<u>План</u> лекции 1

Введение в предмет. Литература. Модели вычислений. Сложность алгоритмов

1.1 Введение в предмет

ПРОБЕЛ В КОНСПЕКТЕ.

1.2 Литература

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974. **Перевод:** А. Ахо, Дж. Хопкрофт, Дж. Ульман, *Построение и анализ вычислительных алгоритмов.* М.: Мир, 1979.

2. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990. **Перевод:** Т. Кормен, Ч. Лейзерсон, Р. Ривест, *Алгоритмы: построение и анализ.* М.: МЦНМО, 2000.

1.3 Модели вычислений

1.3.1 Вычислимость

Алгоритм:

- Вход.
- Работа (*конечное* число *элементарных* (механических) шагов).
- Выход.

Вычисляет функцию / решает задачу.

Массовые задачи. Если задача имеет конкретное условие и ответ, алгоритм для нее — это нонсенс (например: вычислить $100+200$). Следовательно, нас интересуют задачи, где вариантов вопроса много (*массовые задачи*) (каждый вариант — *индивидуальная задача* (*instance*)). Наиболее интересны задачи, где вариантов бесконечно много: например, найти сумму двух натуральных чисел (чисел ведь бесконечно много). Массовая задача — это совокупность индивидуальных задач, т.е. пар (условие, решение); условия и решения можно считать битовыми строками. Нас будут интересовать задачи, для которых критерий того, является ли данная битовая строка правильным решением, может быть записан в виде, едином для всех условий (и строк, претендующих на то, чтобы быть решениями: например, задача о нахождении делителя: условие — целое число, правильное решение — делитель этого числа).

Модель вычислений — формализация понятия алгоритма. Чем проще модель вычислений — тем проще доказывать теоремы.

Тезис Чёрча. Интуитивное понятие вычислимости совпадает с вычислимостью согласно $<\ldots>$ (далее можно подставить любую разумную формализацию — например, РАМ, которой мы и будем пользоваться).

Замечание 1.1. У Чёрча фигурировала не РАМ, а другая, эквивалентная ей, модель.

Замечание 1.2. Существуют невычислимые (алгоритмически неразрешимые) задачи (мы с ними познакомимся позже).

РАМ (Равнодоступная Адресная Машина — Random Access Machine).

- Регистры — $r_0, r_1, r_2, \ldots$ — каждый может содержать целое число (инициализируются нулями).
- Нулевой регистр — специальный (будет видно ниже).
- Программа состоит из шагов (первый, второй, ...), каждый из которых — команда (см. ниже).
- Исполнение программы происходит пошагово (начиная с первого шага). Каждый раз мы переходим к следующему шагу (за исключением команд перехода J...).
- У программы есть *вход* (то, что ей дает пользователь; она может его прочесть) и *выход* (то, что она выдает).

- Команды (с картинкой перемещений):

READ	from input
WRITE	to output
LOAD a	$r_0 \longleftarrow a$
STORE a	$a \longleftarrow r_0$
ADD a	$r_0 \longleftarrow r_0 + a$
NEG	change sign
LSHIFT a	left bit shift of $\lvert r_0 \rvert$
RSHIFT a	right bit shift of $\lvert r_0 \rvert$
JUMP b	
JGTZ b	jump if $r_0 > 0$
HALT	

- Способы адресации (с картинкой: c, $[c]$, $[[c]]$) —

ПРОБЕЛ В КОНСПЕКТЕ.

Замечание 1.3. Компьютер — не РАМ, так как у него конечная память. Чтобы сделать из компьютера “настоящую” машину, которая может решить любую интуитивно алгоритмически разрешимую задачу, надо дополнить его устройством, производящим и подключающим новые карты памяти, когда это понадобится.

Пример 1.1 (MULT(c_1, c_2) для неотрицательных c_1, c_2).

Алгоритм:

Умножаем c_1 по очереди на каждый бит c_2.

