ТЕХНОСФЕРА

Алгоритмы и структуры данных

Лекция 1

Сергей Леонидович Бабичев

Содержание лекции

- Свойства алгоритма.
- Сложность алгоритма. O- и Θ- нотации.
- Исполнитель алгоритма.
- Автоматы.
- Инварианты. Индуктивные функции.
- Абстракции. Интерфейс абстракции.
- Рекурсия. Принцип *разделяй и властвуй*.
- Числа и их представление.
- Основная теорема о рекурсии.
- Быстрое вычисление степеней.

Свойства алгоритма.

Исполнитель

Алгоритм — это последовательность команд для исполнителя, обладающая рядом свойств:

- **полезность**, то есть умение решать поставленную задачу.
- **детерминированность**, то есть каждый шаг алгоритма должен быть строго определён во всех возможных ситуациях.
- **конечность**, то есть способность алгоритма завершиться для любого множества входных данных
- **массовость**, то есть применимость алгоритма к разнообразным входным данным.

Алгоритм для своего *исполнения* требует от исполнителя некоторых *ресурсов*.
Программа есть запись алгоритма на формальном языке.

Исполнители

Одна задача — несколько алгоритмов — разные используемые ресурсы.
Разные исполнители — разные *элементарные действия* и *элементарные объекты*.
Исполнитель «компьютер»:

- устройство *центральный процессор*
- элементарные действия — сложение, умножение, сравнение, переход ...
- устройство *память* как хранителя элементарных объектов
- элементарные объекты —- целые, вещественные числа

Эффективность — способность алгоритма использовать ограниченное количество ресурсов.

Сложность алгоритма. O- и Θ- нотации.

Сложность алгоритма

Что есть сложность алгоритма?

- комбинационная сложность — минимальное число элементов для реализации алгоритма в виде вычислительного устройства.
- описательная сложность — длина описания алгоритма на формальном языке
- вычислительная сложность — количество элементарных операций, исполняемых алгоритмом для неких входных данных.

Нет циклов — описательная сложность примерно коррелирует с вычислительной.
Есть циклы — интересна асимптотика зависимости времени вычисления от входных данных.

Главный параметр сложности алгоритма

Введём понятие *главный параметр* N, наиболее сильно влияющий на скорость исполнения алгоритма. Это может быть:

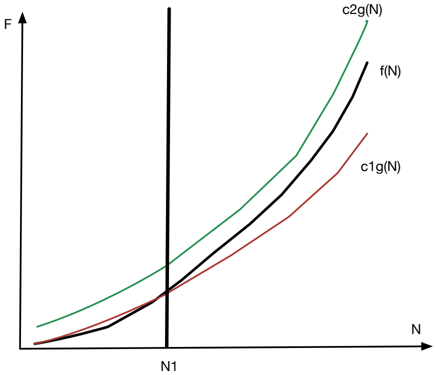
- размер массива
- количество символов в строке
- количеством битов в записи числа
- если таких параметров несколько — обобщённый параметр, функция от нескольких параметров

Нотация сложности. Символ Θ

Далее: сложность $\equiv$ вычислительная сложность.
Функция $f(N)$ имеет порядок $\Theta(g(N))$, если существуют постоянные c_1, c_2 и N_1 такие, что для всех $N > N_1$

$$0 \leqslant c_1 g(N) \leqslant f(N) \leqslant c_2 g(N).$$

$\Theta(f(n))$ — класс функций, примерно пропорциональных $f(n)$

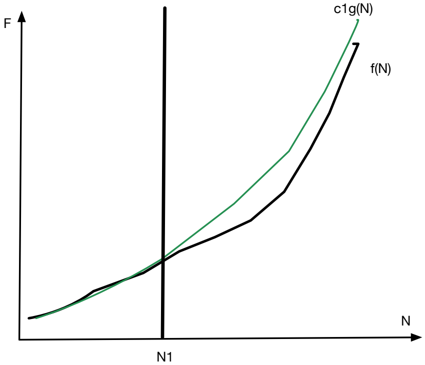


Нотация сложности. Символ O

Функция $f(N)$ имеет порядок $O(g(N))$, если существуют постоянные c_1 и N_1 такие, что для всех $N > N_1$

$$f(N) \leqslant c_1 g(N)$$

$O(f(n))$ — класс функций, ограниченных сверху $cf(n)$.



