Урок 2

Асимптотическая сложность алгоритма. Рекурсия

Для чего необходимо знать асимптотическую сложность алгоритма

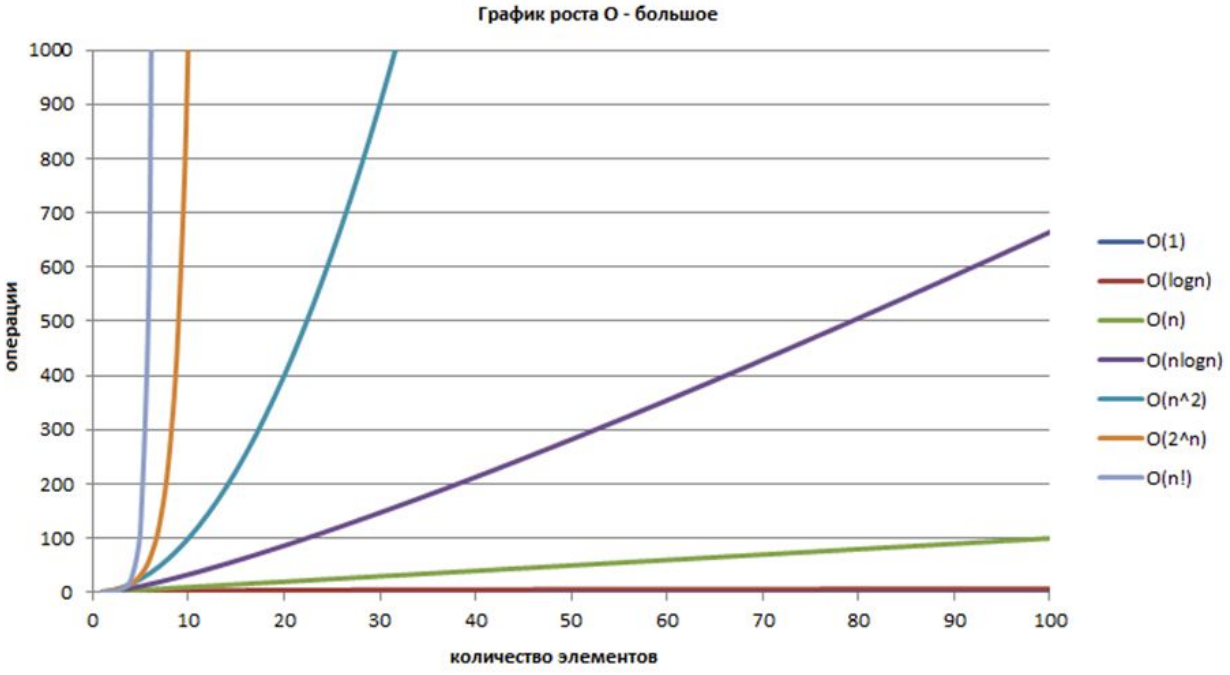
Наверное, многие, читая эти строки, думают про себя, что они всю жизнь прекрасно обходились без знаний об алгоритмах, и, конечно же, в этих словах есть доля правды. Но если встанет вопрос о доказательстве эффективности или, наоборот, неэффективности какого-либо кода, то без формального анализа уже не обойтись, а в серьезных проектах такая потребность возникает регулярно.

Асимптотическая сложность алгоритма

Асимптотическая сложность (производительность) определяется функцией, которая указывает, насколько ухудшается работа алгоритма с усложнением поставленной задачи. Такую функцию записывают в круглых скобках, предваряя прописной буквой O.

Например, доступ к ячейке массива описывается как O(1), так как для доступа к ячейке потребуется всего одна элементарная операция, и сложность здесь не возрастает независимо от размера массива.

$O(N^2)$ означает, что, по мере увеличения количества входных данных, время работы алгоритма (использование памяти либо другой измеряемый параметр) возрастает квадратично. Если данных станет вдвое больше, производительность алгоритма замедлится приблизительно в четыре раза. При увеличении количества входных данных в три раза, она станет меньше в девять раз.



