Лекция 6
Сложность алгоритмов

План:

1. Временная сложность алгоритма
2. Асимптотическая сложность
3. Классы сложности алгоритмов

Центральная задача теории алгоритмов — выяснить, существует ли алгоритм решения той илиной задачи.

Если да, то ***можно ли им воспользоваться на практике, при современном уровне развития вычислительной техники***?

Т.е. способен ли компьютер за приемлемое время получить результат?

1. Временная сложность алгоритма

Исполнение любого алгоритма требует:

- определенного объема памяти компьютера для размещения данных и программы,
- времени процессора по обработке этих данных.

Эффективным называется алгоритм, обеспечивающий наиболее быстрое получение результата в приемлемое время в ограниченном объеме оперативной памяти.

Для сравнения алгоритмов по эффективности *необходимо уметь оценивать сложность алгоритмов*

Временная сложность – быстродействие

Пространственная сложность – объем требуемой памяти

Для определения сложности алгоритма оценивают временную сложность алгоритма

Время работы алгоритма (T) - количество выполненных им элементарных операций (не в секундах!)

Позволяет оценивать именно качество алгоритма, а не свойства исполнителя (быстродействие компьютера)

Размер задачи (n) - объем входных данных, необходимых для ее решения.

Временная сложность алгоритма T(n)- это функция, которая каждому размеру задачи n ставит в соответствие **максимальное** количество элементарных операций выполненных в ходе выполнения алгоритма

Пример математической оценки временной сложности алгоритма

Рассмотрим два алгоритма вычисления значения многочлена степени *n* в заданной точке *x*

$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_i x^i + ... + a_1 x^1 + a_0$

Алгоритм 1:

для каждого слагаемого, кроме a_0 возвести *x* в заданную степень последовательным умножением и затем домножить на коэффициент. Затем слагаемые сложить.

Вычисление *i*-го слагаемого(*i=1..n*) требует *i* умножений.

Значит, всего $1 + 2 + 3 + ... + n = n(n+1)/2$ умножений.

Кроме того, требуется $n+1$ сложение.

Всего $n(n+1)/2 + n + 1 =$ **$n^2/2 + 3n/2 + 1$ операций.**

Алгоритм 2:

вынесем *X*-ы за скобки и перепишем многочлен в виде

$P_n(x) = a_0 + x(a_1 + x(a_2 + ... (a_i + .. x(a_{n-1} + a_n x)))$.

Будем вычислять выражение изнутри.

Самая внутренняя скобка требует 1 умножение и 1 сложение.

Ее значение используется для следующей скобки...

И так, 1 умножение и 1 сложение на каждую скобку, которых.. *n-1* штука.

И еще после вычисления самой внешней скобки умножить на x и прибавить a_0.

Всего *n* умножений + *n* сложений = ***2n*** **операций**.

В теоретической информатике при сравнении алгоритмов используется их ***асимптотическая сложность O()*** , т. е. скорость роста количества операций при больших значениях *n*.

Функция T(n) = $n^2/2 + 3n/2 + 1$

возрастает приблизительно как n^2

Отбрасываем:

- сравнительно медленно растущие слагаемые ($3n/2+1$)
- константные множители $1/2$

Получаем асимптотическую оценку сложности для алгоритма 1: $O(n^2)$ (O – нотация)

Читается как "О большое от эн квадрат".

n	log n	n*log n	n^2
1	0	0	1
16	4	64	256
256	8	2,048	65,536
4,096	12	49,152	16,777,216
65,536	16	1,048,565	4,294,967,296
1,048,576	20	20,969,520	1,099,301,922,576
16,775,616	24	402,614,784	281,421,292,179,456

Правила формирования оценки O ()

1. ***При определении O() берется наиболее быстро растущая часть T(n).***

Например, если в программе одна функция, например, умножение, выполняется O(n) раз, а сложение - $O(n^2)$ раз, то общая сложность программы - $O(n^2)$

2. При оценке O() константы не учитываются

Следствия:

1) Алгоритмы с одинаковым O() могут иметь различную эффективность

Например, один алгоритм делает 2n+1 операций, а другой - 2500n + 1000. Оба они имеют оценку *O(n)*, так как их время выполнения растет линейно.

2) Алгоритм со временем $O(n^2)$ может работать значительно быстрее алгоритма $O(n)$ при малых n.

Например, реальное количество операций первого алгоритма может быть $n^2 + 10n + 6$,

а второго - $1000000*n + 5$. (больше умножений)

Впрочем, второй алгоритм рано или поздно обгонит первый... n^2 растет куда быстрее $1000000*n$.

