Разработка и анализ алгоритмов

Лекция 1

Сергей Леонидович Бабичев

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.

Исполнители

Наш основной исполнитель — язык `C`.

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

Операции

Пример. Неэлементарная операция языка: цикл `for`.

```
int a[10];
// Initializing a
// ...
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 и сложность

- Целые числа — двоичное представление. `int x; unsigned y;`
- Простые элементарные операции: сложение, вычитание, присваивание, побитовые... `x += y; x &= 0x800;`
- Операции посложнее: сравнение, условное присваивание `x = (y > 0);`
- Сложные элементарные операции: целочисленное умножение. `x *= y;`
- Самые сложные: деление не на степень двойки, нахождение остатка, совершение перехода. `x = y % 10; if (x==6) y = 10;`

Представление типов и сложность

Прямолинейно закодированные

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

выполняются много медленнее, чем

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

так как для второго варианта есть готовая машинная команда.

Компиляторы об этом знают и пытаются провести *эквивалентные* преобразования.

Аппаратные исполнители

Популярные архитектуры:

- 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. Серверы (в том числе в TOP5). Начала успешно использоваться для ноутбуков и настольных компьютеров.

Модулярная арифметика

```
unsigned short x = 40000; // x - 16 bits.
x = x + x;
printf("%hu", x);
```

Чему равен x?

Модулярная арифметика

```
unsigned short x = 40000; // x - 16 bits.
x = x + x;
printf("%hu", x);
```

Чему равен x?

x = 14464.
Почему?

$$x \in [0..2^{16}) = [0..65535).$$

Биты нумеруются от 0 до 15 справа налево.
Все биты результата старше 15-го отсекаются.

Модулярная арифметика

Вся компьютерная арифметика основана на тождествах:

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

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

```
unsigned int x,y,z; // 32 bits. [0..4294967295]
...
z = x * y;
```

$$z = (x * y) \pmod{2^{32}}$$

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.

- Предметы разрезать на куски нельзя! Разрешим — задача будет иметь простое решение.
- Для решения задачи достаточно перебрать все возможные подмножества из N предметов, посчитать их общий объём, и если он удовлетворяет нас, то посчитать общую стоимость.
- Этот алгоритм гарантирует, что мы нужное подмножество пропущено не будет.
- Сколько таких подмножеств? 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.

- Предметы разрезать на куски нельзя! Разрешим — задача будет иметь простое решение.
- Для решения задачи достаточно перебрать все возможные подмножества из N предметов, посчитать их общий объём, и если он удовлетворяет нас, то посчитать общую стоимость.
- Этот алгоритм гарантирует, что мы нужное подмножество пропущено не будет.
- Сколько таких подмножеств? 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 (подмножество).
4. Рассмотрим все подмножества, начиная от $000\ldots000$ до $111\ldots111$, для каждого из них подсчитаем значение суммарного объёма.
    1. Если суммарный объём расстановки не превосходит объёма рюкзака, то подсчитывается суммарная стоимость и сравнивается с достигнутым ранее максимумом стоимости.
    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 $O(2^N)$, 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

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

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-сложных*.
- Быстрое (полиномиальное) точное решение таких задач (пока?) не найдено.
- Эта задача к тому же *NP-полная*.
- Если будет найдено решение одной из *NP-полных* задач, то будут решены все задачи из этого класса.
- Сейчас их решают приближённо.

Абстракции.
Интерфейс абстракции.

Решение сложных проблем

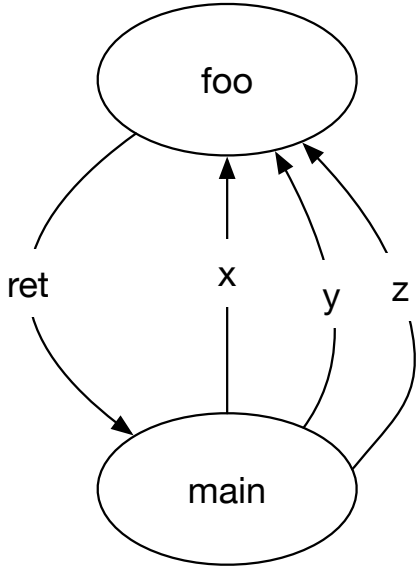
- Рассмотрим один уровень погружения в задачу.
- В дисциплине «системный анализ» решение задачи есть
  1. Декомпозиция задачи, разбиение её на подзадачи.
  2. Анализ, решение подзадач любыми методами.
  3. Синтез результатов.
- *Система* — совокупность связанных и взаимодействующих элементов.
- *Компонент* может быть неделимым или может делиться на *подсистемы*.
- *Подсистема* — часть системы, определяемая связями и отношениями.
- Система состоит из подсистем, каждая из которых может рассматриваться как система.

Две программы

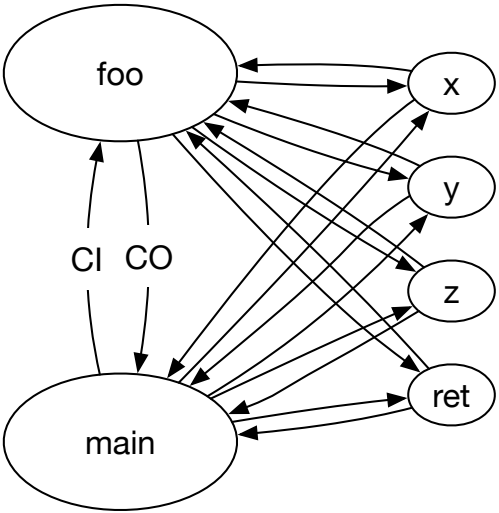
```
// Program A
int foo(int x, double y, char z) {
  int ret = x + y*2 + (z - '0');
  return ret;
}
int main() {
  int x = 10;
  double y = 20.0;
  char z = '7';
  int k = foo(x,y,z);
  printf("k=%d\n", k);
}
```