Назначение регистров (k — номер итерации):

1. Первое число c_1.
2. Второе число c_2, затем $\lfloor c_2/2^{k-1} \rfloor$.
3. $\lfloor c_2/2^k \rfloor$. //Пока не обнулится; тогда — выход.
4. $c_1 \cdot (c_2 \mod 2^k)$ — копится ответ.
5. $c_1 \cdot 2^k$.

Программа:

1. READ
2. STORE [1]
3. STORE [5]
4. READ

5. STORE [2]
6. HALF //Эти шаги вычисляют
7. STORE [3] //k-й бит c_2 как $\lfloor c_2/2^{k-1}\rfloor - 2\lfloor c_2/2^k\rfloor\ldots$
8. ADD [3]
9. NEG
10. ADD [2]
11. JGT 13
12. JUMP 16

13. LOAD [4] //Раз он единица, добавляем $c_1 \cdot 2^k\ldots$
14. ADD [5]
15. STORE [4]

16. LOAD [5] //Вычисляем $c_1 \cdot 2^k$ для очередного $k\ldots$
17. ADD [5]
18. STORE [5]
19. LOAD [3] //Осталось ли что-то от c_2?..
20. JGT 5

21. LOAD [4]
22. WRITE
23. HALT

$\square$

1.3.2 Эффективность

Время работы РАМ.

- Время работы, конечно, зависит от количества шагов.
- Как считать каждый шаг?
  - за единицу — это называется unit cost,
  - за длину участвующих в нем чисел (если участвует $[[c]]$, надо не забыть учесть длину и c, и r_c, и r_{r_c}) — это называется logarithmic cost, поскольку длина числа n — это $\lceil \log |n| \rceil$ (наличие знака не учитываем — это всего один бит).
- Используемая память: $\max\limits_{\text{по времени}} \sum\limits_i \lceil \log r_i \rceil$.

Обычно мы будем иметь в виду logarithmic cost: он более «честный».

Часто мы будем оценивать не время работы РАМ (ее довольно скучно выписывать для каждого алгоритма), а количество некоторых элементарных операций (например, сравнений и перемещений элементов массива при его сортировке). Чтобы получить “настоящее” время работы, надо учесть время исполнения каждой из этих операций на РАМ.

Размер входа. Как мы помним, РАМ дается некий *вход* (который она может постепенно прочесть при помощи операции READ). Если мы при помощи РАМ решаем массовую задачу, то на вход мы подаем условие этой задачи: точнее, условие одной из входящих в нее индивидуальных задач.

Чем больше это условие, тем дольше, скорее всего, будет работать машина (перемножать 1024-битные числа сложнее, чем 2-битные). Мы будем интересоваться временем работы машины как функцией длины входа (то есть длины битового представления такого условия).

Вообще говоря, размер входа зависит от представления данных (например, если условие — граф, надо четко описывать, что мы понимаем под графом — матрицу смежности, список ребер или что-то третье).

Асимптотическое поведение алгоритма. Нас будет интересовать, как время работы алгоритма и другие поглощаемые им ресурсы зависят от размера входа:

- в наихудшем: $T(n) = \max\limits_{I \text{ размера } n} t(I)$,
- в среднем (*простейший вариант!*):

$$T(n) = \frac{\sum\limits_{I \text{ размера } n} t(I)}{\text{кол-во таких } I}.$$

Исполнение инструкций может занимать разное время на разных машинах: может быть

- мультипликативная константа (интерпретация инструкций);
- аддитивная константа на startup.

Поэтому разумно оценивать время с точностью до $O(\ldots)$. (Предупреждение: на практике *обе* упомянутые константы могут оказаться существенными: что толку с того, что алгоритм линейный, если startup занимает 2^{100} шагов?)

Например, алгоритм 1.1 работает время $O(\log\max\{c_1, c_2\} \cdot \log c_2) = O(n^2)$, поскольку размер входа $n = \lceil\log c_1\rceil + \lceil\log c_2\rceil$.

Иногда мы будем также оценивать время относительно других параметров входа (например, можно оценивать время работы алгоритма, умножающего квадратные булевы матрицы, относительно длины матрицы).