3) Основание логарифма внутри символа O() не пишется

Например, пусть у нас есть $O(\log_2 n)$,

но $\log_2 n = \log_3 n / \log_3 2$, а $\log_3 2$,

как и любую константу, *O()* не учитывает.

Таким образом, $O(\log_2 n) = O(\log_3 n)$.

К любому основанию мы можем перейти аналогично, а значит и писать его не имеет смысла.

3. *Классы сложности алгоритмов*

Множество вычислительных проблем, для которых существуют алгоритмы, схожие по временной сложности, называется ***классом сложности***

Наиболее часто встречающиеся классы сложности :

Класс сложности: O(1)

Название: Алгоритмы единичной сложности

Характеристика:
алгоритмы, использующие часть входных данных и игнорирующие все остальные данные. Такие алгоритмы выполняется за один проход вне зависимости от значения *n*

Пример
Дан массив A длины n. Вычислить сумму первых трех элементов массива

Решение этой задачи содержит всего один оператор:
S := A[1] + A[2] + A[3], T(n) = 3

Класс сложности: O(log n)

Название: Алгоритмы логарифмической сложности

Пример: большинство алгоритмов поиска

Пример алгоритма сложности O(log n)

Двоичный поиск

Дан массив, в котором элементы упорядочены по возрастанию.

Найти в нем заданное значение *x* или сообщить, что его нет.

Применяется метод двоичного поиска (дихотомии): сначала ширина интервала поиска — все *n* элементовмассива. На каждом шаге этот интервал делится на 2, процесс завершается, когда левая и правая границы интервала совпадут.

```
L:=1; R:=n+1
нц пока L + 1 < R
  c:=div(L+R, 2) |или c:=L + div(R-L,2)
  если X < A[c] то
    R:=c
  иначе
    L:=c
  все
кц
если A[L] = X то
  вывод "A[", L, "]=", X
иначе
  вывод "Элемент не найден"
все
```

$$T(n) = \log_2 n + 1.$$

Класс сложности: O(n)

Название: Алгоритмы линейной сложности

Характеристика:
Для алгоритмов класса O(n) для каждого входного элемента выполняется только одно действие.

На самом деле, может быть, конечно, не одно, а два, три, и т.д., но главное не больше C*N, где C — константа.

Пример:
программы с конечным числом одномерных циклов, пробегающих весь набор входных данных, идущих друг за другом (вложенных циклов быть не может) + некоторый набор шагов до, после и между циклами. *Просмотр обложки каждой поступающей книги - для каждого входного объекта выполняется только одно действие*

Пример алгоритма сложности O(n)

Линейный поиск

Дан массив, в котором элементы расположены в произвольном порядке. Найти в нем заданное значение *x* или сообщить, что его нет.

Решение этой задачи сводится к последовательному просмотру всех элементов массива

```
nX:=0
нц для i от 1 до n
  если A[i] = X то
    nX:=i
    выход
  все
кц
если nX > 0 то
  вывод "A[", nX, "]=", X
иначе
  вывод "Элемент не найден"
все
```

В этом алгоритме число сравнений (в худшем случае) равно $T(n) = n$, поэтому он имеет линейную сложность.

Класс сложности: O(n* log n)

Название: специального названия не имеет

Пример:
алгоритмы быстрой сортировки, сортировки слиянием и "кучной" сортировки

Класс сложности: $O(n^2)$

Название: Алгоритмы квадратичной сложности

Характеристика:

Для алгоритмов класса $O(n^2)$ каждый входной элемент обрабатывается (или проходится) $C*n^2$ раз. Это означает (для наглядности, только как одну из возможностей) наличие вложенных (двойных) циклов.

Пример:

в основном, все простейшие алгоритмы сортировки

Сортировка обменом (метод пузырька)

1. При 1м проходе по массиву элементы попарно сравниваются между собой: 1й со 2м, затем 2й с 3м, следом 3й с 4м и т.д. Если предшествующий элемент оказывается больше последующего, то их меняют местами.

2. Постепенно самое большое число оказывается последним. Остальная часть массива остается не отсортированной

3. При 2м проходе незачем сравнивать последний элемент с предпоследним. Последний элемент уже стоит на своем месте. Значит, число сравнений будет на одно меньше.

4. На 3м проходе уже не надо сравнивать предпоследний и 3й элемент с конца. Поэтому число сравнений будет на два меньше, чем при первом проходе.

5. В конце концов, при проходе по массиву, когда остаются только два элемента, которые надо сравнить, выполняется только одно сравнение.

6. После этого первый элемент не с чем сравнивать, и, следовательно, последний проход по массиву не нужен.

7. Количество проходов по массиву равно m-1, где m – это количество элементов массива.

