Программирование в задачах радиолокации

КМБО-02-19

21 июня 2020 г.

1 Вычислительная сложность алгоритмов. Пространственная и временная сложность

Замечание 1 *Далее я буду безбожно копипастить лекцию Макса по соответствующей теме, которая лежит в группе.*

Определение 1 ***Классический алгоритм*** *- конечная последовательность операций, понятная исполнителю, строгое исполнение которой решает поставленную задачу.*

Замечание 2 *Также существуют специальные виды алгоритмов (например вероятностные), для которых не всегда определены перечисленные далее св-ва, но поставленную задачу они решают.*

1.1 Свойства алгоритмов

- **Определение 2** ***Дискретность*** *- алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов. При этом для выполнения каждого шага алгоритма требуется конечный отрезок времени, т.е. преобразование исходных данных в результат осуществляется во времени дискретно.*
- **Определение 3** ***Детерминированность*** *(определённость) - в каждый момент времени следующий шаг работы однозначно определяется состоянием системы. Таким образом, алгоритм выдаёт один и тот же результат для одних и тех же исходных данных.*
- **Определение 4** ***Понятность*** *- алгоритм должен включать только те команды, которые доступны исполнителю и входят в его систему команд.*
- **Определение 5** ***Конечность*** *- в более узком понимании алгоритма как математической функции, при правильно заданных начальных значениях алгоритм должен завершать работу и выдавать результат за определённое число шагов. Однако довольно часто определение алгоритма не включает завершаемость за конечное время.*
- **Определение 6** ***Универсальность*** *- алгоритм должен быть применим к разным наборам начальных данных.*
- **Определение 7** ***Результативность*** *- завершение алгоритма определёнными задачей результатами.*

1.2 Сложность алгоритма

Определение 8 ***Вычислительная сложность -*** *функция зависимости ресурсов, затрачиваемых некоторым алгоритмом, от размера и состояния входных данных.*

Основными ресурсами являются:

- Процессорное время (абстракция - время)
- Память (абстракция - пространство)

Замечание 3 *Когда идёт речь о сложности алгоритма, речь идёт не о конкретных величинах (секундах), а об абстрактных, таких, как количество элементарных операций.*

Определение 9 ***Принцип trade-off*** *- улучшение по сложности одного ресурса за счёт ухудшения по другому.*

При анализе сложности алгоритма можно рассматривать сложность:

- в лучшем случае
- в среднем случае
- в худшем случае

Определение 10 ***Гарантированной сложностью*** *называется сложность алгоритма в худшем случае.*

Определение 11 ***Временная сложность*** *- функция от размера входных данных n, равная максимальному или среднему или минимальному количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи.*

Определение 12 ***Пространственная сложность*** *- аналогично определению выше, но про объём памяти.*

2 Асимптотические и амортизационные оценки

Замечание 4 *Здесь мы попытаемся дать чуть более 'алгоритмический' материал, чем тот, что приведён в лекции.*

Определение 13 ***Асимптотическая оценка*** *- функция от времени, либо ограничивающая снизу* ($\Omega(g(n))$) *или сверху* ($O(g(n))$) *временную сложность алгоритма, либо совпадающая с ней*($\theta(g(n))$).

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.

Наиболее распространённые значения O		
Имя	Нотация	Пример, когда чаще возникает
Константная	$O(1)$	Когда нет необходимости в трудоёмких операциях, таких как перенос данных, балансировка и проч. *Пример:* вставка в связный список.
Логарифмическая	$O(logn)$	Когда удаётся на каждом шаге алгоритма отсекать $\frac{n}{k}$ неподходящих вариантов. *Пример:* бинарный поиск - худший случай(на каждом шаге отсекаем по половине от оставшегося массива), поиск в сбалансированном бинарном дереве (при переходе через один узел точно знаем, что одна из веток нам не подходит, т.е. каждый раз бракуем половинку от оставшихся данных).
Линейная	$O(n)$	Когда необходимо проверить каждый элемент коллекции. Это либо цикл на отрезке $[0; n)$, либо эквивалентная ему рекурсия. *Пример:* линейный обход неотсортированного массива для поиска минимума/-максимума.
Линейно-логарифмическая	$O(nlogn)$	Возникает в ситуациях с вложенными циклами, в одном из которых можно сделать оптимизацию и постоянно отсевивать по половинке данных. *Пример:* быстрая сортировка.
Полиноминальная	$O(n^2)$	Ситуация вложенных циклов, каждый из которых честно отрабатывает на каждой своей итерации все элементы коллекции. *Пример:* сортировка пузырьком.
Экспотенциальная	$O(2^n)$	*Пример:* алгоритмы перебора - *brute force.*
Факториальная	$O(n!)$	*Пример:* неэффективные комбинаторные алгоритмы по поиску перестановок, размещений, сочетаний и проч.

2.1 Построение асимптотической оценки

1. Подсчитать максимальное количество элементарных операций, которые совершает алгоритм, в терминах n. Построить функцию $f(n)$ количества операций от времени.

2. Оценить элементарные функции, входящие в $f(n)$: отбрасываем константы, константные множители, элементарные функции, которые являются бесконечно малыми относительно других элементарных функций при $n \to \infty$.

3. Радуемся результату.

