Анализ алгоритмов

Основной задачей анализа алгоритма является получение зависимости того или иного показателя эффективности от *размера входных данных (размера входа, размера задачи)*. Сразу возникает вопрос – как измерять размер входа? Это зависит от конкретной задачи. В одних случаях размером разумно считать *число элементов на входе* (например, поиск элемента в массиве или его сортировка). Иногда размер входа измеряется не одним числом, а *несколькими* (например, число вершин и число рёбер в графе). В некоторых случаях более естественно считать размером входа *общее число бит*, необходимое для представления всех входных данных. Последний способ рассматривается как основной в теории вычислений.

Зависимость времени выполнения алгоритма от размера задачи называется *временно́й сложностью* алгоритма, а зависимость необходимого размера памяти от размера задачи — *пространственной (емкостной) сложностью* алгоритма.

Время работы алгоритма

Пусть n – размер входа (для задач, где размер входа задаётся несколькими числами, рассуждения будут аналогичны). Определим более точно, что понимать под временем работы алгоритма. Поскольку алгоритм – это ещё не программа для конкретной вычислительной машины, время его работы нельзя измерять, например, в секундах. Обычно под временем работы алгоритма понимают число элементарных операций, которые он выполняет. При этом, если алгоритм записан на каком-то псевдокоде или языке высокого уровня, предполагается, что выполнение одной строки требует не более чем фиксированного числа операций (если, конечно, это не словесное описание каких-то сложных действий).

Примечание. Действуя более формально, мы могли бы записать алгоритм с помощью инструкций предполагаемой вычислительной модели,

назначить каждой из них некую стоимость и вывести выражение для стоимости алгоритма. Однако, это достаточно трудоёмко и в большинстве случаев не требуется.

Поскольку почти любой алгоритм содержит ветвления, время его выполнения зависит не только от количества входных данных, но и от самих значений этих данных. В связи с этим записать аналитическую зависимость T(n) невозможно. Для сравнения различных алгоритмов обычно определяют время их выполнения для наихудшего или для среднего случая. Рассмотрим это подробнее.

Время выполнения в худшем и среднем случае

Существуют алгоритмы, время работы которых зависит только от размера входных данных, но не зависит от самих данных (например, поиск суммы элементов заданного массива). Однако чаще всего для разных входных данных одного и того же размера алгоритм работает разное время. Поэтому говорят о времени выполнения алгоритма в *наихудшем случае* (т.е. максимальное время выполнения по возможным входным данным) и о времени выполнения *в среднем.* Время выполнения в среднем можно определить по-разному:

- среднее время работы алгоритма по всем возможных вариантам входных данных;
- ожидаемое время его работы по всем возможным вариантам входных данных с учетом вероятности их появления

Недостатки есть и у того, и у другого способов:

- Первый способ не учитывает, что в реальных задачах данные часто распределены неравномерно.
- При втором способе получается, что мы анализируем алгоритм не в общем виде, а применительно к некой предполагаемой области.

Чаще всего при анализе времени работы алгоритма ограничиваются наихудшим случаем. Причины этого в следующем:

- Время выполнения в наихудшем случае обычно найти гораздо проще, чем в среднем.
- Зная верхнюю границу, мы можем быть уверены, что алгоритм не будет работать дольше ни на каких входных данных.
- Для многих алгоритмов плохие случаи (или близкие к ним) могут происходить очень часто
- Зачастую «средний случай» почти так же плох, как и наихудший. Например, сортировка вставками или любой другой квадратичный алгоритм сортировки.

Тем не менее, время выполнения в среднем также иногда анализируют – например, для тех алгоритмов, где оно *существенно отличается от наиухудшего*, и при этом *вероятность появления «плохих» входных данных достаточно мала* (алгоритм быстрой сортировки Хоара и др.)

Для примера найдём время выполнения алгоритма сортировки массива методом пузырька для худшего случая. Сортировка методом пузырька выполняется следующим образом. Двигаясь от конца массива к началу, мы на каждом шаге сравниваем очередные два соседних элемента. Если первый элемент больше второго, то меняем их местами. Таким образом, после первого прохода по массиву самый маленький элемент поднимется на самый верх массива и займёт нулевую позицию. Второй цикл сортировки выполняется для оставшейся части массива (без первого элемента), в результате следующий по величине элемент окажется в первой позиции массива, и т.д.

Отметим около каждой строки её стоимость (число операций) и число раз, которое эта строка выполняется.

void bubble(int *a, int n)	стоимость	Число раз
{ int i,j,temp;	c_1	N
1 for(i=0; i<n-1; i++)	c_2	$\sum_{i=0}^{n-2}(n-i)$
2 for (j=n-1; j>i;j--)	c_3	$\sum_{i=0}^{n-2}(n-i-1)$
3 if (a[j-1]>a[j])	c_4	
4 { temp:=a[j-1];	c_5	
5 a[j-1]:=a[j];	c_6	
6 a[j]:=temp;		
}		
}		

В худшем случае массив изначально упорядочен по убыванию, и условие в строке 3 всегда истинно, в результате чего строки 4-6 всегда выполняются.