Существуют пять основных правил для расчета асимптотической сложности алгоритма:

1. Если для математической функции f алгоритму необходимо выполнить определённые действия f(N) раз, то для этого ему понадобится сделать O(f(N)) шагов.

2. Если алгоритм выполняет одну операцию, состоящую из O(f(N)) шагов, а затем вторую, включающую O(g(N)) шагов, то общая производительность алгоритма для функций f и g составит O(f(N) + g(N)).
3. Если алгоритму необходимо сделать O(f(N) + g(N)) шагов, и область значений N функции f(N) больше, чем у g(N), то асимптотическую сложность можно упростить до выражения O(f(N)).
4. Если алгоритму внутри каждого шага O(f(N)) одной операции приходится выполнять ещё O(g(N)) шагов другой операции, то общая производительность алгоритма составит O(f(N) x g(N)).
5. Постоянными множителями (константами) можно пренебречь. Если C является константой, то O(C x f(N)) или O(f(C x N)) можно записать как O(f(N)).

Приведённые правила кажутся немного формальными из-за абстрактных функций f(N) и g(N), но ими очень легко пользоваться на практике. Ниже приведено несколько примеров, которые облегчат понимание.

Правило 1

Если для математической функции f алгоритму необходимо выполнить определенные действия f(N) раз, то для этого ему понадобится сделать O(f(N)) шагов.

```
int findMax(int size, int *array)
{
   int result = array[0];
   int i;
   for (i = 1; i < size; i++)
      if (array[i] > result)
         result = array[i];
   return result;
}
```

В качестве входного параметра программа использует массив целых чисел, результат возвращается в виде одного целого числа. В самом начале переменной max присваивается значение первого элемента массива. Затем алгоритм перебирает оставшиеся элементы и сравнивает значение каждого из них с max. Если он находит большую величину, то приравнивает max к ней и по окончанию цикла возвращает наибольшее найденное значение. Алгоритм проверяет каждый из N элементов массива всего один раз, поэтому его производительность составляет O(N).

Правило 2

Если алгоритм выполняет одну операцию, состоящую из O(f(N)) шагов, а затем вторую операцию, включающую O(g(N)) шагов, то общая производительность алгоритма для функций f и g составит O(f(N) + g(N)).

Вернемся к алгоритму FindMax. На этот раз обратите внимание, что несколько строк в действительности не включены в цикл. В следующей программе в комментариях справа приведён порядок времени выполнения все тех же шагов.

```
int FindMax(int size, int *array)
{
   int result = array[0];      // O(1)
   int i;
   for (i = 1; i < size; i++)  // O(N)
      if (array[i] > result)
         result = array[i];
   return result;              // O(1)
}
```

Итак, приведённый алгоритм выполняет один шаг отладки перед циклом и ещё один после него. Каждый из них имеет производительность O(1) (это однократное действие), поэтому общее время работы алгоритма составит O(1 + N + 1). Если использовать обычную алгебру и преобразовать выражение, то получится O(2 + N).

Правило 3

Если алгоритму необходимо сделать O(f(N) + g(N)) шагов, и область значений N функции f(N) больше, чем у g(N), то асимптотическую сложность можно упростить до выражения O(f(N)).

В предыдущем примере мы выяснили, что время работы алгоритма FindMax определяется выражением O(2 + N). Если параметр N начнет возрастать, его значение превысит постоянную величину 2, и предыдущее выражение можно будет упростить до O(N).

Игнорирование меньших функций позволяет пренебречь небольшими задачами отладки и очистки, чтобы сосредоточить внимание на асимптотическом поведении алгоритма, которое обнаруживается при усложнении задачи. Другими словами, время, затраченное алгоритмом на построение простых структур данных перед выполнением объемного вычисления, является несущественным по сравнению с длительностью основных расчётов.

Правило 4

Если алгоритму внутри каждого шага O(f(N)) одной операции приходится выполнять ещё O(g(N)) шагов другой операции, то общая производительность алгоритма составит O(f(N) x g(N)).

