ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМОВ. БАЗОВЫЕ СТРУКТУРЫ ДАННЫХ

или

«ЛЕКЦИЯ, КОТОРУЮ НЕРЕАЛЬНО ДАВАТЬ НА ДИСТАНЦИОНКЕ»

Лекция 3

Раздел «Алгоритмы и структуры данных»

ВВЕДЕНИЕ

Последний раздел курса посвящен структурам данных, алгоритмам над ними и их использованию в различных задачах.

Структура данных — это способ хранения данных в компьютере, обеспечивающий их эффективное (по какому-либо критерию) использование в определенных задачах.

Знание типовых структур данных и умение использовать их в подходящей ситуации является качеством хорошего программиста.

Еще Никлаус Вирт определил для сообщества принцип «Алгоритмы + Структуры данных = Программы», а Линус Торвальдс отмечал, что в программах лучше уделять внимание структурам данных и отношениям между ними, а не специфичным кодовым конструкциям.

Все структуры данных имеют определенные над ними операции, чаще всего к ним относятся доступ, вставка, удаление, поиск. Каждая из операций по-разному эффективна в различных структурах данных.

Чтобы выбирать подходящую под задачу структуру данных, необходимо уметь оценивать алгоритмы, реализующие эти самые операции, причем результат измерения не должен зависеть от машины, на которой будет выполняться программа.

Если отойти от структур данных: задача чаще всего имеет несколько решений и не все алгоритмы равны с точки зрения их практической

пригодности. Чтобы сравнивать алгоритмы и выбирать лучший, людям нужно было разработать соответствующий теоретический аппарат. Этим аппаратом и стала теория алгоритмов.

Теория алгоритмов — смежная наука на стыке математики и информатики, изучающая общие характеристики алгоритмов и формальные модели их представления.

Основные разделы теории алгоритмов:

- проблемы формализации задач;
- классы сложности задач;
- асимптотический и амортизационный анализ (оценка потребления ресурсов при выполнении алгоритма).

Разработка эффективных алгоритмов играет главенствующую роль в задаче оптимизации компьютерных программ.

На этой лекции планировалось наглядно объяснить, как можно оценивать алгоритмы и как эти оценки потом будут использоваться при реализации и использовании различных структур данных, но сделать это в формате дистанционного обучения для меня является задачей со звездочкой. Попробуйте вникнуть в то, что будет дальше. Когда мы вернемся к очному формату занятий, я попробую выбить пару для повторного изучения материала.

Все вопросы по лекции можно присылать мне в телегу.

1. Основные определения

1.1. Понятие алгоритма

Классический алгоритм — конечная последовательность операций, понятная исполнителю, строгое исполнение которой решает поставленную задачу.

(Также существуют специальные виды алгоритмов, например вероятностные (стохастические), для них не всегда определены следующие свойства, но они позволяют решать поставленную задачу).

Свойства классических алгоритма:

- Дискретность — алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов. При этом для выполнения каждого шага алгоритма требуется конечный отрезок времени, то есть преобразование исходных данных в результат осуществляется во времени дискретно.
- Детерминированность (определённость). В каждый момент времени следующий шаг работы однозначно определяется состоянием системы. Таким образом, алгоритм выдаёт один и тот же результат (ответ) для одних и тех же исходных данных.
- Понятность — алгоритм должен включать только те команды, которые доступны исполнителю и входят в его систему команд.
- Завершаемость (конечность) — в более узком понимании алгоритма как математической функции, при правильно заданных начальных данных алгоритм должен завершать работу и выдавать результат за определённое число шагов. Однако довольно часто определение алгоритма не включает завершаемость за конечное время
- Массовость (универсальность). Алгоритм должен быть применим к разным наборам начальных данных.

- Результативность — завершение алгоритма определёнными задачей результатами.

1.2. Сложность алгоритма

Вычислительная сложность — функция зависимости ресурсов, затрачиваемых некоторым алгоритмом, от размера и состояния входных данных.

