Лекция 8.12.25

Полиномиальная и экспоненциальные сложности. Классы P и NP. Полиномиальная сводимость. NP-полные задачи. Проблема тысячилетия. Примеры

Асимптотическая сложность

Как оценить время работы алгоритма? На одном компьютере алгоритм работает 1 секунду, на другом на тех же входных данных 10. Один и тот же алгоритм, скомпилированный разными компиляторами, может работать разное количество времени.

```
(base) ➜  day4 g++ temp.cpp -o temp -O0
(base) ➜  day4 time ./temp
-2147059645./temp  0.82s user 0.02s system 65% cpu 1.286 total
(base) ➜  day4 g++ temp.cpp -o temp -O3
(base) ➜  day4 time ./temp
-2147074960./temp  0.65s user 0.01s system 78% cpu 0.840 total
```

С минимальной оптимизацией компилятора код сортировка массива из 10^7 случайных элементов работает 0,82с, с максимальной оптимизацией - 0,65с. Причина - разный итоговый набор бинарных инструкций для процессора.

Проблема

Невозможно оценить точное время работа алгоритма. Нужно придумать способ оценивать алгоритмы для определения какой алгоритм лучше или хуже.

Решение

Оценивать функциональную зависимость количества операций от размера входных данных или других параметров.

Асимптотическая нотация (сложность алгоритма)

Определение. Пусть $f(n)$ — это какая-то функция. Говорят, что функция $g(n) = O(f(n))$, если существует такие константы c и n_0, что $g(n) < c \cdot f(n)$ для всех $n \geq n_0$.

1. Big O — верхняя оценка в худшем случае
2. Ω — нижняя
3. Θ — и то, и другое одновременно (средний случай)
   На практике 90 % времени нам хватает просто говорить «O(n²)» и все понимают.

- Помимо времени, аналогично оценивают потребляемую память.

Пример

Название	Лучшее время	Среднее	Худшее	Память	Обмены (в среднем)
Сортировка пузырьком (Bubble Sort)	$O(n)$	$O(n2)$	$O(n2)$	$O(1)$	$O(n^2)$
Сортировка вставками (Insertion Sort)	$O(n)$	$O(n2)$	$O(n2)$	$O(1)$	$O(n2)$
Сортировка Шелла (Shellsort)	$O(nlog^2n)$	Зависит от выбора шага	$O(n2)$	$O(1)$	$O(n2)$
Сортировка выбором (Selection Sort)	$O(n2)$	$O(n2)$	$O(n2)$	$O(1)$	$O(n)$
Быстрая сортировка (Quick Sort)	$O(nlogn)$	$O(nlogn)$	$O(n2)$	$O(logn)$ (стек вызовов)	$O(nlogn)$
Сортировка слиянием (Merge Sort)	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$ $O(1)$	$O(nlogn)$
Timsort	$O(n)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	$O(nlogn)$

Правила подсчёта асимптотической сложности

0. Раскрываем функций (считаем их количество операций)
1. Если циклы на одном уровне - складываем
2. Если циклы на разных уровнях - перемножаем

$O(n)$

```
for(int i = 0; i < n; i++) // n итераций
    a += i;

for(int j = 0; j < n; j++) // n итераций
    a += j;
```

$O(n^2)$

```
for (int i = 0; i < n; i++) // n итераций
    for (int j = 0; j < n; j++) // n итераций
        a += j; // n * n выполнений
```

Мастер-теорема

Пусть имеется рекуррентное соотношения:

$$T(n) = \begin{cases} a\,T\left(\frac{n}{b}\right) + O(n^c), & n > 1 \\ O(1), & n = 1 \end{cases},$$

где $a \in \mathbb{N}, b \in \mathbb{R}, b > 1, c \in \mathbb{R}^+$.

Тогда асимптотическое решение имеет вид:

1. Если $c > \log_b a$, то $T(n) = O(n^c)$
2. Если $c = \log_b a$, то $T(n) = O(n^c \log n)$
3. Если $c < \log_b a$, то $T(n) = O\left(n^{\log_b a}\right)$

Пример:

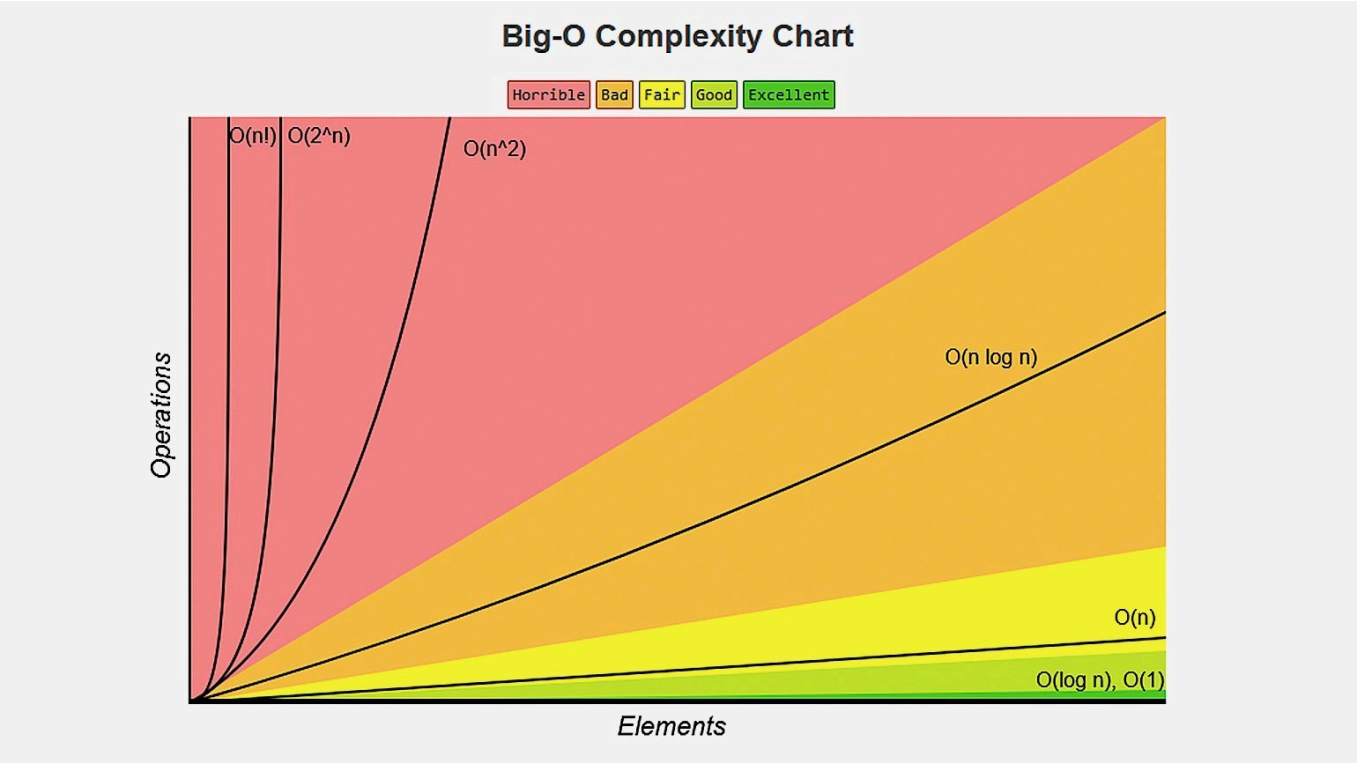
можно использовать для доказательства того, что сортировка слиянием работает за $O(n\ log\ n)$

a = 2, b = 2, c = 1

$T(n) = 2 * T(n/2) + O(n), n > 1$

$T(n) = O(1), n = 1$

График различных асимптотик



Чем меньше операций - тем быстрее работает алгоритм, тем быстрее "прогружаются страницы у пользователя", обратаываются запросы в базах данных и т.д.

Полиномиальная и экспоненциальная сложность

$O(1)$, $O(logn)$, $O(n)$, $O(nlogn)$, $O(n^2)$, $O(n^3)$ — полиномиальные
$O(2^n)$, $O(n!)$, $O(n^n)$ — экспоненциальные и хуже

Примерное время работы разных асимптотик

n	n^3	2^n	$n!$
10	0.001 с	0.001 с	0.003 с
50	0.125 с	35 лет	10^{54} лет
100	1 с	10^{20} лет	невообразимо

Вывод

Если алгоритм экспоненциальный - он работает адекватное время только на очень маленьких n. Всё что мы можем решить в реальной жизни на больших данных - полиномиально

Полиномиальная сводимость

Если мы умеем решать задачу B за полиномиальное время и умеем сводить задачу A за полиномиальное время к B и обратно, то мы умеем решать A за полиномиальное время.
Работает транзитивность. A <-> B <-> C

Класс P и NP

P - множество задач, для которых существует детерминированный алгоритм, работающий за полиномиальное время
NP - множество задач, для которых, если вам дали кандидат на решение, вы можете проверить его правильность за полиномиальное время

Примеры

- P: сортировка, поиск в отсортированном массиве, кратчайший путь в графе (Дейкстра, Форд-Беллман)
- NP, но очевидно в P: всё из P тоже в NP. Мы можем проверить правильность ответа.
- NP, но неизвестно в P ли:
  - Задача коммивояжёра (TSP) - детерминированный алгоритм за $O(2^n n)$
  - Задача о выполнимости булевой формулы (SAT)
  - Задача о клике, о вершинном покрытии
  - Задача о раскраске графа. Дана раскраска, можем проверить корректность за полиномиальное время

Важно: P ⊆ NP (очевидно). Вопрос в том, строгое ли это включение.

NP полные задачи

Задача называется **NP-полной**, если выполняются два условия одновременно:

1. Она лежит в NP (то есть решение можно быстро проверяется).
2. Она «самая трудная» в NP: К ней можно свести полиномиально **любую** другую задачу из NP.

