Просто о сложности. Теория быстродействия. Конспект темы

Всегда ли нужно измерять

Во время оборачивания вектора размером 10 000 элементов программа совершит:

- при вставке в конец — 10 000 операций,
- при вставке в начало — 50 005 000 операций.

Разница впечатляющая, и она отражается на вас и вашем компьютере.

Вставка в начало вектора размера K потребует $K + 1$ операцию, так как нужно переместить K существующих элементов вектора и записать ещё одно число.

Вставка в конец требует только одну операцию записи.

От константы к квадрату

Первое правило асимптотики: убираем всё меньшее, оставляем главное.

Абстрактные понятия «количество операций», «количество сложений», «количество сравнений» не имеют большого смысла, так как разные операции имеют разную продолжительность. Важно, как программа поведёт себя при больших объёмах входных данных.

Асимптотика — скорость роста количества операций. Она обозначается буквой O.

Количество, выражаемое формулой $O(N)$, может при некотором N быть больше, чем другое количество, выражаемое формулой $O(N^2)$. Но скорость роста у $O(N)$ меньше. Поэтому при увеличении N неизбежно настанет момент, когда $O(N^2)$ станет больше.

Самый простой пример асимптотики — **константа**. Она обозначается как $O(1)$ и показывает, что оцениваемое количество известно заранее.

Оцениваем сложность программы

Асимптотика количества операций программы называется **асимптотическою сложностью**, или **сложностью программы**.

Узнать сложность стандартного алгоритма C++ или метода контейнера можно из документации на сайте cppreference.com.

Чтобы вычислить сложность:

- определитесь, что именно измерять: отдельную функцию, алгоритм, уже реализованный в C++, или всю программу целиком;
- подумайте, от каких данных и параметров зависит количество операций, и как их измерить.

Второе правило асимптотики: оценивать худший случай.

Причины выбирать худший случай:

1. Так проще оценивать.
2. Нежелательно, когда программа работает быстро, но при случайном стечении обстоятельств вдруг подвисает.
3. Программа с плохой сложностью может подвисать не случайно, а при специально подобранных входных данных. Этим могут воспользоваться злоумышленники.

Иногда оценивают **среднюю сложность** — усреднённое количество операций, которое делает программа.

Улучшаем сложность

$O(1)$ — это сложность

- получения или изменения элемента `vector`, `deque`, `string` по итератору;
- получения или изменения элемента `vector` и `string` по номеру;
- перемещения итератора `vector` или `deque` на произвольное число в любую сторону;

- перемещение итератора `map` или `set` на один элемент;
- вставки в `stack`, в конец `vector`, в начало или конец `deque`, в любое место `list` по итератору. Для вектора эта сложность амортизированная. Что это значит, вы узнаете в одном из следующих уроков;
- удаления в аналогичных случаях.

$O(N)$ — это сложность

- любых алгоритмов, которые один раз проходят от одного итератора до другого: `find`, `find_if`, `iota`, `count`, `count_if`, `transform`;
- копирования любого контейнера или строки: `vector`, `set`, `map`, `deque`, `stack`;
- деструктора любого контейнера;
- сравнение контейнеров, строк;
- вставки в начало или середину `vector`, середину `deque`;
- удаления в аналогичных случаях.

Для методов контейнера N — это количество элементов контейнера, для алгоритмов — расстояние между итераторами.

Все указанные сложности верны, если такие операции с элементами контейнера как удаление, копирование и сравнение константны, то есть имеют сложность $O(1)$.

Логарифмическая сложность

У логарифмической зависимости есть особенность: если увеличить аргумент N **в несколько раз**, количество действий увеличится **на несколько раз**.

Важный пример логарифмического алгоритма — **бинарный поиск**. Он реализован в алгоритмах `binary_search`, `lower_bound`, `upper_bound`.

Логарифмическая сложность часто встречается в методах `set` и `map`, потому что в них использованы деревья поиска, имеющие логарифмическую глубину.

Опасности экспоненты

Логарифм «наоборот» — это **экспонента**.

Вот как можно решить задачу о числах Фибоначчи:

```
#include <cstdint>
#include <iostream>
#include <string>

using namespace std;

// числа Фибоначчи довольно быстро растут, используем int64_t
int64_t F(int i) {
    // тут обработаем i == 0 и i == 1
    if (i <= 0) {
        return 0;
    }

    if (i == 1) {
        return 1;
    }

    // рекурсивно вызовем саму функцию F
    return F(i - 1) + F(i - 2);
}

int main() {
    int i;

    while (true) {
        cout << "Введите индекс: "s;
        if (!(cin >> i)) {
            break;
        }

        cout << "Fi = "s << F(i) << endl;
    }
}
```

Эти числа F_n задаются так: $F_0 = 0, F_1 = 1$, а остальные получаются по формуле $F_n = F_{n-1} + F_{n-2}$.

