**Алгоритмы и структуры данных.**

**Лекция 1.**

**Алгоритм** – корректно определенная вычислительная процедура, принимающая на вход и выдающая на выход данные определенного вида.

**Вычислительная процедура** – последовательность формальных манипуляций с данными.

*Корректный алгоритм для любых допустимых входных данных выполняется конечное время и выдает корректные выходные данные.*

*Корректные выходные данные должны удовлетворять требованиям, поставленным в решаемой задаче.*

**Структура данных** – способ организации данных, корректно поддерживающий определенный набор операций.

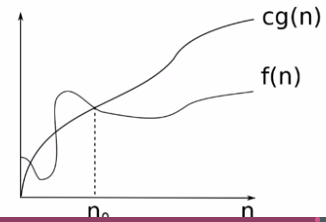
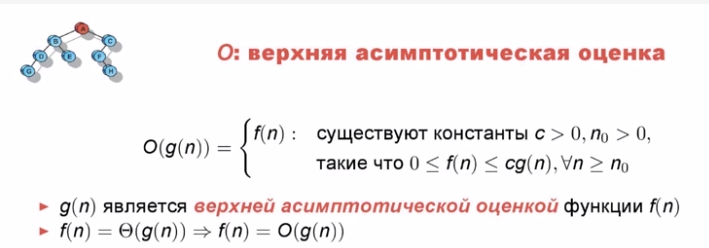
Зачем нужны эффективные алгоритмы? – Ресурсы компьютера ограничены (время работы, память)   
- Маршрутизация пакетов (поиск кратчайших путей)   
- Фильтрация спама (фильтры домена)   
- Базы данных, строковое алгоритмы  
- Сборка генома и тд.

Сложность алгоритмов, пример:

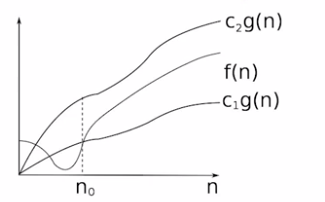
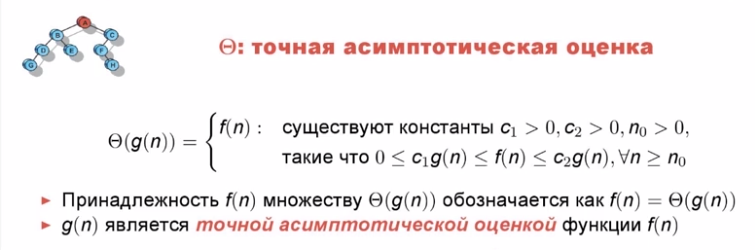
|  |  |  |
| --- | --- | --- |
| Сортировка | Вставками | Слиянием |
| Сложность | ~c1 \* n2 | ~c2 \* n\*log2n |
| От чего зависят c1, c2 – От реализации. На каком языке. | Ассемблер, c1 = 2 | JS, c2 = 100 |
| Вычислитель | Суперкомпьютер | Смартфон |
| Скорость | 1011 операций в секунду | 108 |
| Время | 2\*1016/1011 = 2\*105с | 100\*108\*27/108 = 2700с |

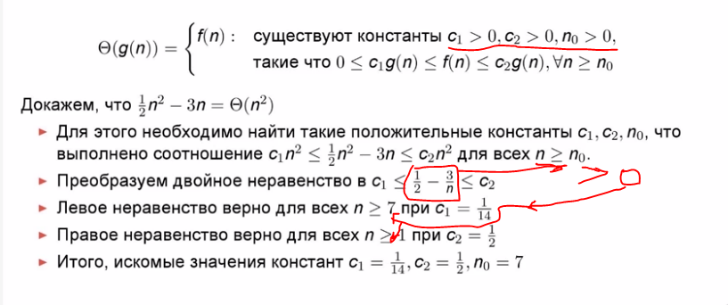
Оценка эффективности работы алгоритмов.

*Асимптотическая обозначения*:

О(g(n)) – Верхняя асимптотическая оценка  
  
т.е. существует такое значение n0, при котором функция f(n) меньше функции cg(n). (см. график)

Чаще всего используется для оценки времени работы алгоритмов. Свойства:   
Пусть f1(n) = О(g1(n)), f2(n) = О(g2(n)), тогда верно следующее:   
- f1(n) + f2(n) = О(max(g1(n), g2(n)))   
- f1(n) \* f2(n) = О(g1(n) \* g2(n))   
- c\*f1(n) = O(f1(n))   
- c + f1(n) = O(f1(n))

Ω(g(n)) - Нижняя асимптотическая оценка  
   
Есть некоторая функция f(n), которая, начиная с некоторого n0 больше функции cg(n). (см. график)   
  
θ(g(n)) - точная асимптотическая оценка.   
Выбирается максимальное g(n) для оценки функции. Т.е. функция с максимальной степенью или самая большая.  