```
// Program B
int x, ret;
double y;
char z;
void foo() {
  ret = x + y*2 + (z-'0');
}
int main() {
  x = 10;
  y = 20.0;
  z = 'z';
  foo();
  printf("ret=%d\n", ret);
}
```

Структурная схема программы A



Структурная схема программы B



Сложность программы

- Блоков данных теперь шесть, кода — два.
- Если блок данных доступен блоку кода на чтение, то стрелка направлена от блока данных к блоку кода, если на запись, то от блока кода к блоку данных.
- Все блоки данных доступны всем блокам кода на чтение, и на запись
- Блоки кода связаны между собой отношениями передачи управления.
- После того, как `main` вызывает `foo`, происходит передача управления (`CI`), после завершения `foo` управление возвращается `main` (`CO`).
- Сложность программы — комбинаторная функция общего количества различных комбинаций всевозможных связей. Она пропорциональна N^2, где N — количество блоков в программе.
- Во второй программе добавление переменной `t` добавит связи от неё до всех блоков кода, а добавление функции `bar` добавит связи от неё до всех глобальных переменных.
- Для уменьшения сложности надо уменьшить либо само количество блоков, либо количество связей между ними, путём запрета определённых связей.

Строительные блоки программ

- Написание программы — решение задачи.
- Задача делится на подзадачи.
- Подзадачи делятся на подподзадачи ...
- К каким подзадачам нам стремиться?

Понятие абстракции

Для разбиения большой задачи на подзадачи удобно применять *объекты*.

Появляются *объекты* — появляются *абстракции* — механизм разделения сложных объектов (*структур данных*) 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.

Метод может быть применён к *объекту* для изменения состояния объекта или для получения неких *свойств* объекта.

Хорошо определённый объект имеет конечное количество методов, которые он поддерживает. Одна из типичных ошибок проектирования — нагружать объект большим количеством методов.

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.
Алгоритмы, рассчитанные на обработку последовательностей, могут иметь сложность по памяти $O(1)$ и по времени $O(N)$.

Пример: абстракция массива

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

```
way 1: int a[100];
way 2: int *b = calloc(100, sizeof(int));
```

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

```
way 1: // not required
way 2: free(b);
```

- **fetch** — обратиться к элементу массива. Основная операция (метод).

```
way 1: int q1 = a[i];
way 2: int q2 = b[i];
```

Пример: абстракция *вектор*

- **create** — создать вектор. Размер вектора можно изменять. Начальный размер можно задать. Вектор может быть и пустым.
- **destroy** — удалить вектор.
- **get/put** — обратиться к элементу вектора на чтение/запись. Во втором случае нам должны предоставить *l-значение*.
- **resize** — изменить размер вектора на требуемый.
- **push** — расширить вектор на один элемент и положить туда передаваемое значение.
- **pop** — уменьшить размер вектора на один элемент и вернуть значение того элемента, который там находился.
- **size** — вернуть количество элементов вектора.
- **empty** — предикат, возвращающий истину, если вектор пуст.

Классификация методов

При реализации абстракций различают методы *полные* или *тотальные* и *частичные*.

- **Тотальные** методы абстракции успешно исполняются при любом начальном состоянии объекта. Для вектора, например, это операции `push`, `empty` и другие.
- **Частичные** методы исполняются только при конечном количестве состояний объекта. Для вектора это, например `pop`.

Абстракция *стек*

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

- **create** — создать стек. Может быть, потребуется аргумент, определяющий максимальный размер стека.
- **push** — занести элемент 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 *очередь*

Она предоставляет понятия *головы* и *хвоста* очереди и методы:

- **create** — создать очередь. Иногда эффективно иметь возможность иметь очередь определённого размера.
- **push** — занести элемент в очередь. Размер очереди увеличивается на единицу. Занесённый элемент становится *головой очереди*.
- **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 на равенство.
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) -> ? or {1,2,3}
```

- **in** — определить принадлежность множеству.