Общее время работы получается следующим: $T(n)=c_1n+c_2+(c_3+c_4+c_5+c_6)=$

$=c_1n+c_2+(c_3+c_4+c_5+c_6) = (c_2+c_3+c_4+c_5+c_6)n^2/2+(c_1+(c_2-c_3-c_4-c_5-c_6)/2)n-c_2$

Асимптотические оценки сложности алгоритмов

В большинстве случаев нет необходимости находить точное число действий, выполняемых алгоритмом. Интерес представляет общий вид зависимости времени работы алгоритма от размера входных данных,

стремящегося в пределе к бесконечности – т.е. *асимптотическая временная сложность (*аналогично можно рассматривать *асимптотическую пространственную сложность)*. Так, для вышерассмотренного примера T(n) имеет вид an^2+bn+c. Однако, можно огрубить данную зависимость ещё сильней и сказать, что T(n) имеет *порядок* n^2. Слагаемые низшего порядка не учитываются, поскольку при больших входных данных они играют незначительную роль. Мультипликативную константу при старшем члене зачастую также опускают. Причины этого в следующем:

Мультипликативная константа может зависеть от разных факторов, не связанных непосредственно с алгоритмом – например, от мастерства программиста, качества компилятора и других факторов.

Для подавляющего большинства известных алгоритмов эта константа находится в разумных пределах. Это означает, что алгоритмы, которые более эффективны асимптотически, оказываются более эффективными и при тех сравнительно небольших размерах входных данных, для которых они используются на практике. Для примера рассмотрим два алгоритма сортировки. Пусть первый алгоритм выполняет сортировку массива из n чисел за $2 \cdot n^2$ операций, второй – за $100 \cdot n \cdot \log_2 n$ операций. Хотя при совсем маленьких значениях n первый алгоритм работает быстрее, с увеличением n второй алгоритм становится значительно более эффективным. Так, при n=10000 второй алгоритм работает в 15 раз быстрее первого, при n=100000 – в 120 раз, а при n=1000000 – более чем в 1000 раз.

Примечание

Асимптотические оценки всё-таки достаточно грубо характеризуют алгоритм, и в ряде случаев при анализе алгоритма (особенно при сравнении алгоритмов одного порядка сложности) стремятся получить и более точные оценки, например, сколько раз выполнится та или иная специфическая операция. Так, для алгоритмов сортировки часто рассматриваются такие характеристики, как число сравнений элементов и число замен.

Рассмотрим теперь данные понятия более строго. Запись $f(n)=\Theta(g(n))$ (читается как “тэта от g от n”), где g(n) - некоторая функция, означает следующее: найдутся такие константы $c_1,c_2>0$ и такое число n_0, что $c_1g(n) \leq f(n)$ $c_2g(n)$ для всех $n \geq n_0$. Функция g(n) в этом случае является *асимптотически точной оценкой* для f(n). На рис. 1.4а. показана иллюстрация данного определения.

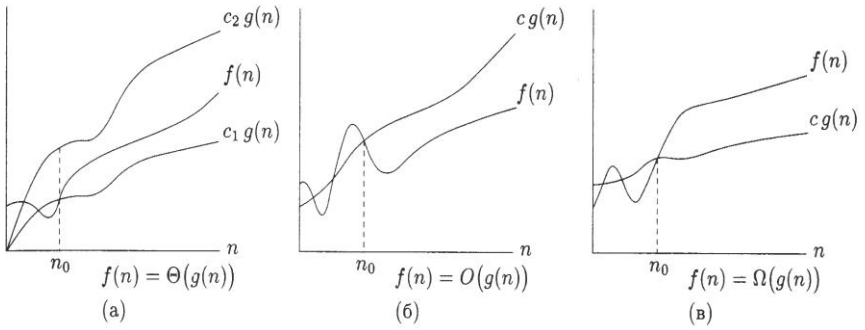

Рис .1.4. Иллюстрации к определениям $f(n)=\Theta(g(n))$, $f(n)=O(g(n)$, $f(n)=\Omega(g(n))$ (рисунок взят из [9]).

Здесь и далее предполагается, что функции f и g неотрицательны по крайней мере для достаточно больших n.

Пример 1. Покажем, что $(n+1)^2=\Theta(n^2)$. Нам нужно найти такие константы c_1 и c_2, что для всех достаточно больших n будут выполняться неравенства: c_1n^2 $(n+1)^2$, $(n+1)^2$ c_2n^2.

Возьмём $c_1=1$, тогда первое неравенство, очевидно, верно для всех n 1. Возьмём $c_2=4$. Тогда имеем:

$(n+1)^2$ $4n^2$ $\Rightarrow$ $(n+1)^2-4n^2$ 0 $(n+1-2n)(n+1+2n)$ 0 $(1-n)(3n+1)$ 0.

