Алгоритмы и структуры данных

Лекция 2.
Рекурсия.

Сергей Леонидович Бабичев

Содержание лекции

- Рекурсия. Принцип *разделяй и властвуй*.
- Числа и их представление.
- Основная теорема о рекурсии.
- Быстрое вычисление степеней.

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 $\rightarrow$ рекурсивный алгоритм

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

Рекуррентность и рекурсия

Рекуррентная форма $\rightarrow$ 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

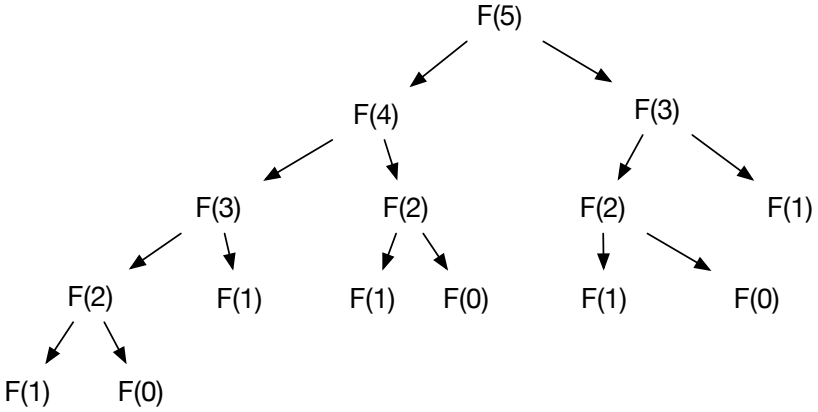
Первый вопрос.
Корректность доказывается по индукции.

1. Из $n = 0$ следует $F_0 \leftarrow 0$
2. Из $n = 1$ следует $F_1 \leftarrow 1$
3. Из $n = 2$ следует $F_2 \leftarrow F_1 + F_0$
4. 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 $\Theta(N)$.

Снова числа Фибоначчи

В математике для $n-$го числа Фибоначчи, как для многих рекуррент, есть *производящая функция*.

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

Известна константа $\Phi = \frac{\sqrt{5}+1}{2}$. Тогда

$$F_n = \frac{\Phi^n - \Phi^{-n}}{\sqrt{5}}.$$

Сложность этого алгоритма есть $\Theta(\Phi^N)$.

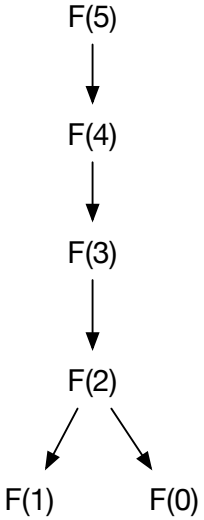
Как ускорить?

Почему так медленно?
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 $\rightarrow$ `int` есть 32 бита, `long long` есть 64 бита.

На 32-битной архитектуре сложение двух 64-разрядных $\rightarrow$ 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 исполнителя.
- 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 двоичными числами 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.

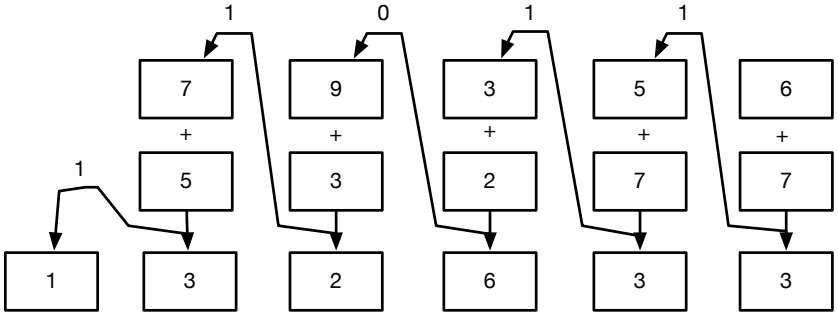
- int есть (1)-числа для 32-битной архитектуры, long long — (2)-числа.
- long long есть (1)-числа и для 32-битной, и для 64-битной архитектур.
- Большие числа требуют представления в виде массивов из элементарных типов.
- Основание системы счисления R для каждой из цифр представления должно быть представимо 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?

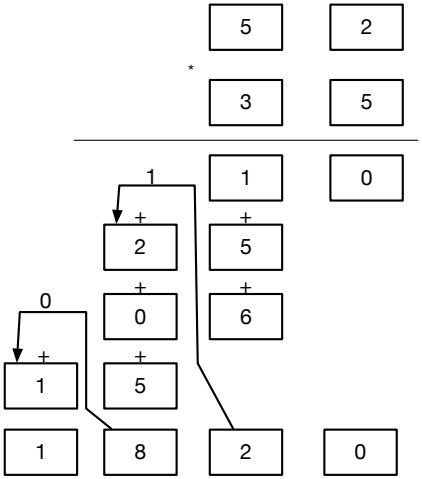


$$\Theta(n)$$

Сложность операций над длинными числами

Как умножать длинные числа?