Приближённое вычисление сложности

- Пусть $F(N)$ — функция сложности алгоритма в зависимости от N
- Тогда если существует такая функция $G(N)$ (асимптотическая функция) и константа C, что

$$\lim_{N \to \infty} \frac{F(N)}{G(N)} = C,$$

то сложность алгоритма $F(N)$ определяется функцией $G(N)$ с коэффициентом амортизации C.

Асимптотика основных зависимостей

Класс сложности определяется по асимптотической зависимости $F(N)$.

- Экспонента с любым коэффициентом превосходит любую степень
- Степень с любым коэффициентом, большим единицы, превосходит логарифм по любому основанию, большему единицы
- Логарифм по любому основанию, большему единицы превосходит 1

- $F(N) = N^3 + 7N^2 - 14N = \Theta(N^3)$
- $F(N) = 1.01^N + N^{10} = \Theta(1.01^N)$
- $F(N) = N^{1.3} + 10\log_2 N = \Theta(N^{1.3})$

На практике чаще используют O-нотацию, так как Θ-нотацию обычно сложнее доказывать.

Зависимость времени исполнения от исходных данных

Пусть имеется массив A длиной N элементов.
Сколько операций потребуется, чтобы обнаружить номер первого вхождения элемента со значением P алгоритмом, заключающимся в последовательном просмотре элементов массива?

- $K_{min} = 1$
- $K_{max} = N$
- $K_{avg} = \frac{\sum_{i=1}^{N} i}{N} = \frac{N \times (N-1)}{2N} = \frac{N-1}{2}$

Подходит ли символ Θ?

Зависимость времени исполнения от исходных данных

Пусть имеется массив A длиной N элементов.
Сколько операций потребуется, чтобы обнаружить номер первого вхождения элемента со значением P алгоритмом, заключающимся в последовательном просмотре элементов массива?

- $K_{min} = 1$
- $K_{max} = N$
- $K_{avg} = \frac{\sum_{i=1}^{N} i}{N} = \frac{N \times (N-1)}{2N} = \frac{N-1}{2}$

Подходит ли символ Θ?
Нет. Для данного алгоритма подходит O-символ:
$f(N) = O(N)$.

Зависимость времени исполнения от исходных данных

Пусть имеется массив A длиной N элементов.
Сколько операций потребуется, чтобы сложить все элементы массива?

- $K = N$

  Подходит ли символ Θ?

Зависимость времени исполнения от исходных данных

Пусть имеется массив A длиной N элементов.
Сколько операций потребуется, чтобы сложить все элементы массива?

- $K = N$

  Подходит ли символ Θ?
  Подходит. $f(N) = \Theta(N)$.

Неполиномиальные задачи. Задача о рюкзаке.

- Имеется:
  - N предметов, каждый из которых имеет объём V_i и стоимость C_i, предметы неделимы;
  - рюкзак вместимостью V.
- Требуется:
  - поместить в рюкзак набор предметов максимальной стоимости;
  - суммарный объём выбранных предметов не превышает объёма рюкзака.

Задача о рюкзаке.

- Обратим внимание на то, что предметы разрезать на куски нельзя. Если это разрешить, то задача будет иметь простое решение.
- Для решения задачи достаточно перебрать все возможные комбинации из N предметов. Это гарантирует то, что мы не пропустим нужной комбинации.
- Для определения количества комбинаций можно рассуждать так, что K предметов можно выбрать из N предметов C_N^K и так для всех K от 0 до N.

$$F(N) = \sum_{K=0}^{N} C_N^K$$

Задача о рюкзаке.

- Обратим внимание на то, что предметы разрезать на куски нельзя. Если это разрешить, то задача будет иметь простое решение.
- Для решения задачи достаточно перебрать все возможные комбинации из N предметов. Это гарантирует то, что мы не пропустим нужной комбинации.
- Для определения количества комбинаций можно рассуждать так, что K предметов можно выбрать из N предметов C_N^K и так для всех K от 0 до N.

$$F(N) = \sum_{K=0}^{N} C_N^K$$

$$F(N) = (1+1)^N = 2^N$$

Одно из решений задачи о рюкзаке

1. Перенумеруем все предметы.
2. Установим максимум стоимости в 0.
3. Составим двоичное число с N разрядами, в котором единица в разряде будет означать, что предмет выбран для укладки в рюкзак (расстановку).
4. Рассмотрим все расстановки, начиная от $000\ldots000$ до $111\ldots111$, для каждой из них подсчитаем значение суммарного объёма.
   4.1 Если суммарный объём расстановки не превосходит объёма рюкзака, то подсчитывается суммарная стоимость и сравнивается с достигнутым ранее максимумом стоимости.
   4.2 Если вычисленная суммарная стоимость превосходит максимум, то максимум устанавливается в вычисленную стоимость и запоминается текущая конфигурация.

