N функции $f(N)$ больше, чем у функции $g(N)$, то можно упростить выражение до $O(f(N))$.

Заметка: В ф-ции search_max_item мы выяснили, что всего будет выполнено $O(4+N)$ действий. Если параметр N начнёт возрастать, его значение превысит постоянную величину 4, а значит выражение можно упростить до $O(N)$.

$\boxed{4}$ Если алгоритму внутри каждого действия $O(f(N))$ одной операции требуется выполнять ещё $O(g(N))$ действий другой операции, то можно утверждать, что в общем алгоритм выполнит $O(f(N) \times g(N))$ действий.

```
def repetition_elements(li: list) -> bool:
  i: int = 0
  j: int = 0
  n: int = len(li)

  while i < n:
    while j < n:
      if i != j and li[i] == li[j]:
        return True
      j+=1
    i += 1
  return False
```

Заметка: в примере есть два цикла зависящих от N. Один вложен в другой. Внешний цикл перебирает все элементы массива, выполняя $O(N)$ итераций. На каждой итерации внутренний цикл повторно пересматривает все элементы, совершая $O(N)$ действий. Следовательно, общая производительность алгоритма составит $O(N \times N) = O(N^2)$.

$\boxed{5}$ Константами можно пренебречь. Если C является константой, то $O(C \times f(N))$ или $O(f(C \times N))$ можно упростить до выражения $O(f(N))$.

Заметка: В алгоритме выше блок проверка if, на самом-то деле проверяет два условия, а значит и общее количество действий внутреннего цикла получается не $O(N)$, а $O(2 \times N)$, тогда общая производительность $O(2 \times N^2)$. Предположим, что значение N увеличится в три раз, т е вместо N будет $3 \times N$, тогда $O(N)$ и $O(2 \times N)$ превратится в $O(3 \times N)$ и $O(2 \times 3 \times N)$, а общее количество действий $O(3 \times N) \times O(2 \times 3 \times N) = O((3 \times N) \times (2 \times 3 \times N)) = O(18 \times N^2)$, но $18 \times N^2 = 9 \times 2 \times N^2 = 3^2 \times 2 \times N^2 = 2 \times (3 \times N)^2$. Получается, что при увеличении объёма входных данных, количество действий увеличится в девять раз (обратное тоже верно, при уменьшении выборки алгоритм ускорится). Обратите внимание, увеличение выборки в 3 раза привело к росту общего количества действий 3^2, таким образом нас волнует не константа, а закон, по которому количество действий зависит от объёма данных — он тут N^2.