Курс kiev-clrs – Лекция 1. Анализ алгоритмов

Иван Веселов

10 января 2009 г.

Содержание

1 Цель лекции

- Изучить средства описания и анализа алгоритмов
- Рассмотреть сортировку слиянием и сортировку вставкой
- Начать использовать асимптотическую нотацию для выражения времени выполнения алгоритмов
- Рассмотреть технику “разделяй и властвуй” в контексте сортировки слиянием

2 Введение

В рамках данного курса изучаются два аспекта:

- анализ алгоритмов
- дизайн алгоритмов

Мы начнём с анализа, т.к. перед тем, как что-то проектировать и создавать, нужно узнать критерии, определяющие насколько хорошо то, что у нас получается. Эти критерии и изучает анализ алгоритмов. Таким образом, анализ алгоритма – это теоретическое изучение производительности программ и использования ими ресурсов.

Но давайте будем реалистами и подумаем над тем, что есть вещи, которые могут быть важнее производительности в контексте разработки программ (спросить какие):

- корректность
- простота
- надёжность
- затраты на разработку (время, затраченное программистом)
- лёгкость в поддержке
- масштабируемость
- модульность
- безопасность

Может тогда изучение алгоритмов нам и не нужно? Нужно, так как:

- они помогают понять масштабируемость
- проводят границу между пригодностью и непригодностью кода, т.е. медленный алгоритм зачастую просто неприменимы для больших входных данных.
- они определяют язык, в терминах которого можно говорить о поведении программ
- скорость - это круто!

  Производительность - как деньги, которыми можно расплачиваться за всё остальное, если у вас есть производительность – можно обменять часть её на дополнительные "фичи", другую часть – на масштабируемость и т.д.
- развивают логическое мышление

3 Сортировка вставкой

Задача сортировки:

Вход: последовательность чисел $\langle a_1, a_2, \ldots, a_n \rangle$

Выход: перестановка $\langle a'_1, a'_2, \ldots, a'_n \rangle$ такая, что $a'_1 \leqslant a'_2 \leqslant ... \leqslant a'_n$ (монотонно возрастающая)

Для описания алгоритмов важно использовать чёткий и выразительный язык. Часто, хорошим вариантом является естественный язык – например русский или английский. Однако, когда необходимо более чёткое и точное понимание всего происходящего в программе мы будем использовать *псевдокод*.

Алгоритм сортировки вставкой:

```
INSERTION-SORT(A, n) //sorts A[1..n]
1  for j ← 2 to n
2       do key ← A[j]
3          i ← j − 1
4          while i > 0 and A[i] > key
5               do A[i + 1] ← A[i]
6                  i ← i − 1
7          A[i + 1] ← key
```

То, что часть начало массива является постоянно отсортированным – является *инвариантом цикла*, то есть таким свойством, которое не меняется при выполнении цикла. С помощью инвариантов доказывается

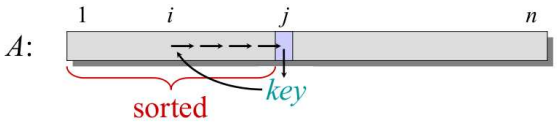

Рис. 1: Принцип сортировки вставкой

корректность алгоритма. Это напоминает метод математической индукции: сначала доказывается, что инвариант выполняется до начала цикла (базис индукции), затем, предположив, что свойство выполняется при счётчике цикла равном $N-1$ (гипотеза индукции) доказывается, что он останется верным и при N. Это можно сравнить с подъёмом по ступенькам: если мы знаем, что можем подняться на первую ступеньку и знаем, что можем переступить на следующую – то это означает, что можно подняться до самого конца лестницы. И наконец последний шаг доказательства корректности алгоритма – это убедиться, что при завершении главного цикла мы получаем решение поставленной задачи (этого шага нет в математической индукции).

