УВАЖАЕМЫЕ СТУДЕНТЫ!

ВАМ НЕОБХОДИМО ВЫПОЛНИТЬ СЛЕДУЮЩЕЕ:

1. Ознакомиться с лекцией и законспектируйте

2. Предоставит отчет конспекта лекции прислать в виде скриншото в течении трех дней.

3. Отправить преподавателю на почту v.vika2014@mail.ru и указать свою Ф.И.О, группу, и название дисциплины тел 072-17-44-9-22

Тема: Расчет объемной сложности алгоритмов

Оценка сложности алгоритмов

Не так давно мне предложили вести курс основ теории алгоритмов в одном московском лицее. Я, конечно, с удовольствием согласился. В понедельник была первая лекция, на которой я постарался объяснить ребятам методы оценки сложности алгоритмов. Я думаю, что некоторым читателям Хабра эта информация тоже может оказаться полезной, или по крайней мере интересной.

Существует несколько способов измерения сложности алгоритма. Программисты обычно сосредотачивают внимание на скорости алгоритма, но не менее важны и другие показатели – требования к объёму памяти, свободному месте на диске. Использование быстрого алгоритма не приведёт к ожидаемым результатам, если для его работы понадобится больше памяти, чем есть у компьютера.

Память или время

Многие алгоритмы предлагают выбор между объёмом памяти и скоростью. Задачу можно решить быстро, использую большой объём памяти, или медленнее, занимая меньший объём.

Типичным примером в данном случае служит алгоритм поиска кратчайшего пути. Представив карту города в виде сети, можно написать

алгоритм для определения кратчайшего расстояния между двумя любыми точками этой сети. Чтобы не вычислять эти расстояния всякий раз, когда они нам нужны, мы можем вывести кратчайшие расстояния между всеми точками и сохранить результаты в таблице. Когда нам понадобится узнать кратчайшее расстояние между двумя заданными точками, мы можем просто взять готовое расстояние из таблицы.

Результат будет получен мгновенно, но это потребует огромного объёма памяти. Карта большого города может содержать десятки тысяч точек. Тогда, описанная выше таблица, должна содержать более 10 млрд. ячеек. Т.е. для того, чтобы повысить быстродействие алгоритма, необходимо использовать дополнительные 10 Гб памяти.

Из этой зависимости проистекает идея объёмно-временной сложности. При таком подходе алгоритм оценивается, как с точки зрении скорости выполнения, так и с точки зрения потреблённой памяти.

Мы будем уделять основное внимание временной сложности, но, тем не менее, обязательно будем оговаривать и объём потребляемой памяти.

Оценка порядка

При сравнении различных алгоритмов важно знать, как их сложность зависит от объёма входных данных. Допустим, при сортировке одним методом обработка тысячи чисел занимает 1 с., а обработка миллиона чисел – 10 с., при использовании другого алгоритма может потребоваться 2 с. и 5 с. соответственно. В таких условиях нельзя однозначно сказать, какой алгоритм лучше.

В общем случае сложность алгоритма можно оценить по порядку величины. Алгоритм имеет сложность O(f(n)), если при увеличении размерности входных данных N, время выполнения алгоритма возрастает с той же скоростью, что и функция f(N). Рассмотрим код, который для матрицы A[NxN] находит максимальный элемент в каждой строке.

```
 for i:=1 to N do
begin
```

```
max:=A[i,1];
for j:=1 to N do
begin
if A[i,j]>max then
max:=A[i,j]
end;
writeln(max);
end;
```

В этом алгоритме переменная i меняется от 1 до N. При каждом изменении i, переменная j тоже меняется от 1 до N. Во время каждой из N итераций внешнего цикла, внутренний цикл тоже выполняется N раз. Общее количество итераций внутреннего цикла равно N*N. Это определяет сложность алгоритма O(N^2).