Свойства алгоритма

Предложенный алгоритм:

1. Детерминированный
2. Конечный
3. Массовый
4. Полезный

Его сложность пропорциональна 2^N, так как требуется перебрать все возможные комбинации предметов.

Сложность решения задачи о рюкзаке

- Много ли времени потребуется на решение задачи для $N = 128$?

Сложность решения задачи о рюкзаке

- Много ли времени потребуется на решение задачи для $N = 128$?
- Предположим, на подсчёт одного решения потребуется 10^{-9} секунд, то есть, одна наносекунда.

Сложность решения задачи о рюкзаке

- Много ли времени потребуется на решение задачи для $N = 128$?
- Предположим, на подсчёт одного решения потребуется 10^{-9} секунд, то есть, одна наносекунда.
- Предположим, задачу будет решать триллион компьютеров (10^{12})

Сложность решения задачи о рюкзаке

- Много ли времени потребуется на решение задачи для $N = 128$?
- Предположим, на подсчёт одного решения потребуется 10^{-9} секунд, то есть, одна наносекунда.
- Предположим, задачу будет решать триллион компьютеров (10^{12})
- Тогда общее время решения задачи будет составлять

$$\frac{2^{128} \times 10^{-9}}{10^{12}} \text{секунд} \approx 10.8 \times 10^{9} \text{лет}$$

NP-задачи

Задача о рюкзаке относится к классу *NP-полных*.
Быстрое (полиномиальное) точное решение таких задач (пока) не найдено.
Если будет найдено решение одной из *NP-полных* задач, то будут решены все задачи из этого класса.
Сейчас их решают приближённо.

Исполнитель алгоритма

Исполнители

Нашим исполнителем будет язык `C++`.

- Элементарные типы отображаются на вычислительную систему `char`, `int`, `double`.
- Элементарные операциями аппаратного исполнителя: операции над элементарными типами и операции передачи управления.
- Типы данных языка есть комбинация элементарных типов данных.
- Операции языка есть комбинация элементарных операций.

Элементарные и неэлементарные операции

Пример: цикл `for` как неэлементарная операции языка.

```
int a[10];
int s = 0;
for (int i = 0; i < 10 && a[i] % 10 != 5; i++) {
    s += a[i];
 }
```

Неэлементарный тип *массив*, его представитель `a`
Элементарный тип `int`, его представитель `s`
Элементарная операция присваивания (инициализации) `s=0`
Неэлементарная операция `for`, состоящая из операций присваивания `i = 0`, двух операций сравнения, и т. д.

Представление типов

Целые числа — двоичное представление.
Простые элементарные операции: сложение, вычитание, присваивание,...
Сложные элементарные операции: целочисленное умножение, деление (не на степень двойки), переход.

```
if (x > 0) t = 1;
else t = 0;
```

выполняется много медленнее, чем

```
t = x > 0;
```

так как для второго есть машинная команда.

Аппаратные исполнители

Имеется много архитектур аппаратных средств.

- `X86` — изобретена `Intel`, лицензирована и производится `AMD`. `int`, адреса — 32 бита. 32 бита и максимально обрабатываемый аппаратно и быстро целочисленный формат.
- `X64` — изобретена `AMD`, лицензирована и производится `Intel`. `int` — 32 бита, но максимально обрабатываемый аппаратно и быстро целочисленный формат — 64 бита.
- `ARM` схожа с `X86`, `ARM64` — с `X64`. Телефоны. Планшеты. Иногда серверы. **Пока** не используется для ноутбуков и настольных компьютеров.

Модулярная арифметика

Задача: найти последнюю цифру значения 3^{7^8}.
Решение: Заметим, что последние цифры степени тройки образуют период.

$$
\begin{aligned}
3^0 &\pmod{10} = 1 \\
3^1 &\pmod{10} = 3 \\
3^2 &\pmod{10} = 9 \\
3^3 &\pmod{10} = 7 \\
3^4 &\pmod{10} = 1 \\
3^5 &\pmod{10} = 3 \\
\ldots
\end{aligned}
$$

Последняя цифра определяется остатком от деления 7^8 на 4.