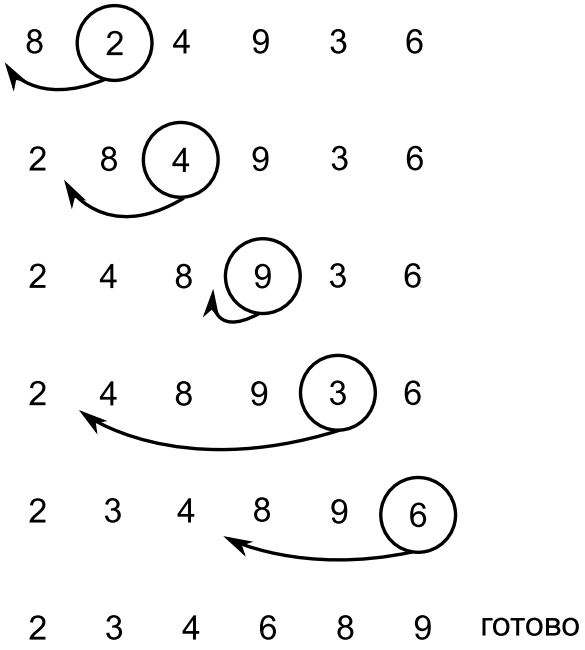

Рис. 2: Пример сортировки вставкой

Время работы алгоритма:

- зависит от входных данных (например они могут быть уже отсортированными)
- зависит от размера входных данных

Таким образом, можно параметризовать время работы по размеру входных данных алгоритма. При этом мы хотим получить верхнюю границу оценки (наихудшее время), чтобы обеспечить некие гарантии пользователю алгоритма.

4 Виды анализа

Можно рассматривать разные случаи:

- анализ наихудшего случая (worst-case analysis) – обычно мы будем проводить именно его.

  $T(n)$ = максимальному времени для любых входных данных размера n

- анализ среднего случая (average case) – иногда будем проводить и его.

  $T(n)$ = ожидаемому в среднем времени для всех вариантов входных данных размера n.

  Здесь нам нужны знания или предположения о том, что именно представляют собой данные, то есть как они статистически распределены. Часто можно предположить что все входные данные – равновероятны.

- анализ наилучшего случая (best case) – от него мало толку, т.к. можно придумать алгоритм, который будет хорошо работать для каких-то конкретных данных и очень плохо для всех остальных, проанализировать лучший случай и радоваться какой классный у нас алгоритм.

Вернёмся к сортировке вставкой. Какое время работы в наихудшем случае для этого алгоритма?

Это зависит от конкретного компьютера:

- относительная скорость – сравниваем на одной и той же машине
- абсолютная скорость – сравниваем на разных машинах

Это всё не очень удобно и понятно, потому здесь нас озаряет *великая идея*!

- игнорировать машинно-зависимые константы
- рассматривать не само время, а его рост, то есть поведение функции $T(n)$, когда $n \to \infty$

5 Θ-нотация

Асимптотическая оценка функции.

Математически определяется как:

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 > 0 \text{ и } n_0 \text{ такие, что } 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n), \forall n > n_0\}$$

Разбирать это мы пока не будем, и в данный момент мы будем определять оценку функции с помощью таких действий:

- отбросить от функции члены младшего порядка
- игнорировать множители-константы

Примеры:

$$5n^3 + 2n^2 - 10n + 657 = \Theta(n^3)$$
$$-7n^2 + 2n + 2 = \Theta(n^2)$$

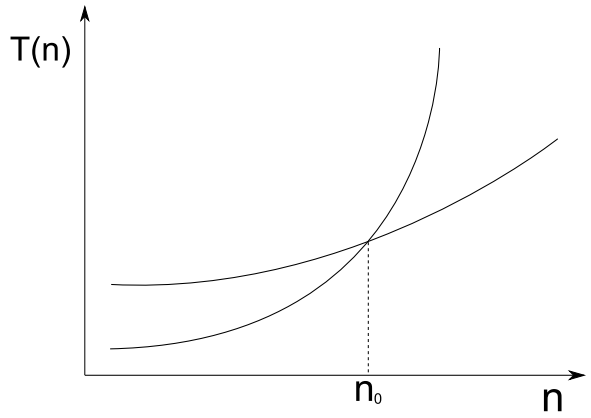

Рис. 3: Сравнение двух полиномиальных оценок

Поскольку нас интересует поведение функций, когда $n \to \infty$, то становится понятно, что алгоритм $\Theta(n^2)$ рано или поздно "победит" алгоритм $\Theta(n^3)$

6 Анализ сортировки вставкой

Худший случай входных данных для этого алгоритма – это массив, отсортированный в обратном порядке (при этом получается больше всего перестановок).

