**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.

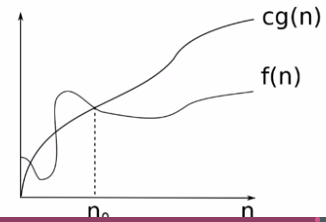
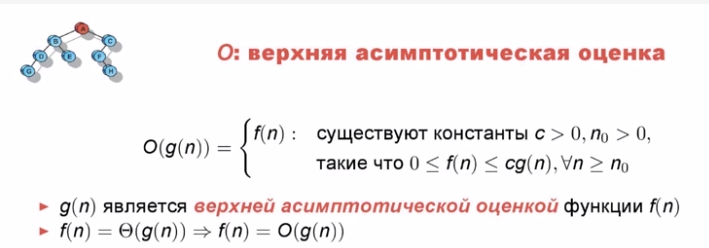
Зачем нужны эффективные алгоритмы? – Ресурсы компьютера ограничены (время работы, память)   
- Маршрутизация пакетов (поиск кратчайших путей)   
- Фильтрация спама (фильтры домена)   
- Базы данных, строковое алгоритмы  
- Сборка генома и тд.

Сложность алгоритмов, пример:

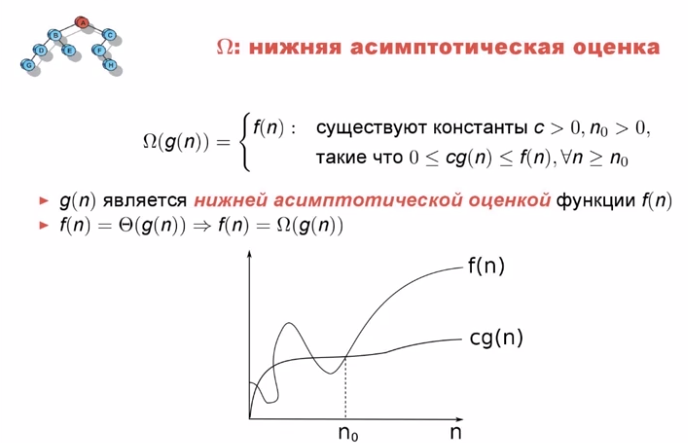
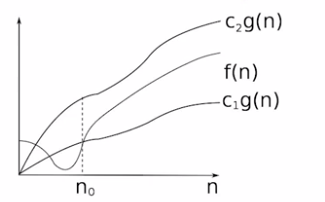
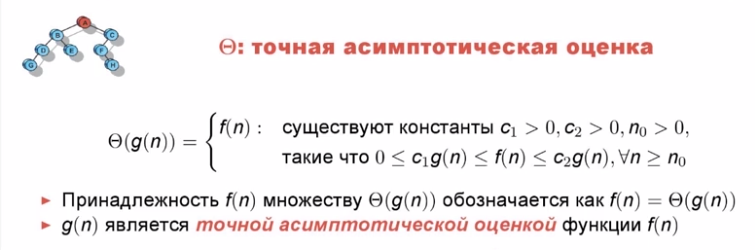
|  |  |  |
| --- | --- | --- |
| Сортировка | Вставками | Слиянием |
| Сложность | ~c1 \* n2 | ~c2 \* n\*log2n |
| От чего зависят c1, c2 – От реализации. На каком языке. | Ассемблер, c1 = 2 | JS, c2 = 100 |
| Вычислитель | Суперкомпьютер | Смартфон |
| Скорость | 1011 операций в секунду | 108 |
| Время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

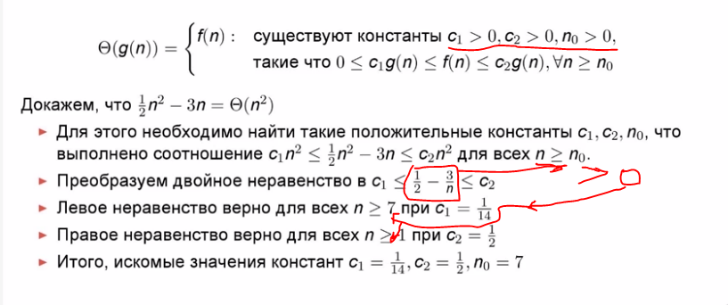
Оценка эффективности работы алгоритмов.

*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.  


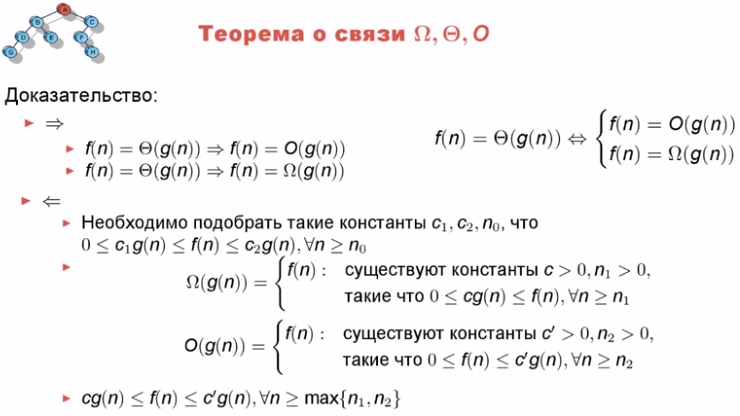


Время работы алгоритма: T(n), где n – целое число, размер входных данных.   
Входные данные можгут быть одним или несколькими числами.  
Пример Algoritm A(int n):   
1. j <- 1  
2. for i=1, to n do  
3. j <- j\*i

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)

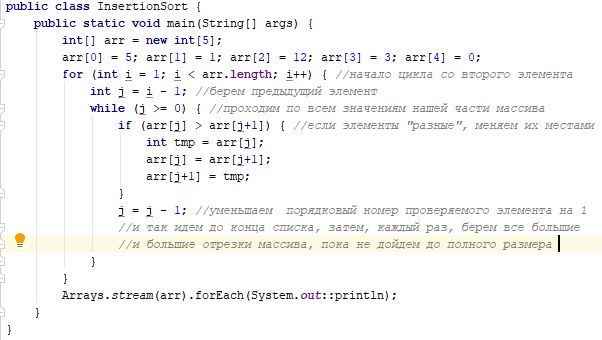
Теорема связи асимптотических оценок:

  
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.**

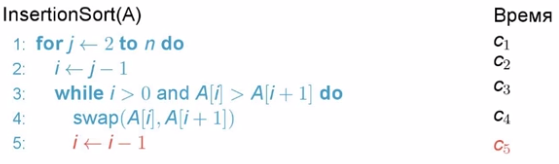
Задача сортировки чисел по возрастанию.   
На вход подается последоватьность чисел a1, …, anВыход: a’1, …, a’n , где a’1 <= a’2 <= a’3 …Пример:  
Вход – 10, 8, 2, 12, 5, 100  
Выход – 2, 5, 8, 10, 12, 100

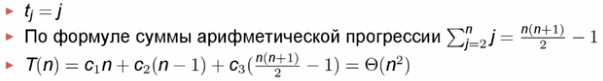
Можно сортировать все, что угодно, не только цифры.

