Оценка сложности алгоритмов

Для вопросов по курсу:

- Почта дисциплины - alg@moevm.info

Префикс в теме письма [ALG_53XX]

Что подразумевается под сложностью алгоритмов по времени и по памяти

- Сложность алгоритма по времени отражает зависимость количества необходимых операций для решения поставленной задачи в зависимости от объема входных данных.
- Сложность алгоритма по памяти отражает зависимость необходимого объема памяти для решения поставленной задачи в зависимости от объема входных данных.

Зачем оценивать алгоритм по времени?

- На этапе выбора алгоритма, зная приблизительные объемы данных, которые будут обработаны, можно определить, будет ли алгоритм эффективен – иначе может возникнуть ситуация “бутылочного горлышка”.
- Оценив объемы входных данных можно подобрать оптимизацию, упрощающую вычислительную, но усложняющую емкостную сложность (по памяти).

  EX: кэширование, сохранение результата предыдущих шагов

Зачем оценивать алгоритм по памяти?

- На этапе выбора алгоритма, зная приблизительные объемы данных, которые будут обработаны, можно определить, будет ли алгоритм работать – хватит ли имеющегося объема памяти.
- В случае, если объема хранилища будет недостаточно, можно применить оптимизацию, усложняющую алгоритм по времени, однако упрощающую по памяти.

  EX: отказ от сохранения предыдущих шагов работы

Методики подсчета сложности

- Асимптотическая оценка

  Грубая оценка, ограничивающая функцию, сверху или снизу или сверху и снизу

- Амортизированная оценка - более универсальная оценка

  Оценка алгоритма различными методами, которые будут рассмотрены ниже

Асимптотическая оценка

Этот метод известен вам из курса математического анализа.

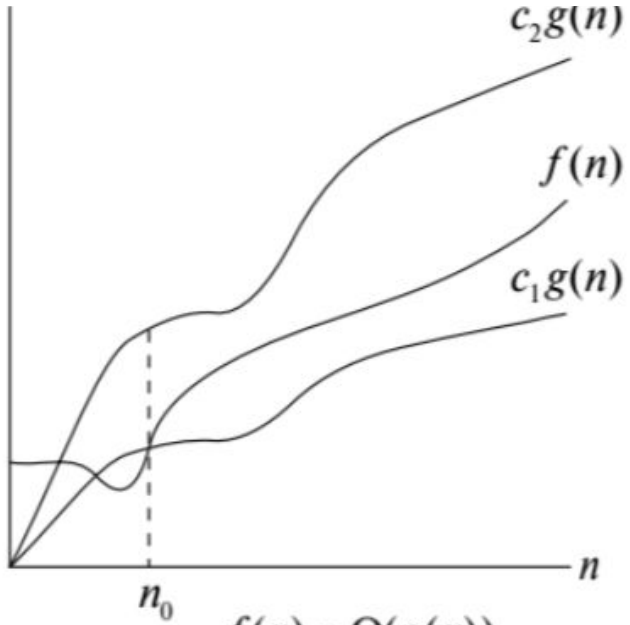
Асимптотически оценить алгоритм быстро и просто, однако это не очень точный метод.

Виды оценок:

1. **Θ**-обозначения
2. **O**-обозначения
3. **Ω**-обозначения

Θ-обозначения

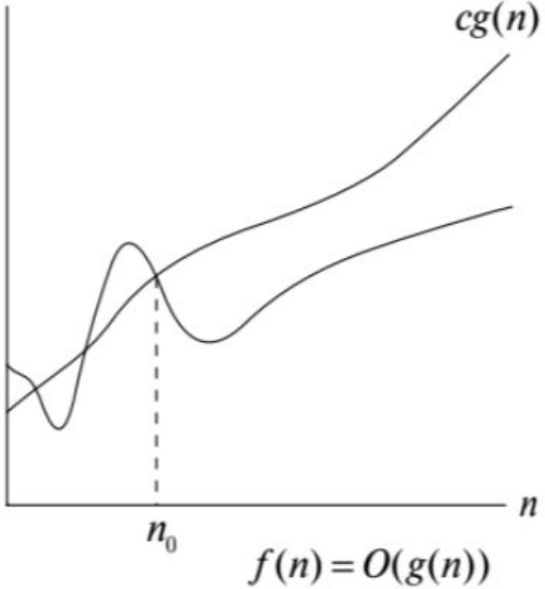
$$\Theta(g(n)) = \left\{ \begin{array}{l} f(n): \text{ существуют положительные константы } c_1, c_2 \text{ и } n_0 \\ \text{такие что } 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n) \text{ для всех } n \geqslant n_0 \end{array} \right\}$$