Модулярная арифметика

Аналогично:

$$
\begin{aligned}
&7^0 \pmod 4 = 1\\
&7^1 \pmod 4 = 3\\
&7^2 \pmod 4 = 1\\
&7^3 \pmod 4 = 3\\
&\dots
\end{aligned}
$$

$7^8 \pmod 4 = 1 \to 3^{7^8} \pmod{10} = 3$

Модулярная арифметика

Вся компьютерная арифметика основана на тождествах:

$$
\begin{aligned}
(a+b) \pmod{m} &= (a \pmod{m} + b \pmod{m}) \pmod{m} \\
(a-b) \pmod{m} &= (a \pmod{m} - b \pmod{m}) \pmod{m} \\
(a\times b) \pmod{m} &= (a \pmod{m} \times b \pmod{m}) \pmod{m} \\
\ldots
\end{aligned}
$$

В качестве m при двоичном представлении выступают числа 2^8, 2^{16}, 2^{32}, 2^{64}

Инварианты.
Индуктивные функции.

Индуктивное программирование. Индуктивные функции.

Пусть имеется множество M. Пусть аргументами функции f будут последовательности элементов множества M а значениями — элементы множества N.

Тогда, если её значение на последовательности

$$x_1, x_2, \ldots x_n$$

можно восстановить по её значению на последовательности

$$x_1, x_2, \ldots, x_{n-1} \text{ и } x_n,$$

то такая функция называется *индуктивной*.
Пример: Если мы хотим найти наибольшее значение всех элементов последовательности, то функция *maximum* — индуктивна, так как

$$maximum(x_1, x_2, \ldots, x_n) = max(maximum(x_1, x_2, \ldots, x_{n-1}), x_n)$$

Индуктивные функции и инварианты

Простейший алгоритм: сумма элементов массива.

```
// Вход: массив a[n]
// Выход: сумма его элементов
int sum = 0;
for (int i = 0; i < n; i++) {
   sum += a[i];
}
```

Применяемая нами операция — использование индуктивной функции.

Значение функции для подмножества является *инвариантом*, то есть, предикатом, значение которого всегда истинно.

Инвариант: значение переменной `sum` в момент времени `i` есть сумма частичного массива от 0 до `i` включительно.

Доказательство корректности алгоритмов

Инвариант — важнейшее понятие при доказательстве корректности алгоритмов.
Путь доказательства корректности фрагмента алгоритма:

1. выбираем предикат, значение которого истинно до начала исполнения фрагмента.
2. исполняем фрагмент, наблюдая за поведением предиката;
3. если после исполнения предикат остался истинным при любых путях прохождения фрагмента, алгоритм корректен относительно значения этого предиката.

Инварианты и предикаты алгоритма

Ещё один инвариант:

```
int m = a[0];
for (int i = 1; i < N; i++) {
   if (a[i] > m) {
      m = a[i];
   }
}
```

Инвариант: $\forall i < N : m_i$ есть наибольшее значение из элементов $a[0] ... a[i]$, то есть, $m_i = maximum$.

Автоматы.

Понятие автомата

- *Автоматы* — произведение множеств *состояний* P и *переходов* T.
- Имеются *начальное состояние автомата* и *заключительное состояние*.
- *Конечный автомат* — автомат с ограниченными множествами состояний и переходов.
- *Вход автомата* — события, вызывающие переходы.
- *Детерминированный конечный автомат* — конечный автомат, в котором одна и та же последовательность входных данных приводит при одном и том же начальном состоянии к одному и тому же заключительному.

Применение автоматов

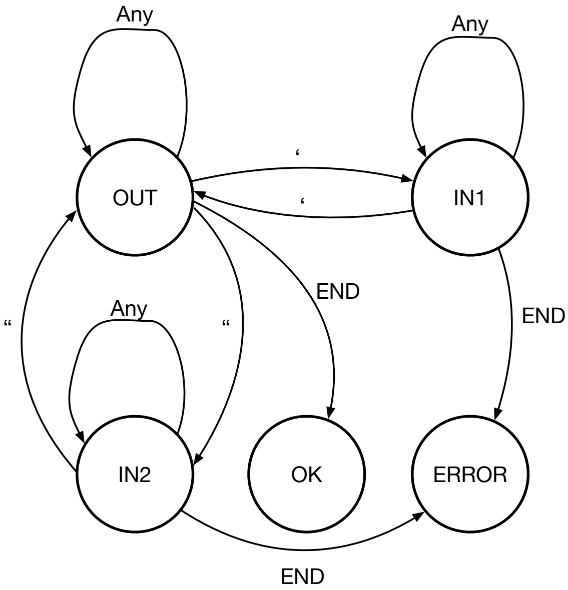
- **Задача**: на вход алгоритма подаётся последовательность символов. Назовём *строкой* любую подпоследовательность символов, начинающуюся на знак одиночной кавычки или двойной кавычки и заканчивающейся ей же. Внутри строк могут находиться любые символы, кроме завершающего.
- Нужно определить корректность входной последовательности.
- Примеры:

```
'abracadabra'-корректно
'abra"shvabra cadabra' - корректно
"" - корректно
"abra'shravra' - некорректно
```