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:  


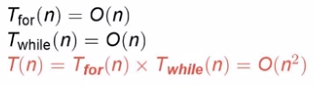
<!-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* -->

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 :

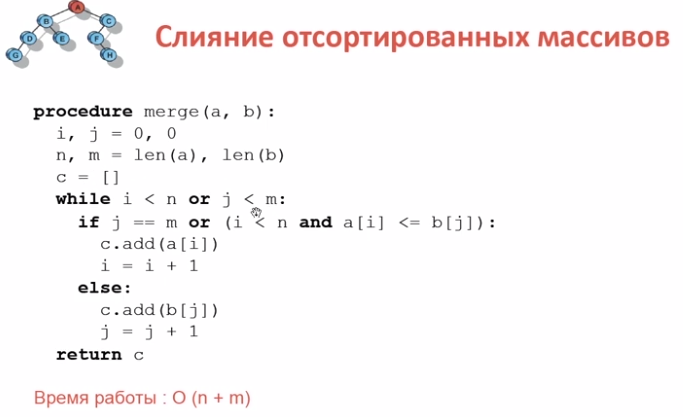
Для суммарного времени выполнения алгоритма достаточно умножить «сколько выполняется одна строка на число раз выполнения этой строки и сложить данные величины».   
  
Но чему же равно tj? Это будет зависеть от входных данных. Рассмотрим разные случаи:  
Лучший случай: элементы уже отсортированы в нужном 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).

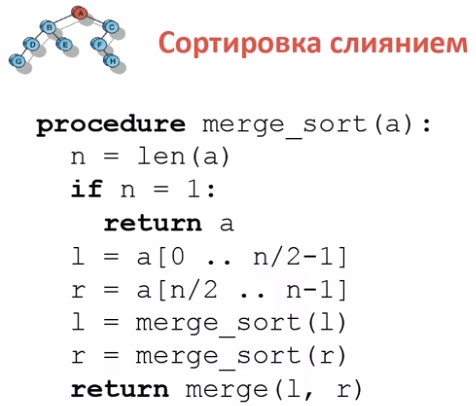
Верхняя оценка времени работы алгоритмов:  
Самый удобный способ оценки работы алгоритма.   
Время работы цикла for - O(n). Для цикла while оно такое же, так как в этом цикле происходит сравнение текущего элемента с предшествующими, а предшествующих элементов не может быть больше, чем n. И по свойству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

.

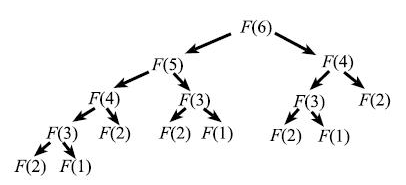
**Лекция 2.**

**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)

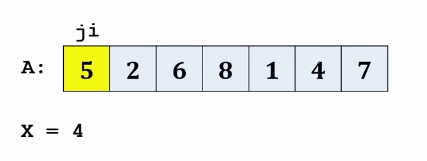
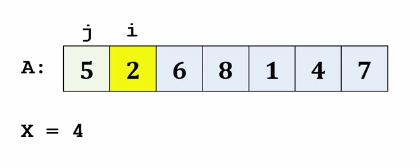
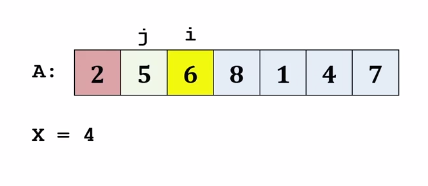
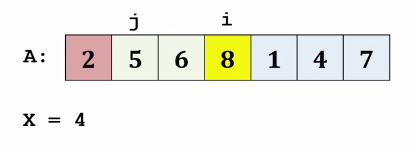
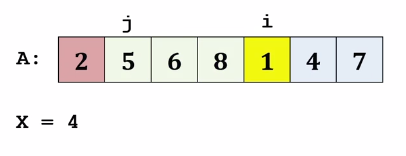
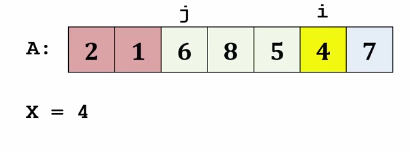
*Код сортировки слиянием:*Вычисляем длину нашего массива. Проверяем, не равна ли она единице. Если равна – то возвращаем сам массив, так как он уже отсортирован. Дальше делим массив на две половинки от 0 до n/2-1 и от n/2 до n-1. Далее рекурсивно вызываем этот же метод для каждой из половинок до тех 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.

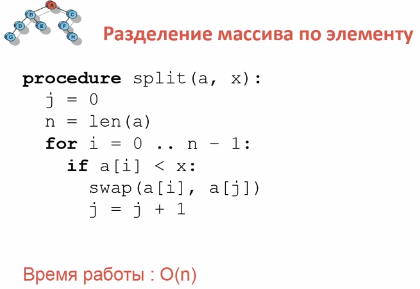
*Дерево рекурсии*

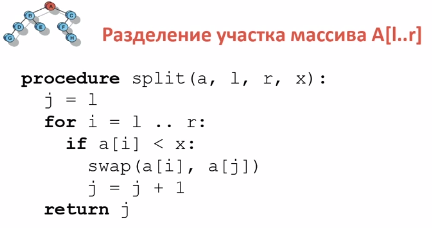
  
Для вычисления F6 нужно вычислить F5 и F4 и т.д., вниз по дереву.  
Основной недостаток, это то, что мы будем вычислять одни и те же значения несколько раз (например F3 или F4). Так для больших n могут возникнуть серьезные проблемы. Способ решения прост – запоминать уже найденные значения. Но для этого 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.

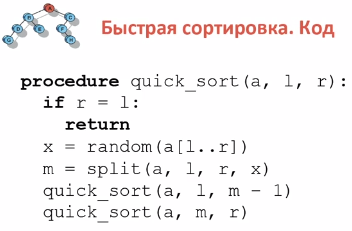
**Быстрая сортировка (сортировка Хоара)**

- 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.