Основными ресурсами являются:

- процессорное время (абстракция — время);
- память (абстракция — пространство).

Говоря о сложности алгоритма, обычно речь идет не о конкретных величинах (секунды), а об абстрактных (количество элементарных операций), так как важно узнать именно порядок роста относительно размера входных данных, а не получить точную характеристику. Диапазон точных характеристик собирается после успешной реализации алгоритма для задачи определенной системы!

Зачастую память и время выполнения имеют обратно-пропорциональную зависимость, поэтому в зависимости от текущей задачи выбирают модификацию алгоритма, оптимальную по выбранному критерию (производительность, память, баланс).

Принцип trade-off — улучшение по сложности одного ресурса за счет ухудшения по другому (например, сохранение результатов вычисления выражений в памяти для улучшения производительности за счет того, что не нужно проводить повторные вычисления).

При анализе сложности алгоритма можно рассматривать сложность:

- в лучшем случае;
- в среднем случае;
- в худшем случае.

Почти всегда интересна сложность именно в худшем случае, она и называется *гарантированной сложностью*. Это логично, ведь если гарантированная сложность удовлетворяет условию задачи, то и в более хороших случаях сложность будет удовлетворительной.

Временная сложность — функция от размера входных данных *n*, равная максимальному/среднему/минимальному количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи.

Пространственная сложность — аналогично, но про объем памяти.

В дальнейшем мы будем уделять большее внимание временной сложности, т.к. это более актуально в современном программировании.

2. Асимптотические оценки

Асимптотический анализ — один из инструментов для определения сложности алгоритма.

Рассмотрение входных данных большого размера и оценка порядка роста времени работы алгоритма приводят к понятию *асимптотической сложности алгоритма*. Алгоритм с меньшей асимптотической сложностью является более эффективным для всех входных данных, за исключением лишь, возможно, данных малого размера.

Подобный подход к исследованию сложности позволяет оценить именно сложность алгоритма, без влияния сторонних факторов (быстродействия машины, используемых типов данных и т.д.).

Обычно нет нужды находить точную функцию, достаточно просто дать *асимптотическую оценку* — найти функцию, которая будет ограничивать функцию вычислительной сложности сверху и/или снизу с точностью до константы.

Оценки выражаются через *O-символику* (что-то подобное вы видели в матанализе).

Оценки бывают верхними (O), нижними (Ω) и точными (Θ). В основном нас будут интересовать верхние оценки.

!!! Все настолько замечательно, что порой обозначение $O(g(n))$ может использоваться и для верхних, и для нижних, и для точных оценок. Это стоит принять, чаще всего это никак не влияет на результат.

Нижние оценки позволяют понять, что нельзя реализовать алгоритм эффективнее, чем полученная нижняя оценка.

Верхние оценки позволяют сделать вывод, что алгоритм будет работать не хуже, чем полученная нижняя оценка.

Оценки одного вида различаются по «силе»: очевидно, что для алгоритма с найденной верхней оценкой $O(n^4)$ семейство функций $O(n^{100})$ тоже будет являться верхней оценкой, но эта оценка будет являться более слабой.

Обозначение	Интуитивное объяснение	Определение
$f(n) \in O(g(n))$	f ограничена сверху функцией g (С точностью до постоянного множителя) асимптотически	$\exists (C>0), n_0 : \forall (n>n_0)\ \lvert f(n)\rvert \le C\lvert g(n)\rvert$ или $\exists (C>0), n_0 : \forall (n>n_0)\ f(n) \le Cg(n)$
$f(n) \in \Omega(g(n))$	f ограничена снизу функцией g (С точностью до постоянного множителя) асимптотически	$\exists (C>0), n_0 : \forall (n>n_0)\ \lvert f(n)\rvert \ge C\lvert g(n)\rvert$
$f(n) \in \Theta(g(n))$	f ограничена снизу и сверху функцией g асимптотически	$\exists (C, C'>0), n_0 : \forall (n>n_0)\ C\lvert g(n)\rvert \le \lvert f(n)\rvert \le C'\lvert g(n)\rvert$
$f(n) \in o(g(n))$	g доминирует над f асимптотически	$\forall (C>0) \exists n_0 : \forall (n>n_0)\ \lvert f(n)\rvert < C\lvert g(n)\rvert$
$f(n) \in \omega(g(n))$	f доминирует над g асимптотически	$\forall (C>0) \exists n_0 : \forall (n>n_0)\ \lvert f(n)\rvert > C\lvert g(n)\rvert$
$f(n) \sim g(n)$	f эквивалентна g асимптотически	$\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 1$