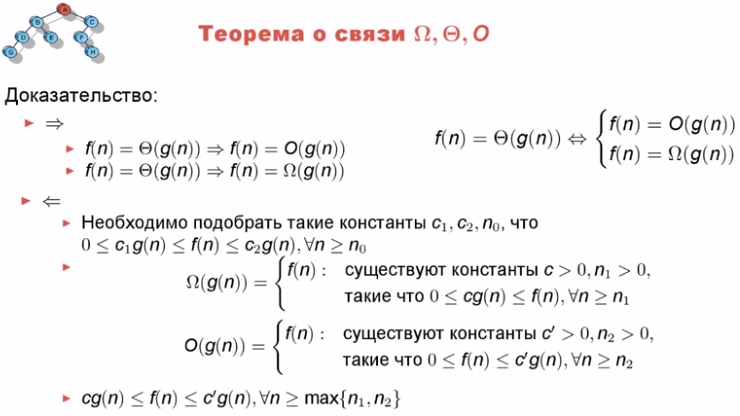


Время работы алгоритма: T(n), где n – целое число, размер входных данных.   
Входные данные можгут быть одним или несколькими числами.  
Пример Algoritm A(int n):   
1. j <- 1  
2. for i=1, to n do  
3. j <- j\*i

Инициализация переменной j выполняется за постоянное время с1  
Увелечение счетчика, проверка условия цикла, время на обновления переменной j в теле цикла обозначим, как с2  
Т.о T(n) = c1 + c2 \* n, т. к. 3 строка выполняется n раз

Асимтотическая оценка времени работы алгоритма: T(n) = θ(n)

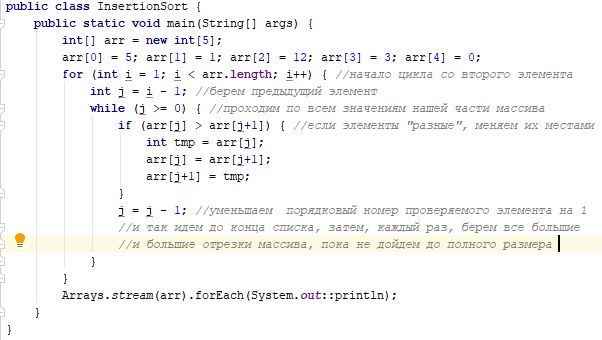
Теорема связи асимптотических оценок:

  
Т.е. если мы имеем θ(g(n)), то получается мы априори имеем верхнюю О(g(n)) и нижнюю Ω(g(n)) границы функции.

Алгоритм сортировки вставками.

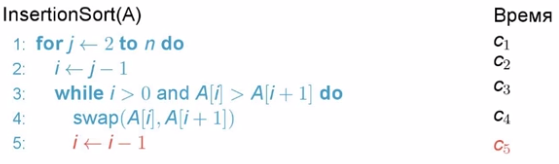
Задача сортировки чисел по возрастанию.   
На вход подается последоватьность чисел a1, …, anВыход: a’1, …, a’n , где a’1 <= a’2 <= a’3 …Пример:  
Вход – 10, 8, 2, 12, 5, 100  
Выход – 2, 5, 8, 10, 12, 100

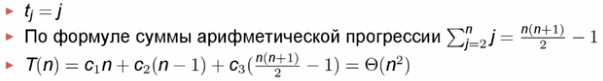
Можно сортировать все, что угодно, не только цифры.

Алгоритм сортировки вставками:  
Начинаем рассматривать числа со второго элемента, его индекс хранится в переменной j. В переменной i хранится индекс предыдущего элемента. (Процедура swap – это обмен двух переменных.) Проверяем условие A[i] > A[i+1] и если верно, то меняем местами элементы. Затем мы уменьшаем i, чтобы пройтись по предыдущим значениям и сравнить их. И так до момента i > 0. Потом возвращаемся к циклу for и у нас меняются j и i. Каждую итерацию мы сравниваем текущий элемент с предшествующим и если он меньше, то меняем их местами.  
<!-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* -->  
5 2 4 6 1 3  
InsertionSort(A)  
1. for j = 2 to n do  
2. i = j – 1   
3. while i > 0 and A[i] > A[i+1] do  
4. swap(A[i], A[i+1])  
5. i = i-1  
Пример на java:  


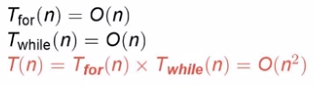
<!-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* -->

Оценка времени работы алгоритма.