Оценивая порядок сложности алгоритма, необходимо использовать только ту часть, которая возрастает быстрее всего. Предположим, что рабочий цикл описывается выражением N^3+N. В таком случае его сложность будет равна O(N^3). Рассмотрение быстро растущей части функции позволяет оценить поведение алгоритма при увеличении N. Например, при N=100, то разница между N^3+N=1000100 и N=1000000 равна всего лишь 100, что составляет 0,01%.

При вычислении O можно не учитывать постоянные множители в выражениях. Алгоритм с рабочим шагом 3N^3 рассматривается, как O(N^3). Это делает зависимость отношения O(N) от изменения размера задачи более очевидной.

Определение сложности

Наиболее сложными частями программы обычно является выполнение циклов и вызов процедур. В предыдущем примере весь алгоритм выполнен с помощью двух циклов.

Если одна процедура вызывает другую, то необходимо более тщательно оценить сложность последней. Если в ней выполняется определённое число

инструкций (например, вывод на печать), то на оценку сложности это практически не влияет. Если же в вызываемой процедуре выполняется O(N) шагов, то функция может значительно усложнить алгоритм. Если же процедура вызывается внутри цикла, то влияние может быть намного больше.

В качестве примера рассмотрим две процедуры: Slow со сложностью O(N^3) и Fast со сложностью O(N^2).

```
procedure Slow;
var
i,j,k: integer;
begin
for i:=1 to N do
for j:=1 to N do
for k:=1 to N do
{какое-то действие}
end;
procedure Fast;
var
i,j: integer;
begin
for i:=1 to N do
for j:=1 to N do
Slow;
end;
procedure Both;
begin
Fast;
end;
```

Если во внутренних циклах процедуры Fast происходит вызов процедуры Slow, то сложности процедур перемножаются. В данном случае сложность алгоритма составляет O(N^2)*O(N^3)=O(N^5).

Если же основная программа вызывает процедуры по очереди, то их сложности складываются: O(N^2)+O(N^3)=O(N^3). Следующий фрагмент имеет именно такую сложность:

```
procedure Slow;
var
i,j,k: integer;
begin
for i:=1 to N do
for j:=1 to N do
for k:=1 to N do
{какое-то действие}
end;
procedure Fast;
var
i,j: integer;
begin
for i:=1 to N do
for j:=1 to N do
{какое-то действие}
end;
procedure Both;
begin
Fast;
Slow;
end;
```

Сложность рекурсивных алгоритмов

Простая рекурсия

Напомним, что рекурсивными процедурами называются процедуры, которые вызывают сами себя. Их сложность определить довольно тяжело. Сложность этих алгоритмов зависит не только от сложности внутренних циклов, но и от количества итераций рекурсии. Рекурсивная процедура может выглядеть достаточно простой, но она может серьёзно усложнить программу, многократно вызывая себя.

Рассмотрим рекурсивную реализацию вычисления факториала:

```
function Factorial(n: Word): integer;
begin
if n > 1 then
Factorial:=n*Factorial(n-1)
else
Factorial:=1;
end;
```

Эта процедура выполняется N раз, таким образом, вычислительная сложность этого алгоритма равна O(N).

Многократная рекурсия

Рекурсивный алгоритм, который вызывает себя несколько раз, называется многократной рекурсией. Такие процедуры гораздо сложнее анализировать, кроме того, они могут сделать алгоритм гораздо сложнее. Рассмотрим такую процедуру:

```
procedure DoubleRecursive(N: integer);
begin
if N>0 then
begin
DoubleRecursive(N-1);
DoubleRecursive(N-1);
end;
end;
```

Поскольку процедура вызывается дважды, можно было бы предположить, что её рабочий цикл будет равен O(2N)=O(N). Но на самом деле ситуация гораздо сложнее. Если внимательно исследовать этот алгоритм, то станет очевидно, что его сложность равна O(2^(N+1)-1)=O(2^N). Всегда надо помнить, что анализ сложности рекурсивных алгоритмов весьма нетривиальная задача.