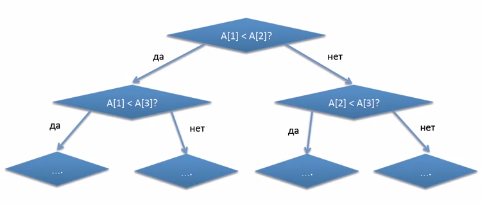
Код сортировки:   
  
Если r == l, то значит мы сортируем кусок размера 1. Этот кусок уже отсортирован -> возвращаемся. Выбираем случайный элемент в диапазоне значений массива с индексами от l до r. Получаем массив из двух кусков от l до m-1 и от m до r. После этого рекурсивно отсортируем оба куска той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

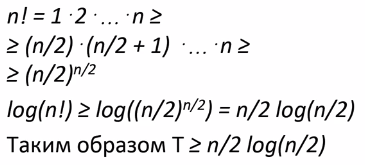
**Код из лекции работает только в том случае, если в массиве все элементы разные.  
Как это исправить??**

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 ☺

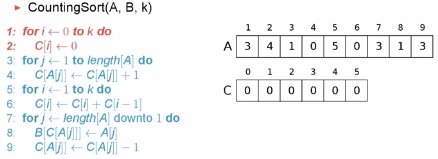
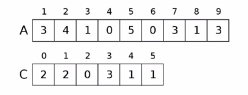
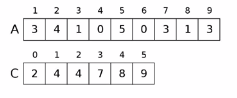
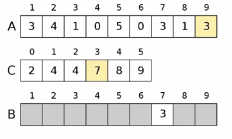
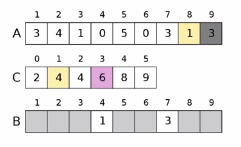
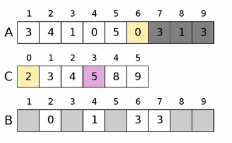
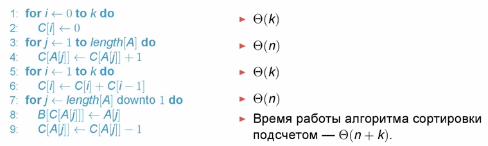
**Лекция 3.**

Алгоритмы сортировки за линейное время.

*Сортировка подсчета.*

Предназначена для сортировки массива из n целых чисел, лежащих в интервале от 0 до k, где k – некоторая константа.  
Если k = O(n), то время работы сортировки подсчетом θ(n).   
Идея сортировки подсчетом – для каждого элемента х определить число элементов, которые меньше х.  
Если, например, число элементов меньших х = 5, в отсортированном массиве элемент х займет 6 позицию.   
Однако, если не все элементы массива разные, то данный подход надо модифицировать.

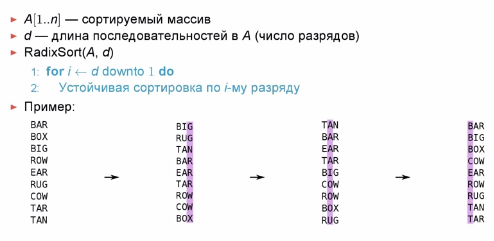
**Алгоритм сортировки подсчетом.**

A[1..n] – сортируемый массив  
B[1..n] – отсортированный массив  
C[1..n] – массив, для подсчета для каждого элемента х числа элементов, не превышающего его  
  
Изначально, массив С заполняем нулями. (шаги 1-2)   
На следующем шаге для каждого элемента массива А подсчитывается, сколько раз он встретился в массиве. Массив С имеет длину самого максимального элемента массива А. (шаги 3-4)   
  
Далее, для каждого элемента массива А подсчитываем, сколько элементов не превышает его. (шаги 5-6). То есть число элементов, не превышающих 0 = числу нулей. Число элементов, не превышающих 1, это число нулей и число единиц, и т.д.  
   
Далее, начиная с конца массива А, мы для каждого элемента определяем его позицию в результирующем массиве В.  
   
  
  
  
  
  
  
  
  
  
Мы смотри, сколько элементов не превосходит рассматриваемый, и ставим элемент на такую позицию в массиве В. Затем мы уменьшаем число элементов, не превышающий рассматриваемый, на 1. (шаги 7-9)   
-> ->   
Оценим время работы:   


**Цифровая сортировка.**

Предназначена для сортировки массива последовательностей одинаковой длины, состоящих из элементов, на которых задано отношение линейного порядка.  
По аналогии с разрядами чисел будем называть элементы, из которых состоят сортируемые последовательности, разрядами.  
Алгоритм состоит в сортировки заданных последовательностей какой-либо устойчивой сортировкой по каждому разряду, в порядке от младшего разряда к старшему.

*Алгоритм цифровой сортировки.*

  
Сначала мы сортируем элементы по последнему разряду, затем по предпоследнему и так до первого.  
Почему важна устойчивость сортировки? Например, последовательности TAN и TAR. Они отличаются только младшим разрядом. Соответственно, они будут отсортированы друг относительно друга после сортировки по последнему младшему разряду. Если мы будем использовать неустойчивую сортировку, то при сортировке по второму и первому разряду порядок может нарушиться, потому что в этих разрядах данные совпадают.

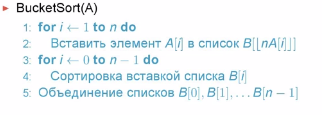
Устойчивая сортировка – сортировка, которая не меняет относительный порядок сортируемых элементов, имеющих одинаковые ключи(значения). Чаще всего устойчивость не соблюдается, так как для ее реализации нужна дополнительная память.

Время работы цифровой сортировки.  
Если время устойчивой сортировки линейно, то время работы цифровой сортировки так же линейно.   
Пусть k – число значений, которые могут принимать элементы сортируемых последовательностей.   
Если k = O(n), то время работы изученной ранее сортировки подсчетом линейно.   
Таким образом, если в качестве устойчивой сортировки выбрать сортировку подсчетом, то время работы цифровой сортировки будет равно θ(n).

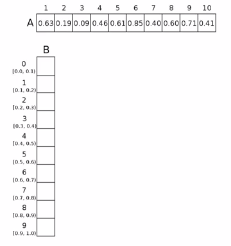
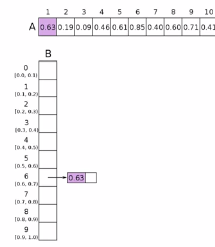
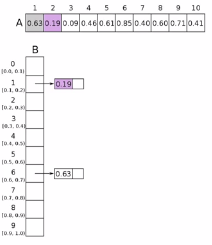
**Карманная сортировка.**

Предназначена для сортировки данных, равномерно распределенных в некотором интервале.  
Идея данной сортировки – разбить интервал сортируемых чисел на n одинаковых интервалов – карманов- и распределить по ним n сортируемых чисел. Число попадает в карма, если оно лежит в подинтервале, соответствующему данному карману. Далее в каждом кармане производится сортировка чисел, а зачем последовательно перечисляется содержимое каждого из карманов. Предполагается, что в каждом кармане окажется НЕ очень много элементов, поэтому в качестве сортировки можно использовать любую сортировку.