Точная оценка:   
Время однократного выполнения каждой строки псевдокода – будем считать, что арифметические операции, занесения значения в память, проверка условий выполняются за фиксированное время c.  
Для однократного выполнения каждой строки необходимо:   
  
Но каждая строка выполняется не по одному разу, а несколько, в зависимости от некоторых условий. Число проверок условия цикла выполняется на одно больше, чем у нас значений, так как последняя проверка поймет, что мы вышли за границы и больше не пойдем в цикл (n раз). Тело цикла выполняется n – 1 раз. Количество выполнений цикла while обозначим за tj, т.о. он выполнится . Строки 4 и 5 псевдокода будет выполняться :

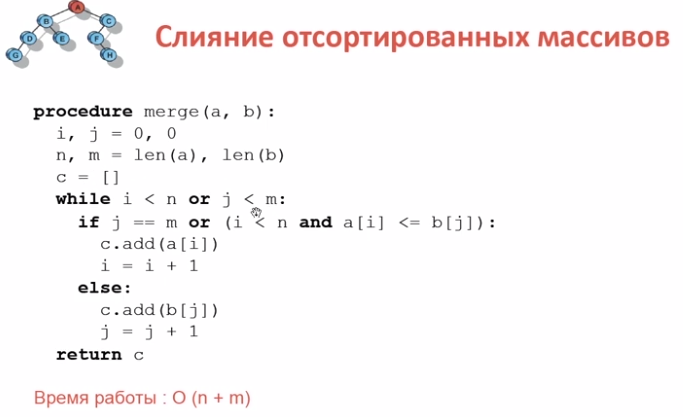
Для суммарного времени выполнения алгоритма достаточно умножить «сколько выполняется одна строка на число раз выполнения этой строки и сложить данные величины».   
  
Но чему же равно tj? Это будет зависеть от входных данных. Рассмотрим разные случаи:  
Лучший случай: элементы уже отсортированы в нужном порядке. Если так, то каждый последующий элемент нужно сравнить с предшествующим и убедиться в корректности.  
  
Худший случай: Элементы входной последовательности отсортированы в порядке, обратном требуемому. Т.е. каждый элемент будет сравниваться со всеми предшествующими ему элементу, чтобы поставить в начало массива. И время tj мы можем вычислить по формуле арифметической прогрессии.  
  
Константы и функции, которые растут медленнее, чем самая быстрорастущая функция можно отбросить.   
В общем случае алгоритм будет работать не хуже, чем верхняя граница (худший случай) О(n2).

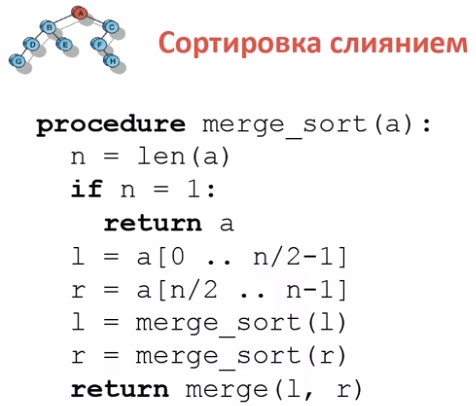
Верхняя оценка времени работы алгоритмов:  
Самый удобный способ оценки работы алгоритма.   
Время работы цикла for - O(n). Для цикла while оно такое же, так как в этом цикле происходит сравнение текущего элемента с предшествующими, а предшествующих элементов не может быть больше, чем n. И по свойству асимптотических умножений, так как цикл wile запускается при каждой итерации цикла for, общее время – их произведение:

.

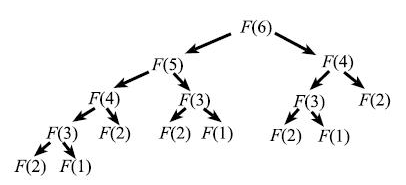
**Лекция 2.**

**Сортировка слиянием (сортировка Фон-Неймана)**

- Использует принцип «разделяй и властвуй»   
- Работает за время O(N\*log(N))   
- Использует О(N) дополнительной памяти   
Ключевой элемент – алгоритм, который сливает два отсортированных массива. На выходе выдается массив из тех же элементов, отсортированных в порядке возрастания.  
  