$f(n) = \Theta(g(n))$

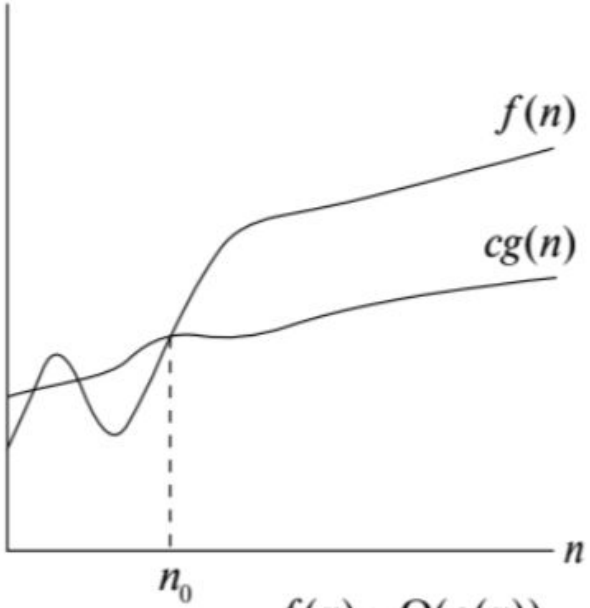
O-обозначения

$$O(g(n)) = \left\{ \begin{array}{l} f(n): \text{ существуют положительные константы } c \text{ и } n_0 \\ \text{такие что } 0 \leqslant f(n) \leqslant cg(n) \text{ для всех } n \geqslant n_0 \end{array} \right\}$$



Ω-обозначения

$$\Omega(g(n)) = \left\{ \begin{array}{l} f(n): \text{ существуют положительные константы } c \text{ и } n_0 \\ \text{такие что } 0 \leqslant cg(n) \leqslant f(n) \text{ для всех } n \geqslant n_0 \end{array} \right\}$$



$f(n) = \Omega(g(n))$

Как считать асимптотическую сложность по времени?

- Необходимо проанализировать, сколько операций будет выполнено за время работы алгоритма, оценить многочлен сверху и применить упрощение до элементарных функций.

Как считать асимптотическую сложность по памяти?

- Необходимо проанализировать, сколько ячеек памяти будет использовано во время работы алгоритма, оценить многочлен сверху и применить упрощение до элементарных функций.
- Алгоритмы, которые не используют дополнительную память (помимо той памяти, которая содержит входные данные) называются in-place алгоритмами, соответственно алгоритмы, задействующие дополнительную память называют не in-place (out-of-place) алгоритмами.

Пример вычисления

Пусть алгоритм на n (натуральное число) входных значениях производит некоторое количество операций f(n).

Пусть $f(n) = 30 + 5n + 7n^2 + 2n^3$

Тогда $f(n) <= const * n^3$, где const - некая константа (очевидно, что константа в данном случае больше, чем 2)

Из равенства $f(n) = O(g(n))$ следует, что $g(n) = n^3$

Тогда данный алгоритм оценивается как $O(n^3)$

Скорость роста сложности вычислений

	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	0,003 мкс	0,01 мкс	0,033 мкс	0,1 мкс	1 мкс	3,63 мкс
20	0,004 мкс	0,02 мкс	0,086 мкс	0,4 мкс	1 мс	77,1 лет
50	0,006 мкс	0,05 мкс	0,282 мкс	2,5 мкс	13 дней	$8,4 * 10^{15}$ лет
100	0,007 мкс	0,1 мкс	0,644 мкс	10 мкс	$4 * 10^{13}$ лет	
1000	0,010 мкс	1,0 мкс	0,966 мкс	1 мс		
10 000	0,013 мкс	10 мкс	130 мкс	100 мс		
100 000	0,017 мкс	0,1 мс	1,67 мс	10 с		
1 000 000 000	0,03 мкс	1 с	29,90 с	31,7 лет		