Объёмная сложность рекурсивных алгоритмов

Для всех рекурсивных алгоритмов очень важно понятие объёмной сложности. При каждом вызове процедура запрашивает небольшой объём памяти, но этот объём может значительно увеличиваться в процессе рекурсивных вызовов. По этой причине всегда необходимо проводить хотя бы поверхностный анализ объёмной сложности рекурсивных процедур.

Средний и наихудший случай

Оценка сложности алгоритма до порядка является верхней границей сложности алгоритмов. Если программа имеет большой порядок сложности, это вовсе не означает, что алгоритм будет выполняться действительно долго. На некоторых наборах данных выполнение алгоритма занимает намного меньше времени, чем можно предположить на основе их сложности. Например, рассмотрим код, который ищет заданный элемент в векторе A.

```
function Locate(data: integer): integer;
var
i: integer;
fl: boolean;
begin
fl:=false; i:=1;
while (not fl) and (i<=N) do
begin
if A[i]=data then
```

```
fl:=true
else
i:=i+1;
end;
if not fl then
i:=0;
Locate:=I;
end;
```

Если искомый элемент находится в конце списка, то программе придётся выполнить N шагов. В таком случае сложность алгоритма составит O(N). В этом наихудшем случае время работы алгоритма будем максимальным. С другой стороны, искомый элемент может находится в списке на первой позиции. Алгоритму придётся сделать всего один шаг. Такой случай называется наилучшим и его сложность можно оценить, как O(1). Оба эти случая маловероятны. Нас больше всего интересует ожидаемый вариант. Если элемента списка изначально беспорядочно смешаны, то искомый элемент может оказаться в любом месте списка. В среднем потребуется сделать N/2 сравнений, чтобы найти требуемый элемент. Значит сложность этого алгоритма в среднем составляет O(N/2)=O(N).

В данном случае средняя и ожидаемая сложность совпадают, но для многих алгоритмов наихудший случай сильно отличается от ожидаемого. Например, алгоритм быстрой сортировки в наихудшем случае имеет сложность порядка O(N^2), в то время как ожидаемое поведение описывается оценкой O(N*log(N)), что много быстрее.

Общие функции оценки сложности

Сейчас мы перечислим некоторые функции, которые чаще всего используются для вычисления сложности. Функции перечислены в порядке возрастания сложности. Чем выше в этом списке находится функция, тем быстрее будет выполняться алгоритм с такой оценкой.

1. C – константа
2. log(log(N))
3. log(N)
4. N^C, 0<C<1
5. N
6. N*log(N)
7. N^C, C>1
8. C^N, C>1
9. N!

Если мы хотим оценить сложность алгоритма, уравнение сложности которого содержит несколько этих функций, то уравнение можно сократить до функции, расположенной ниже в таблице. Например, O(log(N)+N!)=O(N!). Если алгоритм вызывается редко и для небольших объёмов данных, то приемлемой можно считать сложность O(N^2), если же алгоритм работает в реальном времени, то не всегда достаточно производительности O(N).

Обычно алгоритмы со сложностью N*log(N) работают с хорошей скоростью. Алгоритмы со сложностью N^C можно использовать только при небольших значениях C. Вычислительная сложность алгоритмов, порядок которых определяется функциями C^N и N! очень велика, поэтому такие алгоритмы могут использоваться только для обработки небольшого объёма данных.

В заключение приведём таблицу, которая показывает, как долго компьютер, осуществляющий миллион операций в секунду, будет выполнять некоторые медленные алгоритмы.

	N=10	N=20	N=30	N=40	N=50
N^3	0.001 с	0.008 с	0.027 с	0.064 с	0.125 с
2^N	0.001 с	1.05 с	17.9 мин	1. 29 дней	35.7 лет
3^N	0.059 с	58.1 мин	6.53 лет	$3.86 * 10^5$ лет	$2.28 * 10^{10}$ лет
$N!$	3.63 с	$7.71 * 10^4$ лет	$8.41 * 10^{18}$ лет	$2.59 * 10^{34}$ лет	$9.64 * 10^{50}$ лет