8.Количество сравнений в каждом проходе равно m-i, где i – это номер прохода по массиву (первый, второй, третий и т.д.).

9. При обмене элементов массива обычно используется "буферная" (третья) переменная, куда временно помещается значение одного из элементов.

```
for i := 1 to m-1 do
    for j := 1 to m-i do
        if arr[j] > arr[j+1] then begin
            k := arr[j];
            arr[j] := arr[j+1];
            arr[j+1] := k
        end;
```

```
нц для i от 1 до n-1
  нц для j от n-1 до i шаг -1
    если A[j] > A[j+1] то
      c:=A[j]; A[j]:=A[j+1]; A[j+1]:= c;
    все
  кц
кц
```

Для преобразования массива, состоящего из **n** элементов, необходимо просмотреть его **n-1** раз, каждый раз уменьшая диапазон просмотра на один элемент.
Количество сравнений n-1+n-2+n-3+…+1 = n(n-1)/2
Количество присваиваний 3(n-1+n-2+n-3+…+1) = 3n(n-1)/2

$$T(n) = n(n-1)/2 + 3n(n-1)/2$$

Сортировка выбором

1. Найти максимальный элемент (max) в массиве (arr).

2. Поместить его на последнее место (j).

3. Элемент, находившийся в конце массива переместить на место, где прежде находился max.

4. Уменьшить просматриваемую область массива на единицу (j – 1).

5. Снова найти максимальный элемент в оставшейся области.

6. Поместить его в конец просматриваемой области массива.

и т.д.

```
j := n;

while j > 1 do begin
    max := arr[1];
    id_max := 1;
    for i := 2 to j do
        if arr[i] > max then begin
            max := arr[i];
            id_max := i
        end;
    arr[id_max] := arr[j];
    arr[j] := max;
    j := j - 1
end;
```

```
нц для i от 1 до n-1
  nMin:=i;
  нц для j от i+1 до n
    если A[i] < A[nMin] то nMin:=i все
  кц
  если nMin <> i то
    c:=A[i]; A[i]:=A[nMin]; A[nMin]:= c
  все
кц
```

Максимальное количество перестановок на некотором массиве длины n будет равно n-1, при этом число сравнений будет одним и тем же.

В общем случае

Количество сравнений n-1+n-2+n-3+…+1 = n(n-1)/2

Количество присваиваний 3(n-1)

$$T(n) = n(n-1)/2 + 3(n-1)$$

Класс сложности: $O(n^3)$

Название: Алгоритмы кубический сложности

Характеристика:

В алгоритмах класса $O(n^3)$ каждый элемент обрабатывается $C*n^3$ раз (цикл-в-цикле-в-цикле)

Пример:

умножение матриц размера n*n

Класс сложности: $O(n^x)$

Название: Полиномиальные алгоритмы

Характеристика:

Все вышеперечисленные классы являются подклассами общего класса: $O(n^x)$, где x - любое (целое) число.

Это класс алгоритмов, работающих с полиномиальной скоростью (временная сложность которого выражается некоторой полиномиальной функцией от размера задачи n).

Также этот класс алгоритмов называют **P-класс**.

В рамках теории сложности любые алгоритмы, работающие с полиномиальной скоростью, считаются быстрыми.

Поэтому очень много исследований посвящено вопросам типа "возможен ли для данной задачи полиномиальный алгоритм"?

Класс сложности: $O(x^n)$ Класс - EXPTIME

Название: Экспоненциальные алгоритмы

Характеристика:

У *экспоненциальных* алгоритмов сложность увеличивается быстрее любого полинома.

Большинство экспоненциальных алгоритмов – это варианты полного перебора.

Класс сложности: O(n!)

Название: Факториальные алгоритмы

Примеры:

Такие алгоритмы в основном, используются в комбинаторике для определения числа сочетаний, перестановок

Наряду с алгоритмически неразрешимыми задачами, существуют ***объективно сложные*** задачи - трудоемкость которых сохраняется при любом прогрессе вычислительной техники

Пусть осуществляется 1 млн. операций в секунду

	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

Вопросы к лекции

1. Какой алгоритм называют эффективным?

2. Почему для определения сложности алгоритма оценивают не пространственную, а временную сложность алгоритма?

3. Что характеризует асимптотическая оценка сложности алгоритма?

4. В каких случаях алгоритм, имеющий асимптотическую сложность $O(n^2)$, может работать быстрее, чем алгоритм с асимптотической сложностью $O(n)$?

5. Какие классы алгоритмов считаются „медленными“?

Семинар

Оценка сложности простейших алгоритмов сортировки и поиска