Традиционный способ решения

```
bool check(string s) {
    int ps = 0;
    if (s.size() == 0) return true;
    while (ps < s.size()) {
        if (s[ps] == '\'') {
            while (++ps < s.size() && s[ps] != '\'')
                ;
            if (s[ps] == '\'') ps++;
            else if (ps >= s.size()) return false;
        } else if (s[ps] == '"') {
            while (++ps < s.size() && s[ps] != '"')
                ;
            if (s[ps] == '"') ps++;
            else if (ps >= s.size()) return false;
        } else {
            ps++;
        }
    }
    return true;
}
```

Конечный автомат



Конечный автомат

```
bool DFA(string const &s) {
    enum {OUT, IN1, IN2} state = OUT;
    for (auto c: s) {
        if (state == IN1 && c == '\'') state = OUT;
        else if (state == IN2 && c == '"')  state = OUT;
        else if (state == OUT && c == '\'') state = IN1;
        else if (state == OUT && c == '"')  state = IN2;
    }
    return state == OUT;
}
```

Абстракции.
Интерфейс абстракции.

Понятие абстракции

Как только появляются *объекты*, появляются *абстракции* — механизм разделения сложных объектов на более простые, без деталировки подробностей разделения.

Функциональная абстракция — разделение функций, *методов*, которые манипулируют с объектами с их реализацией.

Интерфейс абстракции — набор методов, характерных для данной абстракции.

Пример: абстракция последовательности

- **create** — создать объект последовательности. Атрибуты: для чтения или для записи?
- **destroy** — удалить объект.
- **get** — получить очередной элемент последовательности.
- **put** — добавить элемент в последовательность.

Уже прочитанный элемент второй раз не прочитается.

Пример: абстракция массива

- **create** — создать массив. Статический или динамический?

  ```
  int a[100]; // Статический
  int *b = calloc(100, sizeof(int)); // Динамический
  int *c = new int[100]; // Динамический
  ```

- **destroy** — Удалить массив. Статический или динамический?

  ```
  free(b);
  delete c; // можно и delete [] c
  ```

- **fetch** — Обратиться к элементу массива.

  ```
  int q1 = a[i];
  int q2 = b[i];
  int q3 = c[i];
  ```

Для массива основная операция — это доступ к элементу. Она выглядит одинаково для всех представлений.

Абстракция стек

Одна из удобных абстракций — стек. Он должен предоставлять нам методы:

- **create** — создание стека. Может быть, потребуется аргумент, определяющий максимальный размер стека.
- **push** — занесение элемента в стек. Размер стека увеличивается на единицу. Занесённый элемент становится *вершиной стека*.
- **pop** — извлечь элемент, являющийся вершиной стека и уменьшить размер стека на единицу. Если стек пуст, то значение операции не определено.
- **peek** — получить значение элемента, находящегося на вершине стека, не изменяя стека. Если стек пуст, значение операции не определено.
- **empty** — предикат истинен, когда стек пуст.
- **destroy** — уничтожить стек.

Абстракция *множество*

Множество есть совокупность однотипных элементов, на которых определена операция сравнения
Обозначение: списком значений внутри фигурных скобок.
Пустое множество: $s = \{\}$.

- **insert** — добавление элемента в множество.

```
{1,2,3}.insert(5) -> {1,2,3,5}
{1,2,3}.insert{2} -> {1,2,3}
```

- **remove** — удалить элемент из множества.

```
{1,2,3}.remove(3) -> {1,2}
{1,2,3}.remove(5) -> не определено.
```

- **in** — определить принадлежность множеству.

```
{1,2,3}.in(2) -> true
{1,2,3}.in(5) -> false
```

- **size** — определить количество элементов в множестве

Рекурсия.
Принцип *разделяй и властвуй*.

Числа Фибоначчи. Рекуррентная форма

$\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots\}$

Рекуррентная форма определения:

$$F_n = \begin{cases} 0, & \text{если n} = 0, \\ 1, & \text{если n} = 1, \\ F_{n-1} + F_{n-2}, & \text{если } n > 1. \end{cases}$$

Много алгоритмов первично определяются рекуррентными зависимостями.