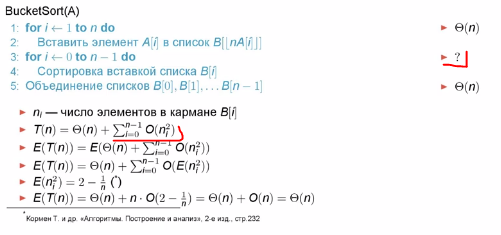
Алгоритм карманной сортировки.

A[1..n] – сортируемый массив  
B[0..n-1] – Массив связных списков (карманов). В нем содержатся указатели на головы этих связных списков.  
   
1-2: Сначала каждый из элементов массива А вставляется в соответствующий ему карман. Для того, чтобы вычислить номер кармана, необходимо вычислить округленное вниз значение произведения числа сортируемых элементов и текущего вставляемого элемента.  
3-4: Каждый из связных списков сортируется вставкой.  
5: Последовательное перечисление отсортированных списков.

Пример работы карманной сортировки:

  
Так как массив А состоит из 10 элементов, то вводим вспомогательный массив В, так же состоящий из 10 элементов. В массиве В хранится указатель на голову соответствующего ему списка(кармана). Например, в В[0] хранится указатель на список, в котором будут содержаться элементы от 0 до 0,1. Последовательно проходим по массиву А и добавляем элементы в массив В.   
-> ->   
Затем, каждый из списков сортируется. Далее мы последовательно проходим по всем спискам и записываем элементы в отсортированный массив А.

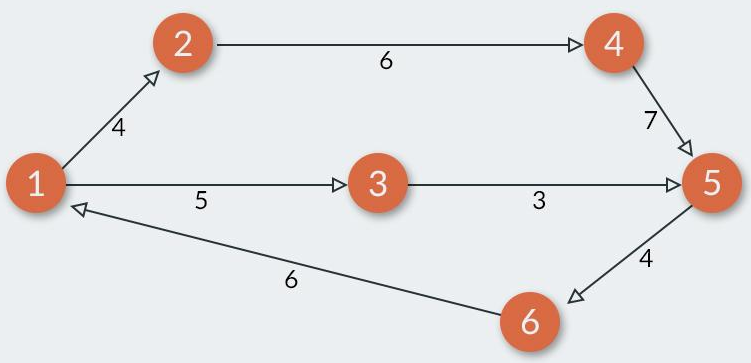
Время работы алгоритма карманной сортировки + псевдокод:

  
Где E(T(n)) – мат.ожидание.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Распечатано до сюда\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Дополнительная лекция. Javarush.**

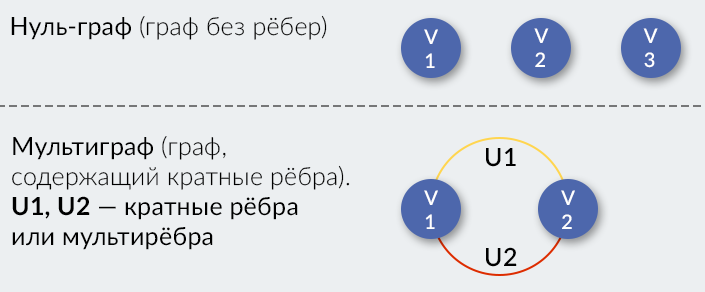
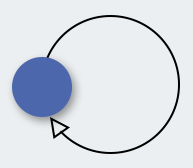
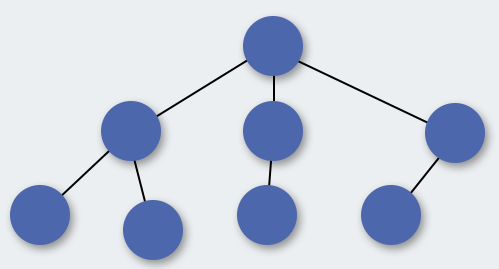
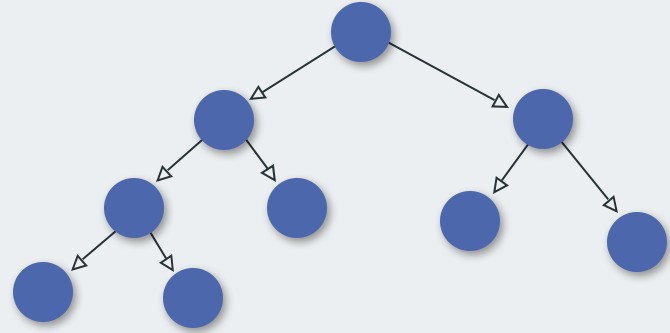
**Деревья, красно-черные деревья.**

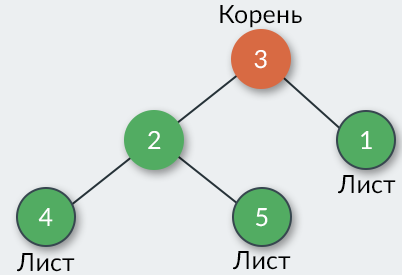
Граф – это система, состоящая из точек и линий, которые их соединяют. Точки называются вершинами, а линии – ребрами.  
   
Жизненный пример: вершины – это города, а ребра – дороги. Тогда поиск самой короткой дороги превращается в задачу «дан граф, найти кратчайший путь между двумя вершинами».

Если в графе используются стрелки – это ориентированный граф, если просто линии – неориентированный граф.

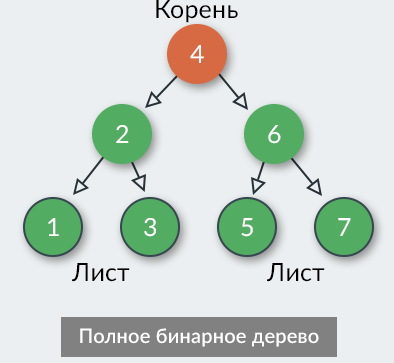
Если в графе каждая вершина соединена ребром с каждой, то граф называется полным. Если в графе по ребрам можно добраться до любой вершины, то он называется *связным*. Граф, состоящий из трех отдельных вершин без ребер вообще, тоже граф, но *несвязный*.

Некоторые типы графов:

   
Петля:  
   
  
  
  
  
Дерево – связный граф без циклов, то есть без петель и кратных ребер.  
   