Steven Skiena, The Algorithm Design Manual, [page 56]

Пример in-place алгоритма

```
function reverse_in_place(a[0..n-1])
    for i from 0 to floor((n-2)/2)
        tmp := a[i]
        a[i] := a[n - 1 - i]
        a[n - 1 - i] := tmp
```

Пример не in-place алгоритма

```
function reverse(a[0..n - 1])
    allocate b[0..n - 1]
    for i from 0 to n - 1
        b[n - 1 - i] := a[i]
    return b
```

Пример сложности операций с массивом

	Начало	Середина	Конец
Вставка	O(n)	O(n)	O(1)
Удаление	O(n)	O(n)	O(1)
Доступ по индексу	O(1)		
Доступ по значению	O(n)		

Пример сложности операций с односвязным списком

	Начало	Середина	Конец
Вставка	O(1)	O(n)	O(1) или O(n)
Удаление	O(1)	O(n)	O(n)
Доступ по индексу	O(n)		
Доступ по значению	O(n)		

Пример сложности операций с двусвязным списком

	Начало	Середина	Конец
Вставка	O(1)	O(n)	O(1) или O(n)
Удаление	O(1)	O(n)	O(1) или O(n)
Доступ по индексу	O(n)		
Доступ по значению	O(n)		

Пример сложности перемножения матриц

Необходимо матрицу A (n*m) умножить на матрицу B (m*p) и получить матрицу C(n*p).

Наивная реализация этого алгоритма предполагает 3 вложенных цикла, с итераторами от 1 до n, от 1 до p, от 1 до m, потому асимптотическая оценка сложности O(n*p*m). Если входные матрицы квадратные, то сложность O(n^3).

Амортизационный анализ и зачем он нужен

- Амортизационный анализ (англ. amortized analysis) — метод подсчёта времени, требуемого для выполнения последовательности операций над структурой данных. При этом время усредняется по всем выполняемым операциям, и анализируется средняя производительность операций в худшем случае.

Нужен, чтобы избежать ситуации, когда считается общая сложность по редкой, но дорогостоящей операции (пример - расширение вектора в C++ во время вставки нового элемента), либо эта операции не учитывается вовсе.

Методы амортизационного анализа

- Агрегирующий анализ (метод усреднения)
- Метод предоплаты
- Метод потенциалов

Основной идеей амортизационного анализа является то, что любая трудоемкая операция меняет состояние программы таким образом, что до следующей трудоемкой операции обязательно пройдёт достаточно много мелких, тем самым «амортизируя» вклад трудоемкой операции.

Агрегирующий анализ

- Основная идея этого метода заключается в том, чтобы найти верхнюю оценку всех операций T(n), а затем приравнять оценку одной операции к значению T(n)/n.

В этом случае сложность выполнения m (<= n) операций будет рассчитываться как m*T(n)/n.

Метод предоплаты

- Основная идея этого метода заключается в том, чтобы каждой операции заранее присвоить амортизированную сложность так, что реальная стоимость простых операций будет ниже амортизированной, а сложных – выше.

В итоге за дорогие операции мы платим заранее частями (накапливая положительный избыток незадействованной сложности, который будет уменьшаться при выполнении дорогих операций). Такая оценка считается корректной, если избыток незадействованной сложности не становится ни в один из моментов времени отрицательным.

Метод потенциалов

- Основная идея этого метода основана на физической модели потенциальной энергии. Вводится некая функция Ф, которая сопоставляет каждому состоянию структуры данных неотрицательное число.

Потенциал Ф представляет собой ту стоимость дорогостоящих операций, которая была отложена с более простых операций, но на данных момент еще не использована. В начальный момент времени Ф = 0. На i-ом шаге Фi = сумме всех отложенных сложность - сумме всех затраченных сложностей.

Вопросы по курсу можно задавать:

Почта дисциплины - alg@moevm.info
Иванов Дмитрий Владимирович - dmitry.ivanov@moevm.info