Рекуррентность и рекурсия

Рекуррентная форма → рекурсивный алгоритм

```
int fibo(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fibo(n-1) + fibo(n-2);
}
```

Три вопроса:

1. Корректен ли он?
2. Как оценить его сложность?
3. Как его ускорить?

Рекуррентность и рекурсия

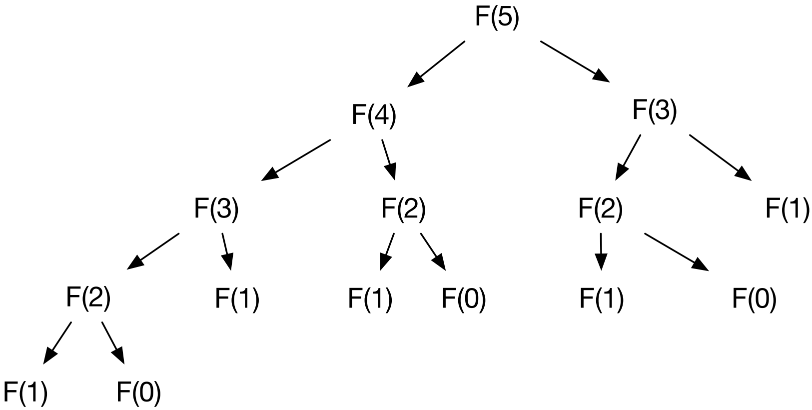
Первый вопрос.
Корректность предложенного алгоритма можно доказать чисто математически, по индукции.

- Из $n = 0$ следует $F_0 \leftarrow 0$
- Из $n = 1$ следует $F_1 \leftarrow 1$
- Из $n = 2$ следует $F_2 \leftarrow F_1 + F_0$
- Из произвольного $n > 1$ следует $F_n = F_{n-1} + F_{n-2}$

Первые два высказывания — база индукции. Третье — наблюдение за тем, что для какого-то из $n > 1$ условие выполняется. Четвёртое — индуктивный переход.

Дерево вызовов функции для $n = 5$

Второй вопрос — сложность алгоритма.



Оценка времени вычисления алгоритма

Пусть $t(n)$ — количество вызовов функции для аргумента n.

$$
\begin{aligned}
t(0) &= 1 \\
t(0) &> F_0 \\
t(1) &= 1 \\
t(1) &= F_1
\end{aligned}
$$

Для $n > 1$

$$t(n) = t(n-1) + t(n-2) \geqslant F_n.$$

Оценка требуемой для исполнения памяти

Требуемая память для исполнения характеризует *сложность алгоритма по памяти*.

- Каждый вызов функции создаёт новый *контекст функции* или *фрейм вызова*.
- Каждый *фрейм вызова* содержит все аргументы, локальные переменные и служебную информацию.
- Максимально создаётся количество фреймов, равное глубине рекурсии.
- Сложность алгоритма по занимаемой памяти равна $O(N)$.

Определение порядка числа вызовов

Числа Фибоначчи удовлетворяют отношению

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \varphi,$$

где $\varphi = \dfrac{\sqrt{5}+1}{2}$, то есть, $F_n \approx C \times \varphi^n$.
Сложность этого алгоритма есть $\Theta(\varphi^N)$.

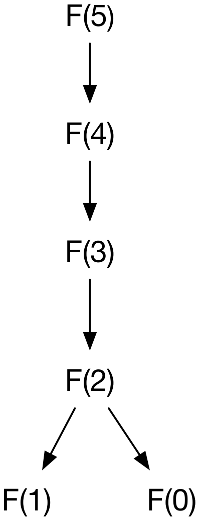
Как ускорить?

Проблема в том, что мы много раз повторно вычисляем значение функции от одних и тех же аргументов.

Третий вопрос: можно ли уменьшить сложность по времени алгоритма, то есть ускорить алгоритм?
Вводим добавочный массив.

```
int fibo(int n) {
  const int MAXN = 1000;
  static int c[MAXN];
  if (n == 0) return 0;
  if (n == 1) return 1;
  if (c[n] > 0) return c[n];
  return c[n] = fibo(n-1) + fibo(n-2);
}
```

Дерево вызовов модифицированной функции для $n = 5$



Числа в алгоритме и их представление в исполнителе

Что есть число в алгоритме?