Рисунок 1 — расшифровка асимптотических обозначений

! Попробуйте графически интерпретировать эти обозначения самостоятельно

Обозначение	Пояснение
$O(1)$	Устойчивое время работы, независимо от размера задачи.
$O(\log \log n)$	Очень медленный рост необходимого времени.
$O(\log n)$	Логарифмический рост — удвоение размера задачи увеличивает время работы на постоянную величину.
$O(n)$	Линейный рост — удвоение размера задачи удвоит и необходимое время.
$O(n \cdot \log n)$	Линеаризованный рост — удвоение размера задачи увеличит необходимое время чуть более чем вдвое.
$O(n^2)$	Квадратичный рост — удвоение размера задачи в 4 раза увеличивает необходимое время.
$O(n^3)$	Кубическое рост — удвоение размера задачи увеличивает необходимое время в восемь раз.
$O(c^n)$	Экспоненциальный рост — увеличение размера задачи на 1 приводит к $c-$ кратному увеличению необходимого времени; удвоение размера задачи увеличивает необходимое время в квадрат

Рисунок 2 – пример оценок сложности

Асимптотические оценки имеют свои особенности:

- для малых *n* пренебрежение константами даст неверные оценки сложности алгоритма, следовательно, можно получить неожиданные результаты после реализации алгоритма. Если в условии задачи есть ограничения на входные данные, оценки нужно проводить точнее;
- если вы пишите «одноразовую» программу, то вряд ли стоит тратить на разработку сильно больше времени, чем будет работать ваш самый неэффективный алгоритм.

Пример:

Рассмотрим асимптотические оценки для алгоритма инкремента в бинарном счетчике.

Задача: дан бинарный счетчик над числом размера *n* бит, у которого определен метод увеличения значения числа на 1. Оценить алгоритм инкремента.

Элементарными операциями будем считать операции сравнения элемента, декремента переменной *idx*, а также выставление бита в новое значение (хотя часто считают, что присваивания выполняются мгновенно, но

никто нам не мешает считать и их: так как вычисления идут с точностью до константы, это не сыграет роли).

Код:

```
// совершенно надуманный класс, хранящий
// в массиве биты числа
// (последний элемент - нулевой бит)
class BinaryCounter {
public:
	BinaryCounter(const size_t size)
		: _size(size)
	{
		_bits = new bool[size];
		memset(_bits, 0, size);
	}

	// добавление единицы
	void inc() {
		// наш _size - это n
		// В лучшем случае - весь массив заполнен нулями
		// В среднем случае - в начале есть биты-единички
		// В худшем случае - все биты единички

		size_t idx = _size - 1;
		while (_bits[idx] == true) {
			_bits[idx] = false;
			if (idx == 0) {
				// Overflow!!11
				return;
			}
			idx--;
		}
		_bits[idx] = true;
	}

	unsigned int decimal() {
		// ...
	}
private:
	bool* _bits;
	size_t _size;
};
```

Возможные случаи:

- текущее состояние массива – все биты 0;
- текущее состояние массива – некоторые биты в начале 1;
- текущее состояние массива – все биты равны 1.