Ориентированное (направленное) дерево:  


Чтобы соединить в связный граф N вершин, надо минимум N-1 ребер. Такой граф называется деревом. При этом одна из вершин называется корнем дерева, а остальные ее ветвями. Ветви дерева, которые не имеют своих ветвей – листья.  


*Бинарное дерево* – дерево, у которого у каждого элемента два потомка. Т.е. их может быть 0 или 2. (выше как раз оно)

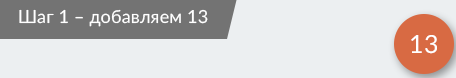
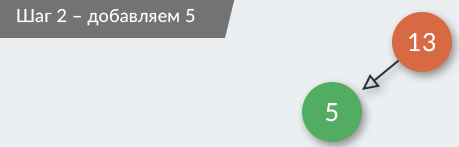
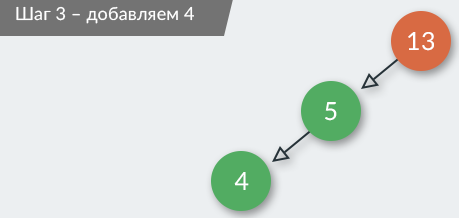
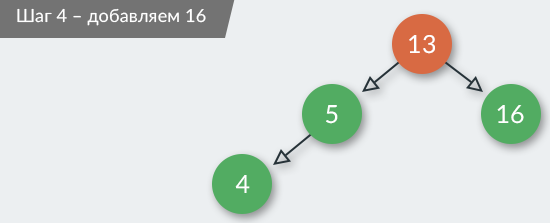
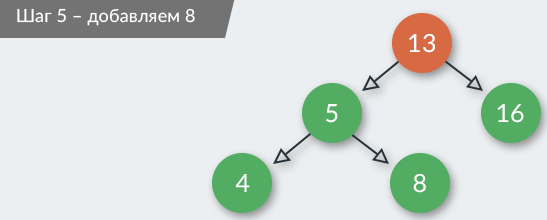
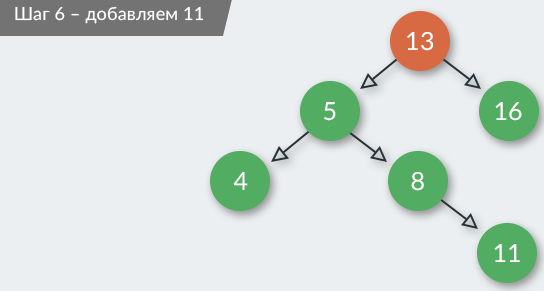
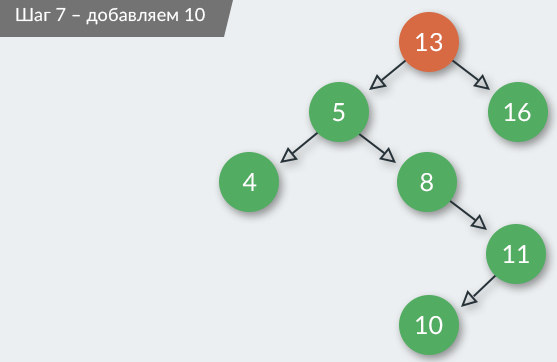
*Полное бинарное дерево* – дерево, у которого у каждой ветви 2 потомка, а все листья без потомков находятся в одном ряду.  


Правило (для каждого элемента) формирования дерева:

*Значение, которое хранится в потомке справа, больше, чем значение в вершине, а значение, которое хранится в потомке слева – меньше, чем значение в вершине.*

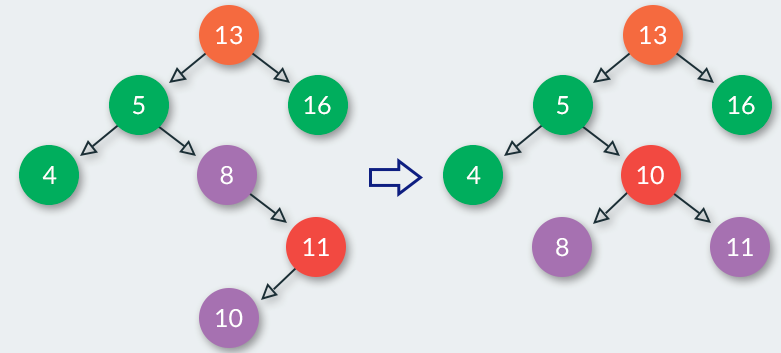
Сортировка элементов дерева выполняется добавлением (обычно).

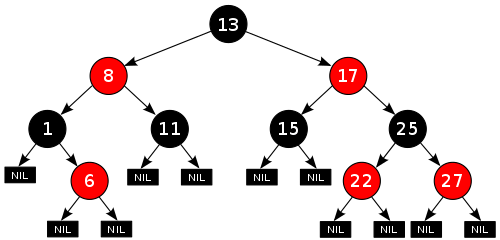
Пример: 13, 5, 4, 16, 8, 11, 10

*Сбалансированное дерево* – дерево без «перескоков», т.е. без длинных ветвей. Если мы подавали бы элементы при «строительстве» дерева в уже отсортированном порядке, у нас бы получилось длинное дерево, состоящее из одной ветви.

Самое эффективное дерево – дерево, которое имеет ветви примерно равной длины. Тогда при каждом сравнении отбрасывается наибольшая часть поддерева.



Красно-черное дерево – самобалансирующееся дерево. Т.е. если после добавления элемента в дереве возникает перекос, оно немного меняет порядок элементов, и все становится ок.  
Каждые вершины окрашены либо в красный, либо в черный цвет. И разные цвета подчиняются разным правилам:   
- Красная вершина не может быть сыном красной вершины.   
- Черная глубина любого листа одинакова (Черная глубина – количество черных вершин на пути из корня).   
- Корень дерева черный.   
- Оба потомка красного узла – черные.   
- Все листья(NIL) – черные.   
- Всякий простой путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число чёрных узлов.  
Высота дерева пропорциональна логарифму от числа узлов.   