Если ты завтра найдёшь полиномиальный алгоритм хотя бы для одной NP-полной задачи → автоматически появятся полиномиальные алгоритмы для всех задач из NP (включая взлом RSA, идеальное планирование, расшифровку генома за секунды и т.д.).

Список NP полных задач (неполный):

- SAT, 3-SAT (булевые формулы)
- Коммивояжёр
- Клика
- Вершинное покрытие
- Раскраска графа
- Тысячи других (список Карпа и дальше)

Вероятностные алгоритмы

Часто для NP-полной задачи существует вероятностный алгоритм (недетерминированный) или приближённый, который работает за полиномиальное время, но не гарантирует идеальную точность решения.

Другие классы задач

Для всех ли задач существует полиномиальная проверка решения? Нет.

Класс задач	**Есть ли полиномиальная проверка решения?**	**Пример**	**Что будет, если дадут «кандидат» в ответ**
P	Да (очевидно)	Сортировка, кратчайший путь	Проверяем за полином
NP	Да по определению	SAT, TSP, 3-цветность графа	Проверяем за полином

Класс задач	Есть ли полиномиальная проверка решения?	Пример	Что будет, если дадут «кандидат» в ответ
co-NP	Да, но для противоположной задачи	Невыполнимость формулы (UNSAT)	Если формула невыполнима — проверка может быть очень тяжёлой
EXP-complete	Нет	Шахматы на доске n×n, Go на n×n	Даже проверить победу за полином нельзя
Undecidable (неразрешимые)	Нет, вообще нет алгоритма проверки	Проблема остановки (Halting problem)	Нет никакого алгоритма, который скажет «да/нет» правильно для всех входов

Самые яркие примеры, где полиномиальной проверки *НЕ* существует

1. **Шахматы/Го/Шашки на доске n×n** Задача: «Даны позиция и размер доски n×n. Выигрывают ли белые при идеальной игре обеих сторон?» → Это EXP-полная задача. → Даже если тебе дали полную стратегию (дерево глубины до 10^n ходов), проверить, что это действительно выигрышная стратегия, ты сможешь только переиграв всю игру — это экспоненциальное время.
2. **Проблема остановки Тьюринга** «Остановится ли данная программа на данном входе?» → Нет никакого алгоритма, который всегда правильно отвечал бы «да» или «нет», не то что полиномиального.
3. **Постова соответствия соответствия (Post Correspondence Problem)** Классическая неразрешимая задача, лежит вне NP и co-NP.
4. **Некоторые задачи из теории автоматов и формальных языков** Например, «Эквивалентны ли два регулярных выражения с операцией пересечения?» — тоже вне NP/co-NP.

Почему тогда в АиСД мы почти всегда говорим именно про NP?

Потому что нас интересуют задачи, которые:

- встречаются в реальной жизни (оптимизация, планирование, криптография, биоинформатика),
- имеют короткий сертификат «да» (то есть быстро проверяются, если ответ «да»).

А всё, что выходит за рамки NP, либо неразрешимо вообще, либо настолько тяжело, что в практике почти не встречается.

Сравнение NP задач и NP-полных задач

Вопрос	**NP-задача**	**NP-полная задача**
В NP?	Да	Да
Проверка решения	Полиномиальная	Полиномиальная
«Сложность» внутри NP	Может быть лёгкой, средней или самой тяжёлой	Самая тяжёлая в NP
Если я найду полиномиальный алгоритм для неё…	Круто, но мир не перевернётся. Только эта задача (и те, что сводятся к ней) станет «лёгкой»	Мир перевернётся. Автоматически все задачи из NP станут полиномиальными (P = NP)
Аналогия	Обычный замок в огромном городе	Мастер-ключ от всех замков города
Примеры	- Проверка простого числа (до 2002 года была просто в NP) - Факторизация (разложение числа на множители) - Изоморфизм графов - Многие задачи на квантовых компьютерах	- SAT, 3-SAT - Коммивояжёр - Клика - Раскраска графа - Тысячи других (список Карпа и дальше)

Проблема тысячелетия. P = NP (проблема перебора)

Одна из центральных открытых проблем в теории алгоритмов.
Нестрого говоря, роблема равенства P = NP состоит в следующем
если положительный ответ на какой-то вопрос можно довольно
быстро *проверить* (за полиномиальное время), то правда ли, что ответ на этот вопрос можно довольно быстро *найти* (также за полиномиальное время и используя полиномиальную память)?
Другими словами, действительно ли проверить решение задачи столь же ресурсозатратно, как и отыскать решение?

Если вдруг завтра кто-то докажет P = NP и даст полиномиальный алгоритм хотя бы для SAT:

- Взлом всей современной криптографии (RSA, ECC и т.д.)
- Моментальное решение огромного количества оптимизационных задач
- Мир изменится кардинальнее, чем от появления интернета

Если докажут P ≠ NP:

- Криптография остаётся в безопасности
- Мы получаем теоретическое обоснование, почему некоторые задачи «трудные по своей природе»

В любом случае продолжаются поиски приближённых и параметризованных алгоритмов