Рассматривая лучший случай, можно заметить, что нам не нужно заходить в цикл вообще, тогда для лучшего случая количество операций равно двум, т.е. нам гарантирована сложность $O(1) = \Theta(1)$.

В среднем случае первые k битов являются единицами, тогда количество операций будет равно $4k + 1$. Для малых k сложность будет константной, но при стремлении k к чему-то очень большому (к n), слагаемое «1» становится несущественным, откуда получаем гарантированную сложность $O(4n + 1) = O(4n) = O(n)$.

В худшем случае получаем аналогичную ситуацию, с разницей, что k изначально равно n, т.е. гарантированная сложность – $O(n)$.

Что будет, если вызвать метод inc несколько раз? Допустим, метод вызывается m раз. Тогда можно (скинуть мне скрин с открытой лекцией, чтобы обрадовать Татьяну Викторовну) дать верхнюю оценку $mO(n) = O(mn)$, а если устремить m к n, то оценкой будет $O(n^2)$. Стоит заметить, что это очень грубая оценка, так как ситуации с последовательным следованием большого количества единиц встречаются не так часто.

3. Амортизированные оценки

Не все алгоритмы имеют одинаковую сложность от запуска к запуску. Для таких алгоритмов, которые должны запускаться несколько раз, ищут амортизационную сложность (ее оценку).

Амортизационная оценка является некоторым усреднением асимптотических оценок для серии запусков алгоритма. Ожидается, что «быстрые» запуски скомпенсируют «медленные», если их будет много больше.

Алгоритмы с амортизационной оценкой лучшей, чем их верхняя асимптотическая оценка нельзя использовать в задачах, где требуется постоянно-эффективное выполнение задачи!

Классическими методами подсчета амортизированной сложности являются:

- банковский метод;
- метод потенциалов.

(Эти методы могут быть рассмотрены после выхода на очную форму обучения или я попробую объяснить их лично, если обратитесь с вопросом).

В некоторых случаях амортизационная сложность находится интуитивно или с помощью каких-то простых математических вычислений.

Вполне очевидно, что для метода inc() в бинарном счетчике подавляющее количество запусков будет иметь сложность $O(1)$, только некоторые из запусков будут деградировать до линейной сложности (причем гарантируется, что после «медленного» запуска будет следовать серия «быстрых»), из чего можно сделать вывод, что амортизированная оценка данного метода равна $Amort(O(1))$.

4. Базовые структуры данных

Базовые (простейшие) структуры данных – на их основе (или на основе их комбинации) строятся более сложные структуры данных.

К базовым относятся массивы и рекурсивные типы данных (в простейшем случае – связные списки).

4.1. Массивы

Массив — базовая структура данных — набор однотипных элементов, расположенных в памяти непосредственно друг за другом. В длинном представлении не нуждается.

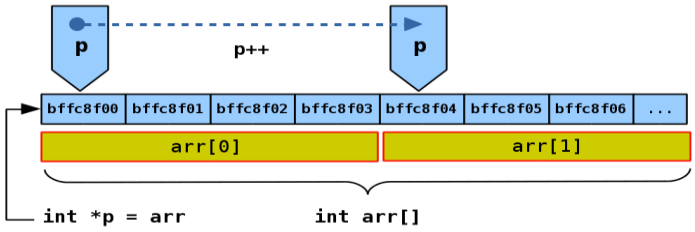

Рисунок 3 – Массив

Является самой простой и часто используемой структурой данных.

Особый интерес представляет расширение динамического массива — **вектор**.

Основные операции над вектором:

- доступ к элементу,
- вставка элемента,
- удаление элемента,
- поиск.

Вектор содержит в себе динамический массив размера capacity. Текущее количество элементов в векторе описывается полем size. Когда size становится равным capacity, происходит создание нового массива с большей емкостью, элементы старого массива копируются в новый, затем старый массив удаляется. Это позволяет поддерживать динамичность с сохранением свойств массива. Примерно тот же принцип с удалением элементов (если нужно сохранить какой-то баланс с памятью).