Цикл работает до тех пор, пока индексы в первом или втором массиве не дойдут до максимальной длины массива. Как только мы доходим до конца одного из массивов, цикл завершается. Условие, которое мы проверяем, это мы или дошли до конца второго массива или (мы еще не дошли до конца первого массива И итый элемент первого массива меньше или равен джитого элемента второго массива). В данном случае в результирующий массив добавляем меньший элемент а\_итое и увеличиваем счетчик «индекса» для массива А на один. В противном случае в результирующий массив добавлем б\_джитое и увеличиваем счетчик идекса массива Б на 1.  
Время работы алгоритма O(n+m)

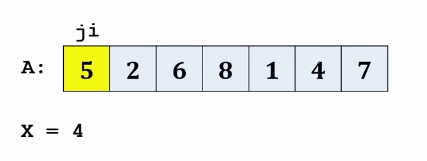
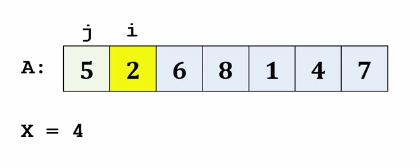
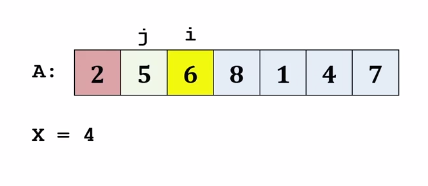
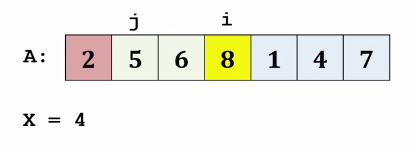
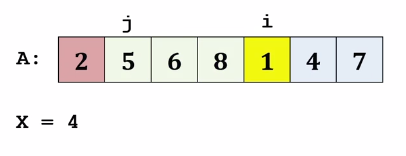
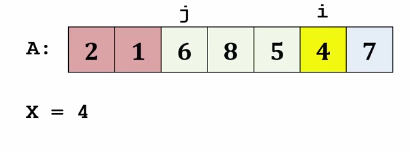
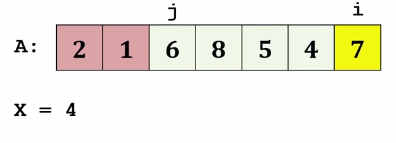
*Код сортировки слиянием:*Вычисляем длину нашего массива. Проверяем, не равна ли она единице. Если равна – то возвращаем сам массив, так как он уже отсортирован. Дальше делим массив на две половинки от 0 до n/2-1 и от n/2 до n-1. Далее рекурсивно вызываем этот же метод для каждой из половинок до тех пор, пока в массиве не будет по одному элементу. Как только в массиве по одному элементу, собираем все в один отсортированный массив методом merge(l,r). Как это происходит за счет рекурсии? Потому что все элементы храняться в стэке и затем мы можем методом merge их сравнивать и складывать в новый массив.

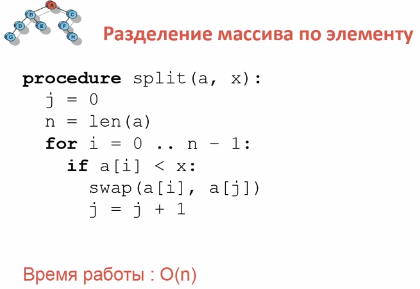
*Дерево рекурсии*

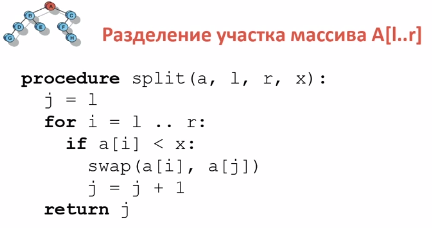
  
Для вычисления F6 нужно вычислить F5 и F4 и т.д., вниз по дереву.  
Основной недостаток, это то, что мы будем вычислять одни и те же значения несколько раз (например F3 или F4). Так для больших n могут возникнуть серьезные проблемы. Способ решения прост – запоминать уже найденные значения. Но для этого придется активно использовать память. Глубина рекурсии не больше log(n).   
Оптимизация:   
- Не создавать каждый раз новый массив, а создать один изначально.   
- Можно обойтись без рекурсии: пойти снизу вверх и сливать их, начиная с одного элемента.