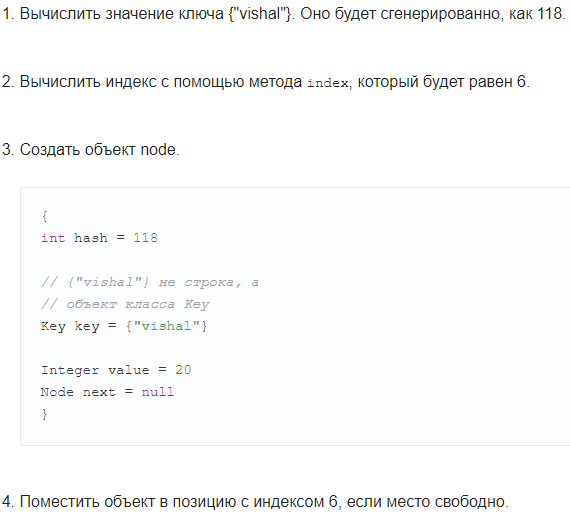
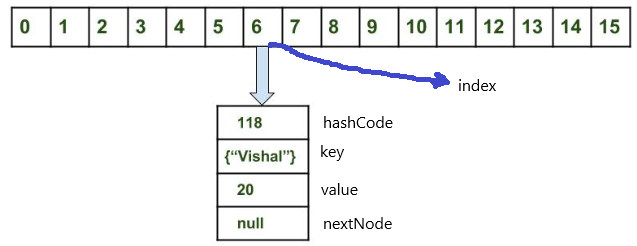
TreeMap в Java основана на красно-черном дереве.   
TreeSet – коллекция, которая хранит в себе элементы в виде упорядоченного дерева. Внутри TreeSet<E> содержится TreeMap<E, Object>. Поэтому у него очень быстрые операции add, remove, contains.  
**Дополнительная информация.** А в HashSet<E> хранятся значения хэшей объектов, другими словами, это HashMap<E, Object>, где ключи – это хэши, а значения – наши объекты. Для объектов обязательно переопределять hashCode() и equals(). И если вдруг мы поменяем объект, и при этом изменятся его внутренние данные, которые используются в вычислении хэша. И тогда Хэш изменится. А значит мы не найдем этот объект в сете. Таким образом поиск в HashSet и HashMap гарантировано работает правильно, если объект Immutable.

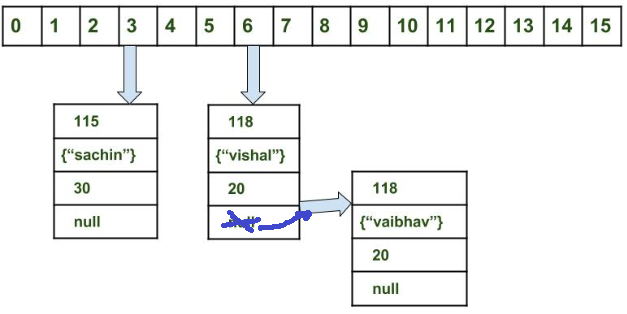
**Методы hashCode() и equals().**

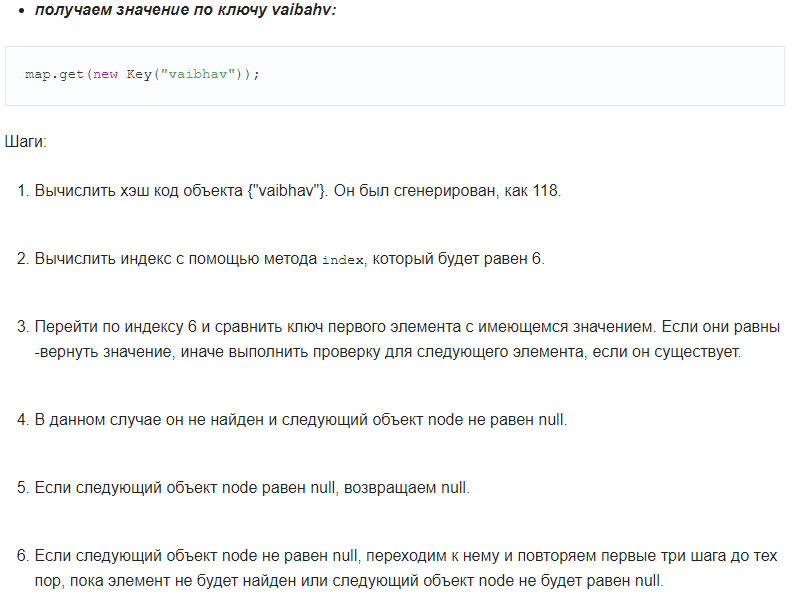
hashCode() – возвращает ссылку памяти объекта в целочисленной форме.   
Если хэш-коды разные, то объекты гарантировано разные.   
Если хэш-коды одинаковые, то НЕ ФАКТ, что объекты одинаковые.  
Если объекты одинаковые, то хэш-коды тоже одинаковые.  
equals() – если не переопределен, то сравнивает ссылки на объекты. Вызов new создает новую ссылку, но если:   
Object obj1 = new Object();  
Object obj2 = obj1;  
То у нас на оба объекта указывают одни и те же ссылки. И тогда метод hashCode() вернет одинаковые значения, а equals() true.

Принцип добавления ключ-значение в хэш-мап:

Корзина (Entry[]) HashMap состоит из следующих элементов:   
- int — хэш  
- K — ключ  
- V — значение  
- Node — следующий элемент

Шаги:  
   
То есть HashMap будет выглядеть следующим образом:   


Если добавлять еще один элемент, для которого хэш-код вычисляется таким же, как у предыдущего, то и индекс (номер корзины, в которую добавляется элемент) будет таким же. То тогда этот элемент будет добавлен в эту же корзину и там уже будет связный список (если будем накладывать и накладывать туда элементы. До определенного момента, а потом это все превратится в сбалансированное дерево – это сделано для оптимизации в Java 8.  


Для получения значения по ключу выполняются следующие действия:   


Важный момент:   
1. Сложность операций get() и put() практически константна до тех пор, пока не будет проведено повторное хэширование.  
2. В случае коллизий, если индексы двух и более объектов node одинаковые, объекты node соединяются с помощью связного списка, т.е. ссылка на второй объект node хранится в первом, на третий во втором и т.д.  
3. Если данный ключ уже существует, значение перезаписывается.  
4. Хэш-код null равен 0.  
5. Когда объект получается по ключу, происходят переходы по связному списку, до тех пор, пока объект не будет найден или ссылка на следующий не будет равна null.

**Дополнительная лекция ТЕХНОСФЕРА.  
Алгоритмы и структуры данных. (лекция 2)**

*Жадные алгоритмы.*

Задачи:

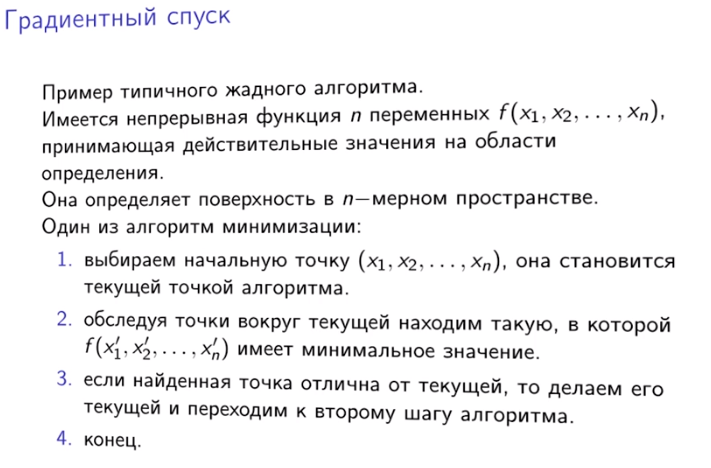
- Путешественник желает посетить несколько городов и потратить минимальную сумму денег.  
Найти порядок городов, с которым он посещает города.