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2) \text{ (арифметическая прогрессия)}$$

Насколько быстр алгоритм сортировки вставкой?

- для маленьких N – его скорость вполне приемлема
- для больших N – нет (квадратичная скорость сортировки – это медленно)

7 Сортировка слиянием

Существуют несколько разных подходов к созданию алгоритмов, например тот, что был рассмотрен в сортировке вставкой – называется **инкрементным**: каждый раз у нас был уже отсортированный массив меньшего размера и мы постепенно добавляли к нему по одному новому элементу в нужное место.

Теперь мы кратко рассмотрим другой подход, который называется методом **декомпозиции** или **разделения** ("разделяй и властвуй"). Многие алгоритмы имею рекурсивную структуру, то есть вызывают себя один или несколько раз для выполнения некоторой полезной подзадачи. Обычно, такие алгоритмы как раз и разрабатываются методом декомпозиции.

Суть метода декомпозиции – три этапа:

- *Разделение* – разбиение задачи на подзадачи меньшего размера
- *Покорение* – решение этих задач рекурсивным образом
- *Объединение* – получение решения основной задачи из решений меньших

Алгоритмом именно такого типа является сортировка слиянием (merge sort). Схема работы процедуры Merge-sort($A[1\,..\,n]$) (оставить место слева для Θ-оценок):

1. Если $n = 1$ – готово, выдать A
2. Рекурсивно отсортировать массивы $A[1\,..\,\lceil n/2 \rceil]$ и $A[\lfloor n/2 \rfloor + 1\,..\,n]$
3. Объединить их (merge)

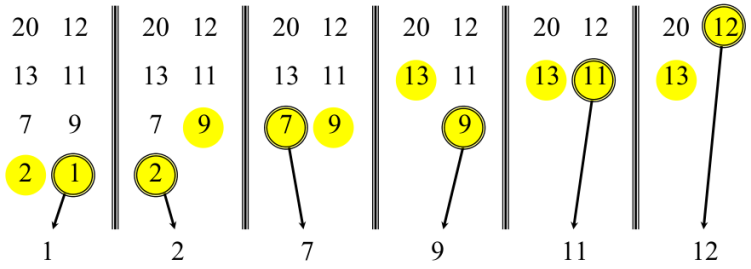

Рис. 4: Процедура слияния

Таким образом основная процедура, в которой выполняется работа – объединение. Рассмотрим как можно объединить два уже отсортированных массива:

(Можно описать как пример с двумя стопками отсортированных карт).

Поскольку нужно сделать как максимум n сравнений и каждое сравнение можно считать атомарной операцией ($\Theta(1)$) – получаем $\Theta(n)$.

Теперь запишем все шаги алгоритма и их оценки:

$\Theta(1)$	1. Если $n = 1$ – готово
$2T(n/2)$	2. Отсортировать $A[1\,..\,\lceil n/2 \rceil]$ и $A[\lfloor n/2 \rfloor + 1\,..\,n]$
$\Theta(n)$	3. Объединить их (merge)

Во втором пункте допущена некоторая неточность – мы приблизили всё к $n/2$, однако можно показать, что апроксиматически это ни на что не влияет.

Таким образом, получаем рекуррентное соотношение:

$$T(n) = \begin{cases} \Theta(1), & \text{если } n = 1 \\ 2T(n/2) + \Theta(n), & \text{если } n > 1 \end{cases} \tag{1}$$

Обычно мы можем пренебречь случаем, когда $T(n) = \Theta(1)$ для достаточно малых n, т.к. он не играет роли при асимптотической оценке, но следует быть внимательными.

В следующей лекции мы изучим несколько методов для получения решений подобных рекуррентностей, а сейчас мы только кратко продемонстрируем один из методов.