Рассмотрим алгоритм, который определяет, содержатся ли в массиве повторяющиеся элементы. (Стоит отметить, что это не самый эффективный способ обнаружения дубликатов.)

```
int FindDublicates(int N, int *array)
{
   int i, j;
   for(i = 0; i < N; i++)
      for(j = 0; j < N; j++)
         if (i != j)
            if (array[i] == array[j])
               return 1;
   // Если мы дошли до этого места, значит дубликатов нет
   return 0;
}
```

Алгоритм содержит два цикла, один из которых является вложенным. Внешний цикл перебирает все элементы массива N, выполняя O(N) шагов. Внутри каждого такого шага внутренний цикл повторно пересматривает все N элементов массива, совершая те же O(N) шагов. Следовательно, общая производительность алгоритма составит $O(N \times N) = O(N^2)$.

Правило 5

Постоянными множителями (константами) можно пренебречь. Если C является константой, то O(C x f(N)) или O(f(C x N)) можно записать как O(f(N)).

Снова посмотрите на алгоритм FindDuplicates из предыдущего примера и обратите внимание на внутренний цикл, который представлен условием if. В рамках этого условия определяется, равны ли друг другу индексы i и j. Если нет, тогда сравниваются величины array[i] и array[j], в случае их совпадения возвращается значение true.

Пренебрегая дополнительным шагом в выражении return (как правило, он выполняется один раз), предположим, что срабатывают оба оператора if (а так и происходит в большинстве случаев), тогда внутренний цикл будет пройден за O(2N) шагов. Следовательно, общая производительность алгоритма составит $O(N \times 2N) = O(2N^2)$. Последнее правило позволяет пренебречь коэффициентом 2 и записать производительность алгоритма в виде $O(N^2)$.

На самом деле, мы возвращаемся к сути асимптотической сложности: нужно выяснить, как поведёт себя алгоритм, если N начнёт возрастать. Предположим, вы увеличите N в два раза, то есть будете оперировать значением 2N. Теперь, если подставить фразу в выражение 2N^2, получится следующее: $2 \times (2N)^2 = 2 \times 4N^2 = 8N^2$. Это и есть наша величина $2N^2$, только умноженная на 4. Таким образом, время работы алгоритма увеличится в четыре раза.

Теперь давайте оценим производительность алгоритма, используя упрощённое по правилу выражение $O(N^2)$. При подстановке в него 2N получим следующее: $(2N)^2 = 4N^2$. То есть наша изначальная величина N^2 возросла в четыре раза, как и время работы алгоритма.

Из всего вышесказанного следует, что независимо от того, будете вы использовать развёрнутую формулу $2N^2$ или ограничитесь просто N^2, результат останется прежним: увеличение сложности задачи в два раза замедлит работу алгоритма в четыре раза. Таким образом, важной здесь является не константа 2, а тот факт, что время работы возрастает вместе с увеличением количества вводов N^2.

Важно понимать, что асимптотическая сложность даёт представление о теоретическом поведении алгоритма. Практические результаты могут отличаться.

Рекурсия

Рекурсией называется механизм работы программы, в котором для решения задачи из подпрограммы вызывается та же самая подпрограмма. Этот способ является альтернативой циклам и в некоторых случаях позволяет реализовать весьма интересные алгоритмы решения задачи.

Следует понимать, что любой рекурсивный метод можно преобразовать в обычный. И практически любой метод можно преобразовать в рекурсивный, если выявить рекуррентное соотношение между вычисляемыми в методе значениями.

Пример 1. Цикл с помощью рекурсии

```
// Пример вывода чисел от a до b с использованием рекурсивного алгоритма
#include <stdio.h>

void loop(int a, int b)
{
   printf("%5d", a);
   if (a < b)
      loop(a + 1, b);
}

int main(int argc, char *argv[])
{
    loop(0, 10);
    return 0;
}
```

Пример 2. Найти сумму цифр числа A

Получить последнюю цифру можно, если найти остаток от деления числа на 10. В связи с этим для разложения числа на составляющие его цифры можно использовать следующий алгоритм:

1. Находим остаток при делении числа A на 10, т.е. получаем крайнюю правую цифру числа.
2. Находим целую часть числа при делении A на 10, т.е. отбрасываем от числа A крайнюю правую цифру.
3. Если преобразованное A > 0, то переходим к пункту 1. Иначе число равно нулю, и отделять от него больше нечего.