Значение функции `fibo(n)` растёт слишком быстро и уже при небольших значениях n число выйдет за пределы разрядной сетки любой архитектуры.

- Алгоритм `fibo` оперирует с числами.
- Программа, реализующая алгоритм `fibo` имеет дело с *представлениями* чисел.

Любой исполнитель алгоритма имеет дело не с числами, а с их представлениями.

Проблема с представимостью данных

В реальных программах имеются ограничения на операнды машинных команд. X86,X64 $\to$ `int` есть 32 бита, `long long` есть 64 бита.

На 32-битной архитектуре сложение двух 64-разрядных $\to$ сложение младших разрядов и прибавление бита переноса к сумме старших разрядов. Три машинных команды.

X86: сложение: 32-битных $\approx$ 1 такт; 64-битных $\approx$ 3 такта.
X64: сложение: 32-битных $\approx$ 1 такт; 64-битных $\approx$ 1 такт.

X86: умножение: 32-битных $\approx$ 3-4 такта; 64-битных $\approx$ 15-50 тактов.
X64: умножение: 32-битных $\approx$ 3-4 такта; 64-битных $\approx$ 4-5 тактов.

Представление длинных чисел

- Длинные числа имеют представление в виде *цифр* в позиционной системе счисления, каждая из которых есть элементарный тип данных.
- Все операции производятся в системе счисления, зависящей от мощности множества представимых цифр.
- Мы привыкли использовать по одному знаку на десятичную цифру.
- Аппаратному исполнителю удобнее работать с длинными числах в системе счисления по основаниям, большим 10 $(2^8, 2^{16}, 2^{32}, 2^{64})$.

`(n)-числа`

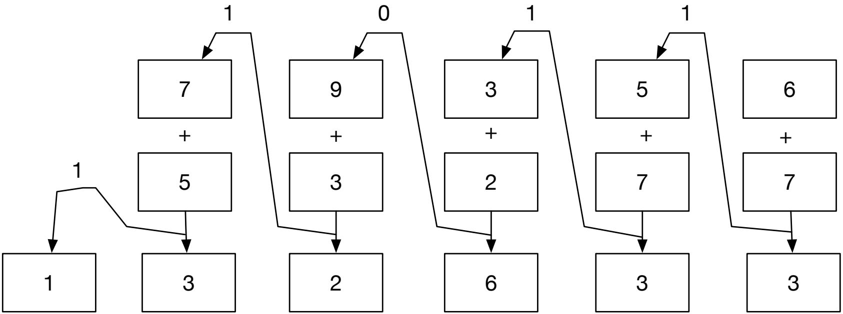
- **Определение:** `(n)-числа` — те числа, которые требуют не более `n` элементов элементарных типов (цифр) в своём представлении.
- `int` есть `(1)-числа` для 32-битной архитектуре, `long long` — `(2)-числа`.
- Большие числа требуют представления в виде массивов из элементарных типов.
- Основание системы счисления `R` для каждой из цифр представления должно быть представимо элементарным типом данных аппаратного исполнителя.

Сложность операций над длинными числами

Сколько операций потребуется для сложения двух `(n)-чисел`?

Сложность операций над длинными числами

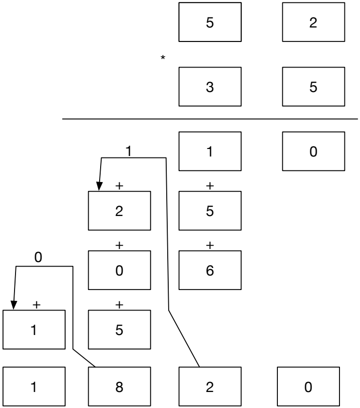
Сколько операций потребуется для сложения двух `(n)-чисел`?



$O(n)$

Сложность операций над длинными числами

Как умножать длинные числа?
Школьный алгоритм:



$O(n^2)$

Алгоритм быстрого умножения

Можно ли быстрее?

Алгоритм быстрого умножения

Можно ли быстрее?

Да. Используя принцип *разделяй и властвуй*.
Быстрый алгоритм умножения был изобретён Гауссом в 19 веке и переизобретён Анатолием Карацубой в 1960-м году.

Разделим число (n) на две примерно равные половины:

$$N_1 = Tx_1 + y_1$$
$$N_2 = Tx_2 + y_2$$

При умножении в столбик

$$N_1 \times N_2 = T^2 x_1 x_2 + T(x_1 y_2 + x_2 y_1) + y_1 y_2.$$

Это — четыре операции умножения и три операции сложения.
Число T определяет, сколько нулей нужно добавить к концу числа в соответствующей системе счисления.