`F` вычисляется рекурсией. Но глубина рекурсии не превышает `i`. `i` будет в пределах 100, что вполне допустимо. На первый взгляд программа работает неплохо:

```
Введите индекс: 0
Fi = 0
Введите индекс: 1
Fi = 1
Введите индекс: 2
Fi = 1
Введите индекс: 3
```

```
Fi = 2
Введите индекс: 4
Fi = 3
Введите индекс: 5
Fi = 5
Введите индекс: 6
Fi = 8
Введите индекс: 10
Fi = 55
Введите индекс: 20
Fi = 6765
Введите индекс: 30
Fi = 832040
Введите индекс: 40
Fi = 102334155
```

Но перед вычислением 40-го числа была подозрительная пауза.

Чтобы оценить проблемы производительности, нужно измерить время, внеся изменения в `main`:

```
...

int main() {
    for (int i = 0; i < 100; ++i) {
        int64_t fi;
        {
            LOG_DURATION("Number "s + std::to_string(i));
            fi = F(i);
        }
        cout << fi << endl;
    }
}
```

Программа выдаст:

```
Number 0: 0 ms
0
Number 1: 0 ms
1
Number 2: 0 ms
1
...
Number 27: 3 ms
196418
Number 28: 3 ms
317811
Number 29: 7 ms
```

```
514229
Number 30: 12 ms
832040
Number 31: 15 ms
1346269
Number 32: 36 ms
2178309
Number 33: 96 ms
3524578
...
Number 43: 4004 ms
433494437
Number 44: 6674 ms
701408733
Number 45: 10793 ms
1134903170
Number 46: 17412 ms
1836311903
```

Проблемы начались примерно с 27-го числа и начали увеличиваться лавинообразно.

`F` вызывает две такие же функции и будет работать примерно столько, сколько обе они вместе взятые.

Количество операций возрастает минимум в два раза каждые два числа. Такая сложность называется **экспоненциальной** и обозначается $O(a^N)$.

Обычно программы с такой сложностью непригодны для использования, потому что требуют много времени даже при маленьких входных данных.

Чтобы улучшить алгоритм, достаточно помнить два предыдущих числа:

```
int64_t F2(int i) {
    if (i == 0) {
        return 0;
    }

    int64_t prev0 = 0, prev1 = 1;

    for (int t = 1; t < i; ++t) {
        int64_t next = prev0 + prev1;
        prev0 = prev1;
        prev1 = next;
    }

    return prev1;
}
```

Арифметика сложности: три правила вычисления

Правило поглощения

Если программа выполняет несколько действий, сложность всей программы будет равна сложности самого медленного действия. При суммировании нескольких сложностей побеждает самая большая.

Правило суммирования

Иногда сложность программы зависит от нескольких параметров. Например, в этой функции:

```
void SortPair(std::vector<int>& v1, std::vector<int>& v2) {
    sort(v1.begin(), v1.end());
    sort(v2.begin(), v2.end());
}
```

Пусть размер первого вектора равен N, а второго — M. Тогда функция выполняет две операции со сложностями $O(N \log N)$ и $O(M \log M)$, но применить правило поглощения тут не получится: неизвестно, какая из этих двух сложностей окажется больше. Приходится оставить сумму: $O(N \log N + M \log M)$.

Правило умножения

Допустим, сложность функции `F(int k)` равна $O(k)$. Нужно оценить сложность такого кода:

```
bool F(int k);

bool G(int n) {
    for (int i = 0; i < n; ++i) {
        if (F(i)) {
            return true;
        }
    }
    return false;
}
```

Понадобятся две вещи:

- сложность тела цикла;
- асимптотика количества итераций.

Сложность тела цикла оцените по худшему случаю, когда `i == n-1`. Самое сложное — это вызов `F(i)`, который в этом случае имеет сложность $O(n-1) = O(n)$.

Количество итераций тоже оцените по худшему случаю: когда все итерации функция `F` возвращала `false`. Тогда будет $O(n)$ итераций.

По правилу умножения надо перемножить оба результата: сложность тела и количество итераций. В примере оба равны $O(n)$, так что получится $O(n \cdot n) = O(n^2)$.

Амортизированная сложность

Не путайте **амортизированную сложность** со средней. Некоторые алгоритмы могут работать быстро или медленно в зависимости от входных данных. Например, сортировка. Обычно она работает за $O(N \log N)$, но если вектор отсортирован изначально, сортировка может определить это за $O(N)$ и завершиться. В лучшем случае её сложность будет $O(N)$.

Средней сложностью называют усреднение сложности по всем возможным входным данным. Это не гарантирует быструю работу в каждом случае.

Амортизированная сложность гарантирует быструю работу — вернее, быструю работу программы в итоге, после выполнения многих подобных операций.

Амортизированная сложность не гарантирует, что каждый вызов функции или метода будет быстрым. Но если таких вызовов сделать много, плохих случаев окажется мало, и в совокупности количество операций будет небольшим.

Не худший случай

Чтобы оценить, сколько примерно времени будет работать ваш алгоритм, ориентируйтесь на правило: ядро процессора выполняет около миллиарда действий в секунду.