Примеры на нахождение асимптотических оценок, после которых всё станет ясно:

Рассмотрим следующий код:

```
std::vector<int> vec(n);
/*vec заполнили*/
int m = vec[0];

for(int i = 0; i < nl ++i){
    if(vec[i] >= m)
        m = vec[i];
}
```

Теперь предположим, что для процессора элементарными являются следующие операции:

- Присваивание переменной значения;
- Доступ к элементу вектора по индексу;
- Сравнение двух чисел;
- Инкрементация значения;
- Основные арифметические операции.

Предположим также, что выбор между $if - else$ происходит мгновенно, время потребляет только проверка на выполняемость условия в if.

Теперь разберём код:

1. В первой строке мы проводим 2 операции: получаем значение элемента по индексу, присваиваем его переменной - эти операции не зависят от n.
2. Далее выражениям при заходе в цикл (инициализации $i = 0$, сравнению в выражении $i<n$) требуется по одной операции. Даже если $n = 0$, эти два выражения сожрут 2 операции.
3. После захода в цикл каждую итерацию будем так или иначе иметь ещё по две операции: инкремент $++i$ и сравнение $i<n$ - они будут давать $+2$ на каждой итерации, $\Rightarrow$ от них будет $2n$ операций.
4. Теперь получим некоторую функцию времени от входных данных $f(n) = 2n+4$, где $4 = 2$ операции в первой строке $+2$ операции для инициализации цикла.
5. В теле цикла мы имеем сравнение в конструкции if, которое будет выполняться каждую итерацию, там же мы будем получать $i -$ ый элемента вектора$\Rightarrow$ имеем $+2n$ операций в цикле.
6. Теперь встаёт вопрос о запуске тела условного оператора: в лучшем случае мы туда даже заходить не будем, в худшем - зайдём n раз и получим каждый раз по 2 операции: вызом по индексу и присваивание, итого имеем $2n$.
7. Теперь уточним $f(n)$: $f(n) = 4 + 2n + 2n + 0$ для лучшего случая и $f(n) = 4 + 2n + 2n + 2n$ для худшего случая.

Теперь давайте разбираться с функцией в худшем случае:

1. При анализе имеет значение только то, как себя ведут самые быстро растущие элементарные функции. Теперь смотрим на нашу: $f(n) = 6n+4$ - константы при росте n не растут $\Rightarrow$ избавляемся от них и получаем $f(n) = 6n$.
2. Теперь откинем множитель перед оставшейся элементарной функцией. Себе объясним такую небрежность тем, что мы говорим об абстракции, а в реальности такие множители могут либо раздуваться, либо наоборот исчезать - всё зависит от используемого ЯП. Теперь имеем $f(n) = n$ и, следовательно, $O(f(n)) = O(n)$.

Замечание 5 *Как видно, мы оценивали здесь* ***худший случай****. При оценке лучшего случая, как нетрудно убедиться, мы получим ту же самую оценку, следовательно, можно даже заключить, что в этой ситуации мы имеем оценку времени алгоритма* $\theta(n)$.

Потренируемся находить асимптотики для следующих функций $f(n)$ (они же отображения $f : n \to t$):

1. $f(n) = 5n + 12 \Rightarrow O(f(n)) = n$
2. $f(n) = 109 \Rightarrow O(f(n)) = 1$ - здесь $109 = 109 * 1$, множитель убираем, а 1 - элементарная функция наибольшего роста среди оставшихся.

3. $f(n) = n^3 + 1999n + 1337 \Rightarrow O(f(n)) = n^3$ - если подходить нестрого, скажем, что в любом случае n^3 растёт быстрее линейной функции при стремлении к ∞. Но мы же математики, поэтому подойдём формально: $\lim_{n\to\infty} \frac{n^3}{199n} = \infty$.

Если всё-таки осталось что-то непонятое, рекомендую ознакомиться с этими статьями:

`https://habr.com/ru/post/173821/`

`https://habr.com/ru/post/196560/`

2.2 Построение амортизационной оценки

Замечание 6 *Здесь мы расскажем об алгоритме* ***группировки****(ещё называют алгоритмом агрегации). Помимо него есть* ***банковский алгоритм*** *и его обобщение -* ***метод потенциалов****. Последние два счастья в рунете изложены сложновато или не до конца полно, а искать на английском в 12 часов ночи уже не хочется.*

Алгоритм группировки состоит в следующем:

1. Пусть t_i - реальное время, потраченное на конкретную операцию. Тогда всего времени на n операций было потрачено $\sum_{i=1}^{n}(t_i)$ времени.
2. Теперь поймём, что $\sum_{i=1}^{n}(t_i) \geq n$, т.к. $t_i \geq 1$.
3. Наконец, получим амортизационную оценку: $amort(f(n)) = \frac{\sum_{i=1}^{n}(t_i)}{n}$

Поясним на примере:

Задача 1 *Необходимо найти амортизационную оценку для операции push_back в вектор, который мы реализовывали в течение семестра.*

Решение 1 *Начнём с такого наблюдения: пусть у нас есть изначально пустой вектор и в нём выделена память на 4 элемента.*
Первая, вторая, третья, четвёртая операции push_back выполнятся за $O(1)$. С пятой же операцией мы получим $O(n)$ - она же оценка худшего случая, т.к. в векторе произойдёт перевыделение памяти и перенос старых данных в новый контейнер.
Теперь построим амортизационную оценку:

$$\frac{1+1+1+1+n}{n} = \frac{4+n}{n} = 1 + \frac{4}{n} \xrightarrow[n\to\infty]{} 1 \Rightarrow amort(O(f(n))) = amort(O(1))$$