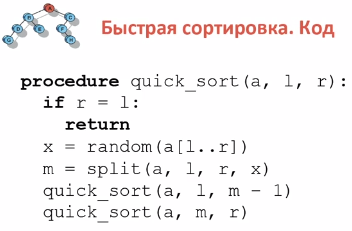
**Быстрая сортировка (сортировка Хоара)**

- Использует принцип «разделяй и властвуй»   
- Использует генератор случайных чисел   
- Работает в среднем за время O (N \* log(N)). В худшем – O(n2)   
- Использует O(1) дополнительной памяти

Основная процедура быстрой сортировки – разделение массива.  
Пример:   
Дан массив А -> Выберем случайный элемент массива Х -> Разделим массив на два: меньше Х и больше или равен Х.  
Разбиваем на куски того же самого массива, чтобы не тратить лишнюю память.  
То есть Мы проходим по всему массиву с двумя индексами: i,j.  
I отвечает за «кучку» элементов, меньших Х, J отвечает за «кучку» элементов, больших Х.   
Таким образом изначально i=0,j=0. И идем слева направо, сравнивая каждый элемент с Х. Если элемент больше, то отправляем во вторую «кучку», присваивая индексу j индекс этого элемента. Идем дальше. Если элемент меньше Х, то отправляем этот элемент в первую «кучку», для этого берем самый левый элемент во второй кучке и меняем местами элементы с индексами i и j. И так получается, что все элементы, которые левее j – относятся к первой кучке, что правее – ко второй.   
 ->  ->  ->  
 ->  ->  ->   
 ->   
  
Когда количество элементов в первой кучке увеличивается на 1, то и j увеличивается на 1, то есть сдвигается вправо.

*Пример кода:*   
  
Проходим по всему массиву от 0 до n-1. Сравниваем каждый элемент с Х. Если элемент меньше Х, то меняем местами текущий элемент (который сравнивался) и элемент под номером j. Также j увеличиваем на 1.

Разделение куска массива для рекурсивного вызова:  
   
Массив a, а также границы от l до r и саму переменную х, с которой ведется сравнение.  
И теперь цикл у нас не по всему массиву, а только от l до r. Метод возвращает значение j, то есть индекс первого элемента во второй кучке. То есть левая кучка начинается от l и заканчивается j-1, а правая начинается от j и заканчивается в r.

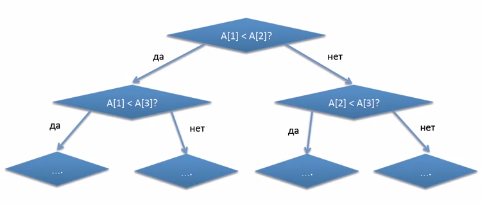
Код сортировки:   
  
Если r == l, то значит мы сортируем кусок размера 1. Этот кусок уже отсортирован -> возвращаемся. Выбираем случайный элемент в диапазоне значений массива с индексами от l до r. Получаем массив из двух кусков от l до m-1 и от m до r. После этого рекурсивно отсортируем оба куска той же самой процедурой.

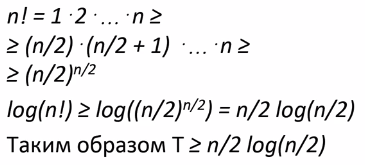
**Код из лекции работает только в том случае, если в массиве все элементы разные.  
Как это исправить??**

Нижняя оценка на время работы сортировки сравнением.

Пример проще:   
Докажем, что максимум в массиве можно найти не быстрее, чем за O(n).  
Пусть просмотр одного элемента занимает С.  
Если программа работала меньше, чем С\*n, значит она не посмотрела какой то элемент массива. Если этот элемент максимальный, программа выдаст неверный ответ, так как у нее нет всех данных.

Докажем, что любой алгоритм сортировки работает за время не меньше C \* n\* log(n) , C – const.  
  
Будем считать, что элементы абстрактные объекты.  
Единственная операция – сравнить два объекта друг с другом.  
(на практике не всегда так)

Для доказательства воспользуемся деревом решений:   
   
Это If.  
Рассмотрим программу, которая сортирует массив из двух элементов. Если первый меньше, оставляем, как есть. Если больше, меняем местами.  
Если больше элементов, то это больше сравнений:  
   
Каждый раз мы из одной ветки получаем две и идем дальше сравнивать.  
Если сделано Т сравнений, то есть не более 2Т вариантов ответа.  
Для сортировки нужно выбрать одну из n! перестановок массива   
Чтобы программа работала всегда правильно, количество вариантов, которое она может получить (2Т), должно быть не меньше, чем количество всех вариантов, которые могут быть правильными ответами (n!):  
2Т >= n!  
Прологарифмировав обе части неравенства, получим:  
T >= log2(n!)

Оценим log2(n!):  
   
Это грубая оценка. Но нам пойдет ☺