Алгоритм Карацубы

Алгоритм Карацубы находит произведение по другой формуле:

$$N_1 \times N_2 = T^2 x_1 x_2 + T((x_1 + y_1)(x_2 + y_2) - x_1 x_2 - y_1 y_2) + y_1 y_2$$

$N_1 = 56,\ N_2 = 78,\ T = 10$

$$x_1 = 5, y_1 = 6$$
$$x_2 = 7, y_2 = 8$$
$$x_1 x_2 = 5 \times 7 = 35$$
$$(x_1 + y_1)(x_2 + y_2) = (5 + 6)(7 + 8) = 11 * 15 = 165$$
$$y_1 y_2 = 6 \times 8 = 48$$
$$N_1 \times N_2 = 35 * 100 + (165 - 35 - 48) * 10 + 48 = 4368$$

Три операции умножения и шесть сложения.

Основная теорема о рекурсии.

Оценка асимптотического времени алгоритма

Как определить, какой порядок сложности будет иметь рекурсивная функция, не проводя вычислительных экспериментов?

Рекурсия есть разбиение задачи на подзадачи с последующей консолидацией результата.
Пусть

- a — количество подзадач
- размер каждой подзадачи уменьшается в b раз и становится $\left\lceil \frac{n}{b} \right\rceil$.
- Сложность консолидации пусть есть $O(n^d)$.

Время работы такого алгоритма, выраженное рекуррентно, есть

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$$

Основная теорема о рекурсии

Пусть $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$ для некоторых $a > 0, b > 1, d \geqslant 0$. Тогда

$$T(n) = \begin{cases} O(n^d), & \text{если } d > \log_b a, \\ O(n^d \log n), & \text{если } d = \log_b a, \\ O(n^{\log_b a}), & \text{если } d < \log_b a. \end{cases}$$

Оценка сложности алгоритма Карацубы

- Коэффициент порождения задач $a = 3$.
- Коэффициент уменьшения размера подзадачи $b = 2$.
- Консолидация решения производится за время $O(n) \to d = 1$

Так как $1 < \log_2 3$, то это третий случай теоремы $\to$ сложность алгоритма есть $O(N^{\log_2 3})$.

Операция умножения чисел (n) при умножении в столбик имеет порядок сложности $O(n^2)$.

Много операций сложения $\to$ при малых N выгоднее «школьный» алгоритм.

Ещё о сложности

Введём вектор-столбец $\begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, состоящий их двух элементов последовательности Фибоначчи и умножим его справа на матрицу $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}.$$

Для вектора-столбца из элементов F_{n-1} и F_n умножение на ту же матрицу даст:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n-1} + F_n \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}.$$

Таким образом,

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

Возведение в степень

Для нахождения $n-$го числа Фибоначчи достаточно вычислить $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$.

Можно ли возвести число в n-ю степень за меньше, чем $n-1$ число операций?

Быстрое вычисление степеней.

Возведение числа в квадрат есть умножение числа на себя.

$$x^{16} = (x^8)^2 = ((x^4)^2)^2 = (((x^2)^2)^2)^2$$

$$x^{18} = (x^9)^2 = (x^8 \cdot x)^2 = (((x^2)^2)^2 \cdot x)^2$$

Рекуррентная формула

$$x^n = \begin{cases} 1, & \text{если } n = 0 \\ (x^{\frac{n}{2}})^2 & \text{если } n \neq 0 \wedge n \pmod 2 = 0 \\ (x^{n-1}) \times x & \text{если } n \neq 0 \wedge n \pmod 2 \neq 0 \end{cases}$$

Рекурсивная функция быстрого умножения

SomeType — некий тип данных.

```
SomeType pow(SomeType x, int n) {
   if (n == 0) return (SomeType)1;
   if (n % 2 != 0) return pow(x, n-1) * x;
   SomeType y = pow(x, n/2);
   return y*y;
}
```

Оценка сложности алгоритма быстрого умножения

$$25_10 = 11001_2$$

- n — нечётное? $\to$ обнуление последнего разряда.
- n — чётное? $\to$ вычёркивание последнего разряда.
- каждую из единиц требуется уничтожить, не изменяя количества разрядов.
- каждый из разрядов требуется уничтожить, не изменяя количества единиц.

Сложность есть $O(\log N)$.

Спасибо за внимание.

Следующая лекция —
жадные алгоритмы.