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

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

Абстракция *список*

Она предоставляет понятия *головы*, *текущего* списка, *следующего* и *следующего* элементов и методы:

- **create** — создать список и вернуть его голову.
- **insert** — занести элемент в определённое место списка. Варианты: в начало, в конец, перед текущим элементом, после текущего элемента.
- **find** — найти элемент с заданным значением.
- **empty** — предикат. Истинен, когда список пуст.
- **destroy** — уничтожить очередь.

Взаимоотношение абстракций

- Уровень абстрагирования может быть разным.
- Некоторые абстракции можно выражать через другие.
- Например, стек и очередь можно реализовать через списки.
- Удобно создать *базис* абстракций для реализации сложных алгоритмов через более простые.
- Такой базис мы и будем создавать.

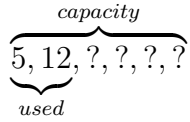
Ещё о сложности. Амортизационный анализ.

Реализация вектора

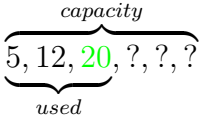
- Основная задача вектора — максимально быстрая операция `fetch`.
- Вторая задача — быстрое исполнение `push` и `pop`.
- Реализация вектора в виде связного списка даст сложность `push` и `pop` в $O(1)$, но `fetch` в $O(N)$, где N — размер вектора.
- Требуется реализация с использованием массива.

Реализация вектора

`capacity` — число элементов, под которые заказана память, `used` — число реально хранящихся элементов.

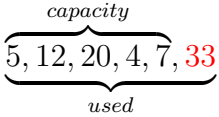


Операция `push_back` увеличивает `used` на 1 и размещает очередной элемент в новой позиции.

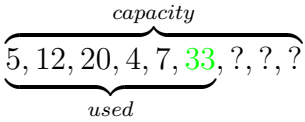


Реализация вектора

Если `used` должен стать больше `capacity`, то массив расширяется.



становится



Реализация вектора: сложность операции push_back

- Мы видим два варианта развития событий:
  - `fast_path`: $used < capacity \to T = O(1)$.
  - `slow_path`: $used \geqslant capacity \to T = O(new\_capacity + used + used)$.
- Второй вариант встречается реже, но имеет большую сложность.
- Выгоден ли такой алгоритм?

Реализация вектора: сложность операции push_back

- Пусть начальный размер вектора s, и каждый раз после переполнения он увеличивается на d.
- Тогда при добавлении n элементов:

$$T = O\left(\overbrace{\underbrace{1 + \cdots + 1}_{s} + (3s + d) + \underbrace{1 + \cdots + 1}_{d} + (3(s + d) + d)) + \ldots}^{n}\right)$$

- Пусть k — количество «длинных» операций.
- Тогда итоговый размер массива $n = s + kd$.
- Общее количество операций

$$N(k) = s + (3s + d) + d + (3s + 4d) + d + \cdots + 3(s + (k - 1)d + d)$$

Матожидание сложности операции, приведённое к одной операции

$$\lim_{k \to \infty} \frac{N(k)}{n(k)} = k.$$

Реализация вектора: другая стратегия операции push_back

- Пусть начальный размер вектора s, и каждый раз после переполнения он увеличивается в $q > 1$ раз.
- Тогда при добавлении n элементов:

$$T = O\left(\overbrace{\underbrace{1 + \cdots + 1}_{s} + (sq + s + s) + \underbrace{1 + \cdots + 1}_{d} + (sq^2 + sq + sq) + \ldots}^{n}\right)$$

- Пусть k — количество «длинных» операций.
- Тогда итоговый размер массива $n = sq^k$.
- Общее количество операций

$$N(k) = s + (s(q + 2)) + sq - s + (sq(q + 2)) + sq^2 - sq + \ldots$$

Матожидание сложности операции, приведённое к одной операции

$$\lim_{k \to \infty} \frac{N(k)}{n(k)} = 1.$$

Амортизационный анализ

- При исполнении алгоритма имеются операции двух типов: длинные и короткие.
- Длинная операция исполняется достаточно редко, но после её исполнения появляется возможность исполнить много коротких, «размазывая» или «амортизируя» её сложность.
- Мы применили «аггрегирующий» анализ, через матожидание.
- Ещё используют «метод предоплаты» и «метод потенциалов».

Амортизационный анализ: *метод предоплаты*

Другие названия: *метод монеток*, *метод бухучёта*.

- Рассмотрим вектор при $q = 2$.
- Предположим, что мы заводим счёт в банке и каждая операция изменяет этот счёт либо добавляя d_i единиц, либо снимая w_i единиц.
- Требуется найти значения d_i и w_i такие, что при исполнении алгоритма счёт не становится отрицательным, а после его исполнения — близок к нулю.
- Для простых операций будем класть пять монеток.
- Что происходит в сложных:
  - Одну монетку мы кладём на будущее.
  - Мы заказываем память $2n$ и на это уйдут по две монетки из резерва.
  - Мы копируем старый массив в новый и на это уйдут по монетке их резерва.
  - Мы уничтожаем старый массив длины n за время $O(n)$, израсходовав по одной монетке.
- Ни в один момент времени не имеется отрицательного числа монет.