8 Дерево рекурсии

Преобразовываем нашу задачу к виду: $T(n) = 2T(n/2) + cn$, где константа $c > 0$. И начинаем выписывать сумму в виде дерева:

Рис. 5: Дерево рекурсии – корневой узел

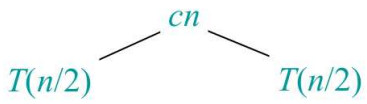

Рис. 6: Дерево рекурсии – шаг второй

Разворачиваем согласно соотношению 1 (см. рис. 6, 7)

Самые нижние узлы дерева (листья) – это подзадачи размером 1, то есть уже массивы длины 1, которые естественно отсортированы, потому никакой работы по их сортировке проводить не надо, потому в листьях пишется $\Theta(1)$ (см. рис. 8)

Поскольку каждые раз происходит уменьшение задачи вдвое, то определение высоты дерева по сути является эквивалентным вопросу – "сколько раз нужно поделить число на два, чтобы получить 1", и ответ на него – lgn, то есть двоичный логарифм (мы будем использовать обозначения из книги, несмотря на то, что в русскоязычной литературе принято обозначать так десятичный логарифм). Рис. 9

Теперь начнём суммировать значения в узлах дерева по уровням. (см. рис. 10, 11, 12)

Наконец просуммируем уровень листьев – самых нижних узлов дерева, т.к. их n, умножая на $\Theta(1)$ получим $\Theta(n)$ (см. рис. 13)

Теперь просуммируем значения сумм всех уровней, чтобы определить оценку для всего алгоритма, в итоге получаем $\Theta(nlgn)$ (см. рис. 14)

Выводы: поскольку $\Theta(nlgn)$ растёт намного медленнее, чем $\Theta(n^2)$, то сортировка слиянием использует намного меньше операций и потому работает быстрее. Это становится заметно на практике уже при $n > 30$