Школьный алгоритм:



$$\Theta(n^2)$$

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.

Сложность алгоритма Карацубы

- Стал ли алгоритм иметь меньшую сложность?
- Проделаем несколько экспериментов чтобы определить количество умножений и сложений для больших чисел.
- Как определить сложность такого алгоритма? Можно попробовать выразить количество операций одного уровня через количество операций другого уровня. Обозначим количество операций умножения уровня k за m_k, а количество операций сложения — за a_k.
- Чтобы произвести одну операцию умножения, требуется провести три операции умножения и шесть операций сложения предыдущего уровня.
- Следующий уровень определяет числа, с количеством чанков в два раза большим, чем предыдущий уровень.
- Количество чанков равно 2^k, где k — номер уровня.
  Можно записать это так, первый элемент пары — число умножений, второй — число сложений:

$$m_k = (3m_{k-1}, 6a_{k-1}). \tag{1}$$

Рекурренты

- Как определить количество операций сложения в зависимости от количества операций предыдущего уровня?
- Чему равно a_k в зависимости от a_{k-1} и m_{k-1}? Перефразируем.
- Чтобы совершить операцию умножения чисел на уровне k нужно 3 операции умножения чисел уровня $k-1$ и 6 операций сложения чисел уровня $k-1$.
- Для сложения чисел на уровне k нужно в два раза больше операций, чем для такого же сложения, на уровне $k-1$. Таким образом,

$$a_k = (0m_{k-1}, 2a_{k-1}). \tag{2}$$

Всё это можно записать вместе:

$$\begin{cases} m_k &= (3m_{k-1}, 6a_{k-1}), \text{ если } k > 0; \\ a_k &= (0m_{k-1}, 2a_{k-1}), \text{ если } k > 0; \\ m_0 &= (1, 0); \\ a_0 &= (0, 1) \end{cases} \tag{3}$$

Исследуем рекурренту от двух переменных

- Найдём руками несколько первых членов этой рекуррентной последовательности.

$$
\begin{aligned}
(m_0, a_0) &= ((1,0),(0,1)) \\
(m_1, a_1) &= ((3,6),(0,2)) \\
(m_2, a_2) &= ((9,30),(0,4)) \\
(m_3, a_3) &= ((27,114),(0,8) \\
(m_4, a_4) &= ((81,390),(0,16) \\
&\dots \\
(m_{10}, a_{10}) &= ((59049,348150),(0,1024))
\end{aligned}
\tag{4}
$$

Исследуем рекурренту

- Для чисел с 1024 чанками количество операций умножения оказалось 59049, а операций сложения — 348150.
- Операция целочисленного умножения в несколько десятков медленнее операции целочисленного сложения.
- Что будет с точки зрения простого количества операций? К чему стремится это количество?
- Количество операций умножения каждый раз увеличивается в три раза при увеличении в два раза количества чанков.
- Отношение количества умножений к количеству чанков при стремлении количества чанков к бесконечности тоже стремится к бесконечности.

$$\lim_{k \to \infty} \frac{m_k}{2^k} = \infty. \tag{5}$$

Если наше отношение мы обозначим как $f(k) = \frac{m_k}{2^k}$, то наша задача — найти такую функцию $g(k)$, которая при стремлении k к бесконечности стремится примерно также.

Находим нужную функцию

Такая функция есть и это

$$g(k) = \frac{3^k}{2^k}. \tag{6}$$

Можно сказать, что

$$\lim_{k \to \infty} \frac{f(k)}{g(k)} = 1. \tag{7}$$

Учитываем числа сложений

- Наблюдая над ростом m_k и a_k при росте k, видим закономерность, что их отношение тоже к чему-то стремится (оно стремится к $\frac{1}{6}$, но мы не будем это доказывать).
- Если нас интересует скорость роста *общего количества операций*, то есть *суммы* $m_k + a_k$, то и тут имеется предел:

$$\lim_{k\to\infty} \frac{m_k + a_k}{g(k)} = c. \tag{8}$$

  Здесь c — константа.
- Этот факт докажет основная теорема о рекурсии.

Итоговая сложность

- Итак, функция количества операций в алгоритме Карацубы растёт примерно с такой же скоростью (предел при стремлении аргумента к бесконечности равен единице), как функция $g(k) = \dfrac{3^k}{2^k}$.
- Количество чанков на уровне k равно 2^k, сведём к явным степеням от него:

$$\Theta\left(\frac{3^k}{2^k}\right) = \Theta(N^{\log_2 3}) \approx \Theta(N^{1.58}). \tag{9}$$

- Школьный алгоритм имеет сложность $\Theta(N^2)$. При $N = 1024$ школьный в 18 раз медленнее.
- В реальной жизни, где умножение — достаточно длительная операция, разница будет в сотню раз.

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.

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

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

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

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

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? $\rightarrow$ обнуление последнего разряда.
- n — чётное? $\rightarrow$ 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 и линейные алгоритмы.