Способ 1. Нерекурсивный

```
#include <stdio.h>
int sumDigit(long a)
{
  int s = 0;
  while (a > 0)
  {
     s = s + a % 10;
     a = a / 10;
  }
  return s;
}
int main(int argc, char *argv[])
{
   int n;
   printf("Input number:");
   scanf("%d", &n);
   printf("Summ digit: %d", sumDigit(n));
```

```
    return 0;
}
```

Способ 2. Рекурсивный

Для того, чтобы решить эту задачу рекурсивно, необходимо составить рекуррентное соотношение. Здесь оно выводится довольно просто:

S(N) = 0, при N = 0
S(N) = S(N / 10) + N % 10, при N > 0

Так как при целочисленном делении N обратится в 0, алгоритм обязательно закончит своё выполнение.

Реализация:

```
#include <stdio.h>
int sumDigit(long a)
{
    if (a == 0)
        return 0;
    else
        return sumDigit(a / 10) + a % 10;
}
int main(int argc, char *argv[])
{
    int n;
    printf("Input number:");
    scanf("%d", &n);
    printf("Summ digit:%d", sumDigit(n));
    return 0;
}
```

Как работает рекурсия. Задача ЕГЭ

Ниже на языке программирования C записаны рекурсивные функции F и G.

```
int F(int n) {
    if (n > 2)
        return F(n - 1) + G(n - 2);
    else
        return n;
}

int G(int n) {
    if (n > 2)
        return G(n - 1) + F(n - 2);
    else
        return 3 - n;
}
```

Чему будет равно значение, вычисленное при выполнении вызова G(6)?

Решение

Здесь мы просто расписываем вызовы функций, как будто их вызывает сама программа. На решении этой задачи можно легко увидеть, что мы сначала производим «спуск» по лестнице к начальным значениям, а потом обратный «подъем».

спуск | | |/	G(6) = G(5) + F(4) G(5) = G(4) + F(3) G(4) = G(3) + F(2) G(3) = G(2) + F(1) G(2) = 1 G(1) = 2 F(4) = F(3) + G(2) F(3) = F(2) + G(1) F(2) = 2 F(1) = 1	8 + 5 = 13 4 + 4 = 8 2 + 2 = 4 1 + 1 = 2 1 2 4 + 1 = 5 2 + 2 = 4 2 1	/|\ | | | подъем

Числа Фибоначчи

Напечатать первые 20 чисел Фибоначчи. Напишите рекуррентное соотношение самостоятельно.

```
#include <stdio.h>
int long fib(int n)
{
   if (n == 0) return 0;
   if (n == 1) return 1;
   return fib(n - 1) + fib(n - 2);
}
int main(int argc, char *argv[])
{
   int i;
   for (i = 0; i < 20; i++)
      printf("%d ", fib(i));
   return 0;
}
```

С рекурсией нужно быть осторожным, так как она может породить большое количество вызовов функций. Например, на рисунке ниже показано, сколько раз произойдет вызов функции для вычисления пятого числа Фибоначчи.

Все рекурсивные алгоритмы можно привести к итерационным алгоритмам. Например, решение задачи о вычислении чисел Фибоначчи может быть реализовано следующим образом.

Реализация вычисления чисел Фибоначчи с помощью массива:

```c
#include <stdio.h>
#define ARR_SIZE 20
int main(int argc, char *argv[])
{
    int f[ARR_SIZE];
    f[0] = 0;
    f[1] = 1;
    int i;
    for (i = 2; i < ARR_SIZE; i++)
        f[i] = f[i - 1] + f[i - 2];
    for (i = 0; i < ARR_SIZE; i++)
        printf("%d ", f[i]);

    return 0;
}
```

Рекурсивный перебор

В некотором несуществующем машинном алфавите четыре буквы «m», «o», «r» и «e». Нужно вывести на экран все слова, состоящие из len букв, которые можно построить из букв этого алфавита.