Полученное неравенство также выполняется для всех n 1. Таким образом, действительно $(n+1)^2=\Theta(n^2)$.

Пример 2. Покажем, что $3^n \neq \Theta(2^n)$. Для этого убедимся, что неравенство 3^n $c_2 2^n$ не может выполняться для всех достаточно больших n ни для какого фиксированного c_2. Имеем:

3^n $c_2 2^n$ $(3/2)^n$ c_2.

Какая бы большая ни была константа c_2, выражение $(3/2)^n$ при достаточно больших n всё равно превысит её. Отсюда следует, что 3^n $\Theta(2^n)$.

К сожалению, далеко не всегда для алгоритма легко получить точную асимптотическую оценку, и приходится удовлетворяться оценками менее точными. Говорят, что f(n)=O(g(n)) (читается “O большое от g от n”), если найдётся такая константа c>0 и такое число n_0, что f(n) cg(n) для всех n n_0 (рис. 1.4,б). Функция g(n) представляет собой *верхнюю асимптотическую оценку* функции f(n) (где f(n) - например, время работы или другая характеристика алгоритма).

Зная оценку времени выполнения алгоритма T(n)=O(g(n)), мы можем сказать, что число операций алгоритма не превышает g(n), умноженному на некоторую константу. Однако, мы не знаем, действительно ли алгоритм будет выполнять столько операций – возможно, на самом деле их значительно меньше. Например, для алгоритма сортировки пузырьком, рассмотренного выше, можно сказать, что $T(n)=O(n^4)$. Однако, оценка $T(n)=O(n^3)$ характеризует алгоритм более точно, а $T(n)=O(n^2)$ - ещё точней (она совпадает с точной асимптотической оценкой). При анализе алгоритма нужно стремиться получить верхнюю оценку как можно ниже.

Аналогично определяется *нижняя асимптотическая оценка*. Говорят, что f(n)=Ω(g(n)), если найдётся такая константа c>0 и такое число n_0, что f(n) cg(n)) для всех n n_0 (рис. 1.4в). Зная, что временя выполнения алгоритма T(n)=Ω(g(n)), мы можем сказать, что число операций алгоритма не меньше, чем g(n), умноженное на некоторую константу. Однако, данная характеристика не показывает, насколько действительно больше операций выполняет алгоритм в действительности. При анализе алгоритма нужно стремиться получить как можно более высокую нижнюю оценку.

Точная асимптотическая оценка времени работы алгоритма находится где-то между его нижней и верхней оценками. В частности, несложно показать, что если $T(n)=O(g(n))$ и $T(n)=\Omega(g(n))$, то $T(n)=\Theta(g(n))$.

Примечание

Иногда нижняя граница определяют несколько по-другому [3]: говорят, что $f(n)=\Omega(g(n))$, если существует такая константа c, что $f(n)$ $cg(n))$ *для бесконечно большого количества значений n*. Например, алгоритм, который выполняет n операций для чётных значений n и n^2 для нечётных, с этой точки зрения будет иметь максимальную нижнюю границу $\Omega(n^2)$, тогда как для ранее данного определения – только $\Omega(n)$. Однако, для большинства практических алгоритмов оценки совпадают.

Наиболее часто встречающиеся асимптотические оценки

В следующей таблице приведены некоторые наиболее часто встречающиеся асимптотические оценки сложности. Для каждой из них также приводится примерное время работы соответствующей программы на некоторых входных данных, а также его изменение при увеличении размера входных данных в 10 раз (будем считать, что 10 миллионов простых операций выполняются примерно за 1 секунду).

Таблица1.1

Типичные временные оценки сложности

Оценка	Пояснение и примеры типичных задач	Время работы при n=10	Время работы при n=100
$\Theta(1)$	Время выполнения алгоритма	100 нс	100 нс

	не зависит от объёма входных данных		
$\Theta(\log n)$	Логарифмическое время. Например, двоичный поиск, поиск в сбалансированном дереве и др.	332 нс	664 нс
$\Theta(n)$	Линейное время. Например, каждый элемент массива обрабатывается постоянное число раз.	1 мкс	10 мкс
$\Theta(n \cdot \log n)$	Обычно характерно для программ, использующих стратегию ”разделяй и властвуй”, например, алгоритмы быстрой сортировки или сортировка слиянием	3,32 мкс	66,44 мкс
$\Theta(n^k)$	Полиномиальная сложность. Большое число разнообразных алгоритмов.	10 мкс *(для k=2)*	1 с *(для k=2)*

$\Theta(k^n)$	Экспоненциальная сложность. Характерна для переборных алгоритмов.	102 мкс *(для k=2)*	$4{,}2 \cdot 10^{15}$ лет *(для k=2)*
$\Theta(n!)$	Факториальная сложность. Также может встретиться в переборных алгоритмах.	0,363 с	$1{,}08 \cdot 10^{146}$ лет