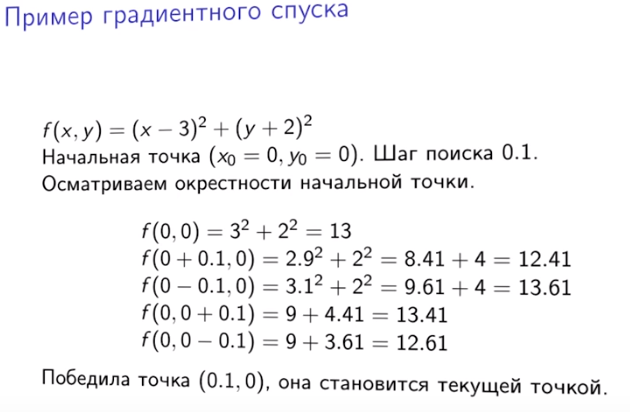
- Управление дорожного движения хочет определить длительность фаз светофора, при котором будет обеспечен наибольший трафик.

- Задача о рюкзаке (вес предмета-ценность). Рюкзак ограниченной вместимости.

Все эти задачи относятся к экстремальным значениям (максимум и минимум). Решение таких задач – оптимизация.

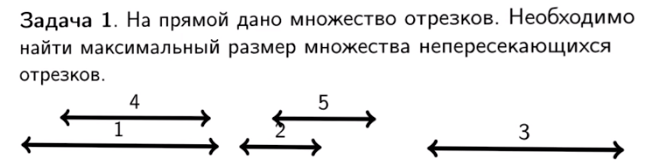
Жадные алгоритмы   
 - состоят из итераций  
 - принимают решение на каждом шаге, стараясь найти локально оптимальное решение.



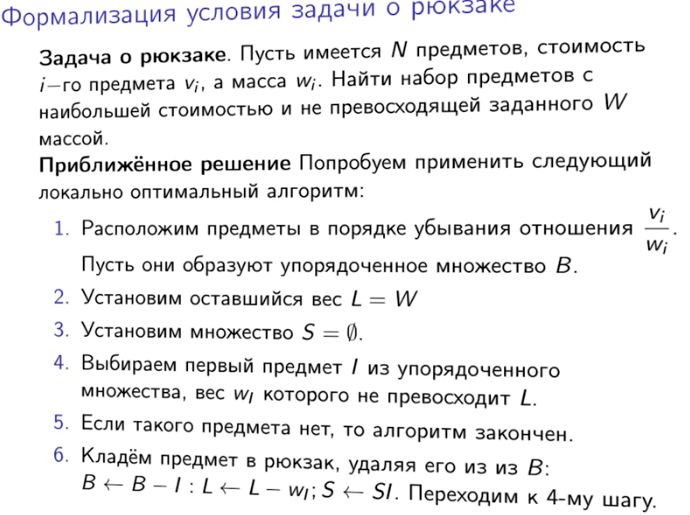


Данный алгоритм склонен к нахождению локального минимума.  
Для жадных алгоритмов иногда приходится выбирать «А по какому критерию мы будем делать следующий шаг?»

Задача об интервалах.

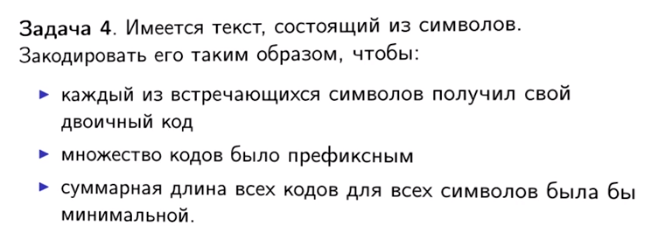


Чтобы решить задачу жадным алгоритмом, нужно выбрать какой-нибудь критерий.  
Для данной задачи:   
- Упорядочить отрезки по какому-либо признаку.   
- Рассматриваем отрезки по одному. Если он не перекрывается с каким-либо из уже внесенным в выходное множество, то добавляем его в этом множество.   
Жадность заключается в том, что каждый раз, когда мы видим подходящий вариант(рассматривая каждый очередной отрезок), то сразу хватаем его.  
  
*Главной проблемой жадных алгоритмов* является отсутствие «заглядывания» вперед. Они проверяют локально оптимально по какому-либо критерию шаги и надеются, что решение будет глобально оптимальным. Тщательный выбор критерия поможет найти оптимальное решение.

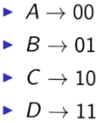
Задача о рюкзаке:   
Точное решение имеет сложность O(2N).   
Приближенное решение:  


*Сжатие информации. Алгоритм Хаффмана.*

Принадлежит к классу жадных алгоритмов.

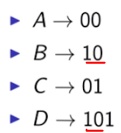


Пример:

Пусть имеется текст, состоящий из четырех символов:   
  
Его длина – 21 символ.   
По теореме об информации (число символов для кодировки = log(N), где N – число «возможных» букв ). Этот текст можно закодировать так:   
  
Общая длина кода составит 42 бита.

*Префиксный код* – код, в котором не имеется кодовых слов, начинающихся с других кодовых слов.

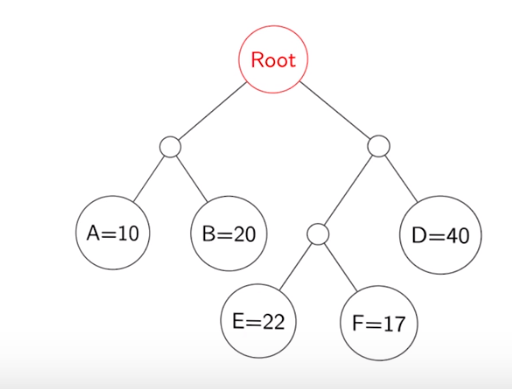
Префиксный код == код, удовлетворяющим условиям Фано.