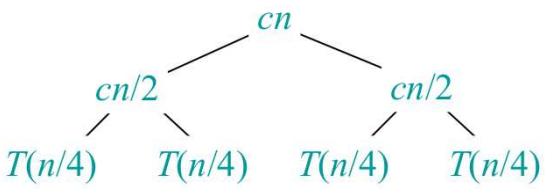

Рис. 7: Дерево рекурсии – шаг третий

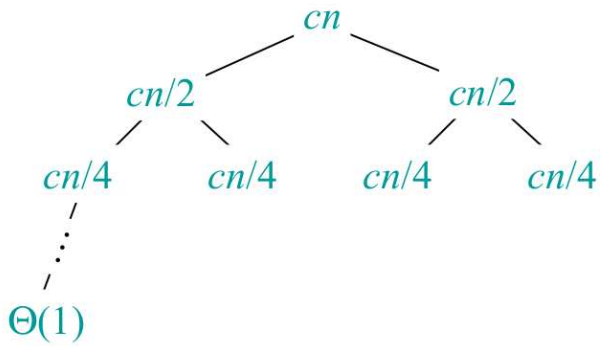

Рис. 8: Дерево рекурсии с листьями $\Theta(1)$

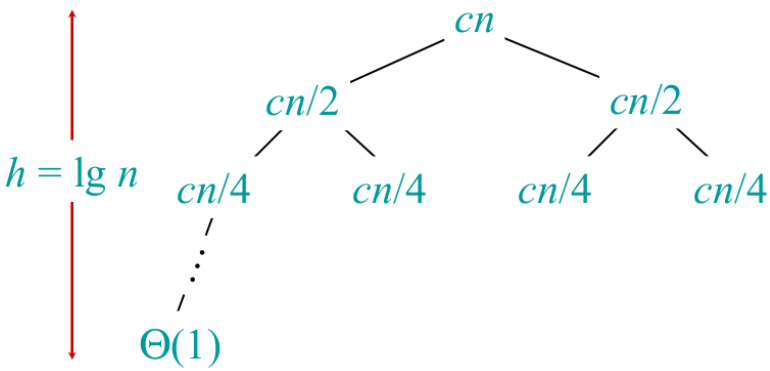

Рис. 9: Дерево рекурсии с указанной высотой

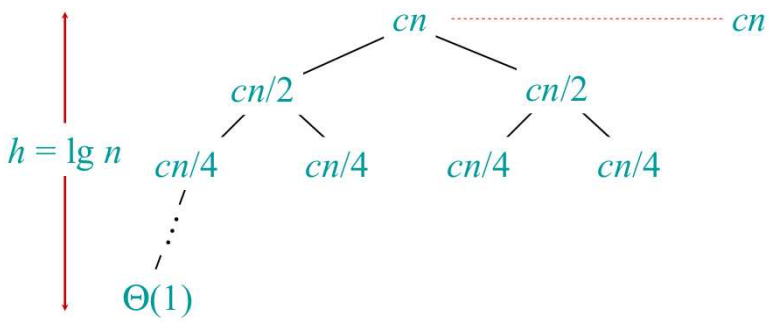

Рис. 10: Дерево рекурсии с суммой уровня 1

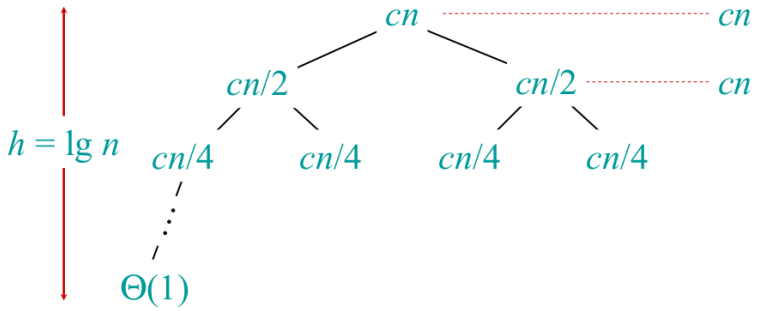

Рис. 11: Дерево рекурсии с суммой уровней 1-2

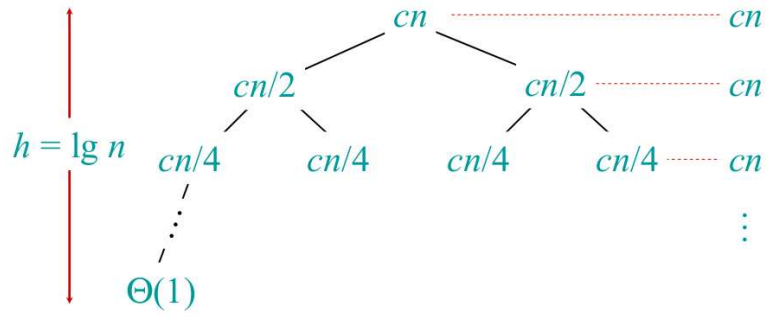

Рис. 12: Дерево рекурсии с суммой уровней 1-3

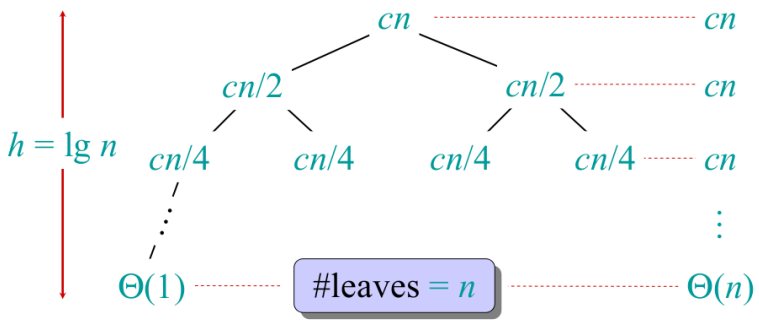

Рис. 13: Дерево рекурсии с суммой значений в листьях

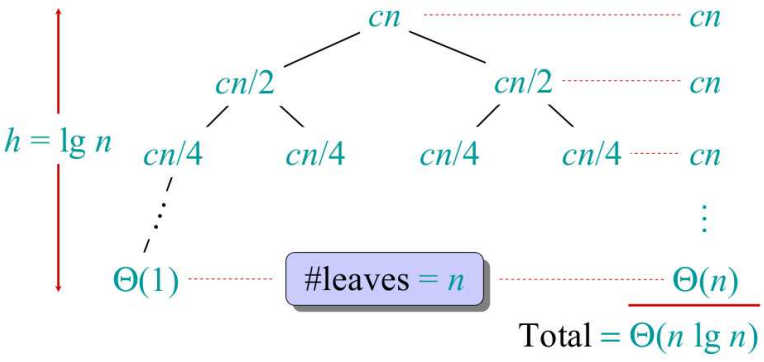

Рис. 14: Дерево рекурсии