Существует две стратегии изменения capacity:

- аддитивная: capacity = oldCapacity + delta;
- мультипликативная: capacity = coef * oldCapacity.

На практике чаще всего используется мультипликативная стратегия.

Пример:

Пусть $loadFactor = \frac{size}{capacity}$, $coef = 2$.

$$\begin{cases} capacity := 2 * capacity, if\ loadFactor > 1 \text{ (хотим добавить, но некуда)}, \\ capacity := 0.5 * capacity, if\ loadFactor == \frac{1}{4} \text{ (при удалении)}, \\ capacity := 1, if\ size == 0 \text{ (например, при создании вектора)}. \end{cases}$$

Оценки операций:

	Лучший случай	Средний случай	Худший случай
Доступ	O(1)	O(1)	O(1)
Вставка	Amort(O(1)) [вставка в конец]	O(n) [вставка не в конец]	O(n) [вставка в начало]
Удаление	Amort(O(1)) [удаление с конца]	O(n) [удаление не из конца]	O(n) [удаление первого элемента]
Поиск	O(1) [искомый – первый]	O(n) [искомый – не первый]	O(n) [искомый – последний]

Прочие достоинства: минимальный overhead, простота.

Используется в задачах, где нужно гарантировать быстрый доступ к элементам.

4.2. Рекурсивные структуры

Структуры, которые содержат в себе указатель на объект такой же структуры, являются рекурсивными.

Пример: связные списки.

Связный список — структура данных, в которой каждый элемент содержит указатель на следующий элемент (односвязный список) или и на следующий, и на предыдущий (двусвязный список).

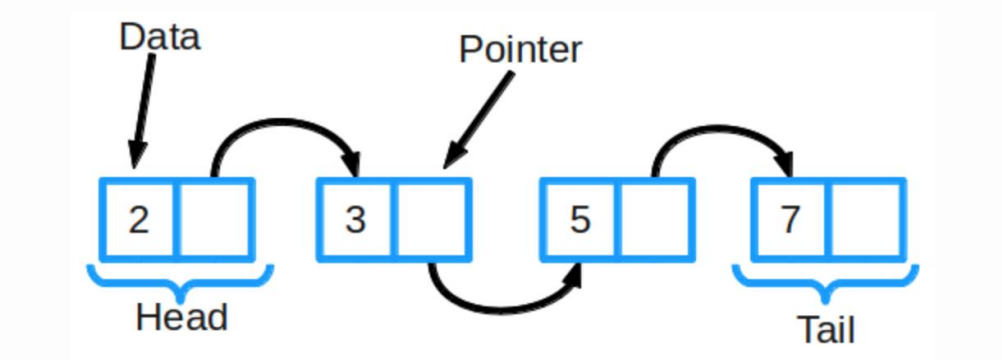

Рисунок 4 – Односвязный список

Основные операции над списками:

- доступ к элементу,
- вставка элемента,
- удаление элемента,
- поиск.

Оценки операций:

	Лучший случай	Средний случай	Худший случай
Доступ	O(1)	O(n)	O(n)
Вставка (если знаем, куда вставлять)	O(1)	O(1)	O(1)
Удаление (если знаем, откуда удалять)	O(1)	O(1)	O(1)
Поиск	O(1) [искомый – первый]	O(n) [искомый – не первый]	O(n) [искомый – последний]

Прочие недостатки: overhead на хранение указателей – на x64 двусвязный список из 10000 элементов навезет лишние 10000 * 2 * 8 = 156.25 КБ.

Используется в задачах, где нужно гарантировать одинаково быстрое в любой момент времени добавление или удаление узлов.

ВЫВОДЫ

1. нужно уметь понимать О-символику, так как это используемый в сообществе программистов язык для описания сложности алгоритмов, никуда вы от нее не денетесь;

2. выбор структуры данных играет важную роль для решения поставленной задачи.