Это типичная задача на перебор вариантов, которую удобно свести к задаче меньшего размера. Будем определять буквы слова последовательно, одну за другой. Первая буква может быть любой из четырёх. Предположим, что сначала первой мы поставили букву «m». Тогда, чтобы получить все варианты с первой буквой «m», нужно перебрать все возможные комбинации букв, оставшихся len-1 позиций.

m	*	*	*

Далее поочередно ставим на первое место остальные буквы, повторяя процедуру.

w[0] = ’m’; //перебор последних символов без первого
w[0] = ’o’; //перебор последних символов без первого
w[0] = ’r’; //перебор последних символов без первого
w[0] = ’e’; //перебор последних символов без первого

То есть мы свели задачу для длины len к четырем задачам длины len-1.
Осталось ответить на вопрос, когда закончить рекурсию. Тогда, когда все символы будут расставлены.

Вот пример реализации алгоритма на языке C:

```
#include <stdio.h>
#include <string.h>

int count = 0;

void more()
{
   char word[] = "...";             // Длина слова. Чем больше точек, тем длиннее слово
   findWords("more", word, 0);
}

void findWords(char* A, char *word, int N)
{
    if (N == strlen(word))        // Слово построено
    {
        printf("%d %s\n", ++count, word);
        return;
    }
    int i;
    char *w;
    w = word;
    for(I = 0; I < strlen(A); i++)
    {
        w[N] = A[i];
        findWords(A, w, N + 1);// Рекурсия
    }
}

int main(int argc, char *argv[])
{
    more();
    return 0;
}
```

Ханойская башня

Ханойская башня является одной из популярных головоломок XIX века. Даны три стержня, на один из которых нанизаны восемь колец, причём кольца отличаются размером и лежат меньшее на большем. Задача состоит в том, чтобы перенести пирамиду из восьми колец за наименьшее число ходов на другой стержень. За один раз разрешается переносить только одно кольцо, причём нельзя класть больший элемент на меньший. Здесь можно поиграть в игру, чтобы лучше понять ее смысл.

Рекурсивное решение

Пусть на стержне 1 имеется n дисков. Как решить задачу при n = 1, совершенно ясно: переложить этот диск на третий стержень, и всё. И при n = 2 решение понятно: верхний диск переносим на стержень 2, второй диск переносим на стержень 3 и, наконец, меньший диск переносим на стержень 3.

Тогда пусть мы умеем переносить башню из n−1 дисков со стержня 1 на любой другой стержень в полном соответствии с предписанными правилами. Берем теперь башню из n дисков. Верхние n−1

дисков переносим на другой стержень. Оставшийся диск перекладываем на свободный стержень и теперь на него, пользуясь тем же алгоритмом, переносим n−1 дисков. Задача решена.

Решение, как вы видите, построено на том, что при исполнении алгоритма для n дисков мы обращаемся к тому же алгоритму для n−1 дисков. Ситуация, в которой какой-то алгоритм, сам или через другие алгоритмы, вызывает себя в качестве вспомогательного, называется рекурсией. Сам алгоритм при этом называется рекурсивным.

Рекурсивно решаем задачу «перенести башню из n−1 диска на второй штырь». Затем переносим самый большой диск на третий штырь и рекурсивно решаем задачу «перенести башню из n−1 диска на третий штырь».

```
#include <stdio.h>

void TowerOfHanoi(int from, int to, int other, int n)
{
   if (n > 1) TowerOfHanoi(from, other, to, n - 1);
   printf("%d %d\n", from, to);
   if (n > 1) TowerOfHanoi(other, to, from, n - 1);
}

void main()
{
   TowerOfHanoi(1, 2, 3, 3);
}
```

Приложение

Задача. Закраска замкнутой области

Определите, как заполнится массив Map после выполнения данной программы:

```
#include <stdio.h>
#include "mylib.h"

int map[10][10] = {
    {0, 0, 0, 0, 1, 1, 0, 0, 0, 0},
    {0, 0, 0, 1, 0, 0, 1, 0, 0, 0},
    {0, 0, 0, 1, 0, 0, 1, 0, 0, 0},
    {0, 1, 1, 0, 0, 0, 0, 1, 1, 0},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {0, 1, 1, 0, 0, 0, 0, 1, 1, 0},
    {0, 0, 0, 1, 0, 0, 1, 0, 0, 0},
    {0, 0, 0, 1, 0, 0, 1, 0, 0, 0},
    {0, 0, 0, 0, 1, 1, 0, 0, 0, 0}
};

int k = 0;
void paint(int x, int y)
{
   if (map[x][y] == 0)
   {
      k++;
      map[x][y] = k;
      Paint(x, y - 1);
      Paint(x - 1, y);
      Paint(x, y + 1);
      Paint(x + 1, y);
   }
}

int main(int argc, char *argv[])
{

   paint(4, 4);
   return 0;
}
```

Оказывается, заливка в различных редакторах тоже использует рекурсию.

Ханойская башня без использования рекурсии

Решение задачи о Ханойской башне без рекурсии на C с использованием стека, реализованного через массив.

```
#include <stdio.h>
struct State
{
	int n;
	int src;
	int dest;
	int tmp;
	int step;
};
#define T State
#define MaxN 1000
T stack[MaxN];
int N = -1;
void push(T i)
{
	if (N<MaxN)
	{
		N++;
		stack[N] = i;
	}
	else printf("Stack overflow");
}
T pop()
{
	if (N != -1) return stack[N--];
	else printf("Stack is empty");
}
T* back()
{
	if (N != -1) return &stack[N];
	else printf("Stack is empty");
}
void tower(int n, int src, int dest, int tmp)
{
	{
		State state;
		state.n = n;
		state.src = src;
		state.dest = dest;
		state.tmp = tmp;
		state.step = 0;
		push(state);
	}
	while (N != -1)
	{
		State* state = back();
		switch (state->step)
		{
		case 0:
			if (state->n == 0)
				pop();
			else
			{
				++state->step;
				State newState;
				newState.n = state->n - 1;
				newState.src = state->src;
				newState.dest = state->tmp;
```

```
                                newState.tmp = state->dest;
                                newState.step = 0;
                                push(newState);
                        }
                        break;
                case 1:
                        printf("%d->%d\n",state->src,state->dest);
                        ++state->step;
                        State newState;
                        newState.n = state->n - 1;
                        newState.src = state->tmp;
                        newState.dest = state->dest;
                        newState.tmp = state->src;
                        newState.step = 0;
                        push(newState);
                        break;
                case 2:
                        pop();
                        break;
                }
        }
}

int main()
{
        // Перекладываем с первого на второй
        // 1 параметр - количество колец
        tower(3, 1, 2, 3);
        getchar();
}
```

Домашнее задание

1. Реализовать функцию перевода чисел из десятичной системы в двоичную, используя рекурсию.
2. Реализовать функцию возведения числа a в степень b:
    a. Без рекурсии.
    b. Рекурсивно.
    c. *Рекурсивно, используя свойство чётности степени.
3. **Исполнитель «Калькулятор» преобразует целое число, записанное на экране. У исполнителя две команды, каждой присвоен номер:
    1. Прибавь 1.
    2. Умножь на 2.

    Первая команда увеличивает число на экране на 1, вторая увеличивает его в 2 раза. Сколько существует программ, которые число 3 преобразуют в число 20:
    a. С использованием массива.
    b. *С использованием рекурсии.

Записывайте в начало программы условие и свою фамилию. Все решения создавайте в одной программе. Задания со звёздочками выполняйте в том случае, если решили задачи без звёздочек.

Дополнительные материалы

1. Вычислительная сложность

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Род Стивенс, Алгоритмы. Теория и практическое применение. Издательство «Э», 2016.
2. Никлаус Вирт, Алгоритмы и структуры данных. Новая версия для Оберона, ДМК, Москва, 2010.
3. Пол Дейтел, Харви Дейтел. C для программистов. C введением в C11, ДМК, Москва, 2014.