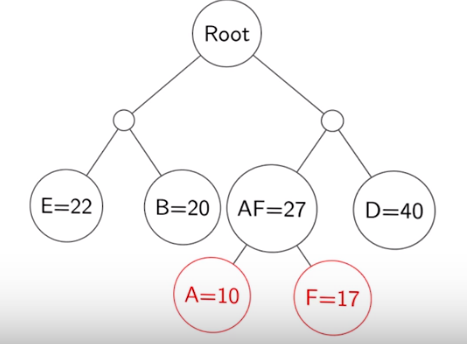
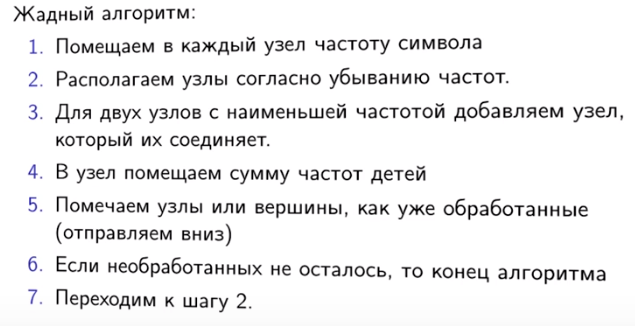


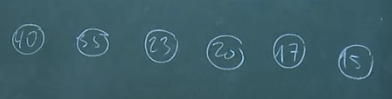
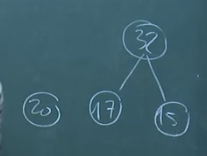
Код выше (для примера выше) не является префиксным, так как кодовое слово D начинается с кодового слова B.

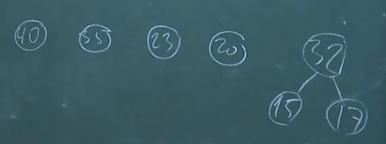


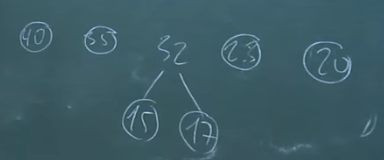
**Свойства оптимального дерева:**   
- из каждого узла должно исходить ровно два пути.   
- не должно быть пустых вершин.   
- самое длинное кодовое слово должно быть парным.

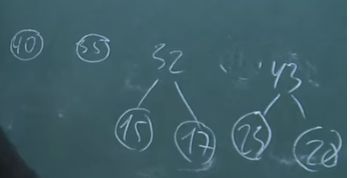
 - не оптимальное дерево

Для построения оптимального:   
1) Находим два самых «редких» символа (A, F). Они должны находится на самой длинной ветке. Если это не так, то можно поменять местами с символами с самой длинной ветки.  
2) Создаем некий вспомогательный узел (AF), который равен сумме этих узлов.  
  
3) Повторяем шаги 1-2, пока дерево не будет оптимальным.  
Более подробный алгоритм:  


 ->  ->

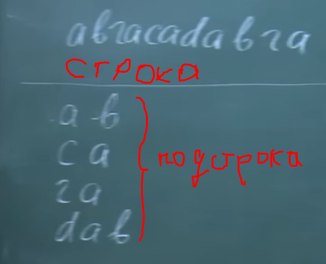
 ->

 ->

 ->

Таким образом, мы получаем рекурсивный алгоритм, в котором мы находим пары, находим их сумму, пары опускаем вниз, сумма занимает место в ряду и так повторяем.

**Задача:**



Можно ли из подстрок составить строку?

Решение:

**Лекция 4.**

**Элементарные структуры данных**

*Стек и очередь.*

Позволяют работать с динамическими множествами.

Динамическое множество – множество, которое поддерживает операции добавления и удаления элементов.

*Стек* – динамическое множество, операция удаления в котором определен следующим образом:   
- первым удаляется элемент, который был помещен в стек последним   
- Last In First Out

Реализация стека:   
Элементы хранятся в массиве S[1..n]. Размер стека не более n.   
top – индекс последнего помещенного элемента в стек.

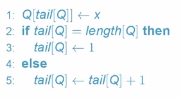
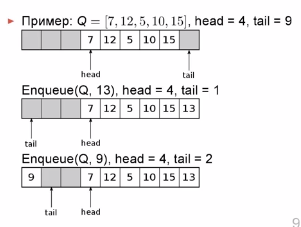
Если переменная top == 0, то стек пуст. Проверка выполняется за O(1).

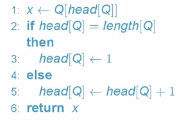
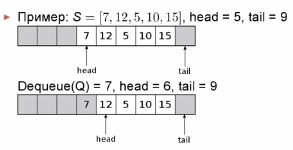
При добавлении в стек, необходимо проверить, не заполнен ли стек. Если все ок, то можем увеличить переменную top на 1 и в эту вершину положить элемент. Добавление выполняется за O(1).

Извлечение из стека так же без проверки не обходится. Возвращаем последний элемент с индексом top и уменьшаем индекс на 1. Извлечение выполняется за O(1).

*Очередь* – динамическое множество, операция удаления в котором определена следующим образом:   
- первым удаляется элемент, который был помещен в очередь раньше, чем все остальные элементы, содержащиеся в очереди  
- First In First out

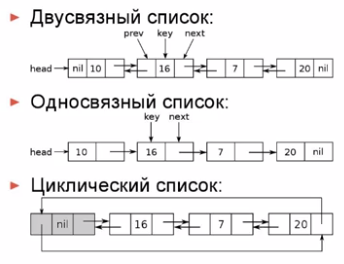
Реализация очереди:   
Элемент очереди хранится в массиве Q[1..n] и размер не превышает n.  
head – индекс первого элемента, tail – индекс места в массиве, куда будет добавлен новый элемент.

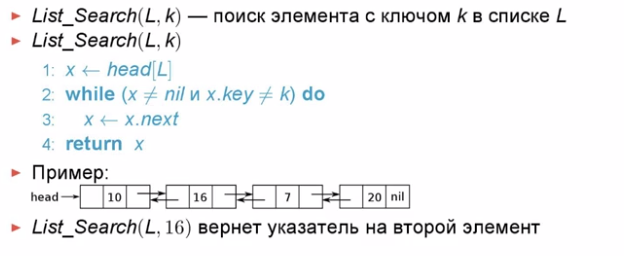
Добавление элементов в очередь:  
 плюс проверки на пустоту и полность.  
Проверка ифов позволяет сделать нам закольцованную очередь. Т.е. если в начале у нас есть место, то мы присваиваем индексу значение 1.   
 время работы О(1)

Извлечение элемента из очереди:   
 время работы О(1)   
Первым элементом удаляется самый первый элемент, который находится в голове. А затем устанавливаем новое значение переменной хеад.  


*Связанные списки*

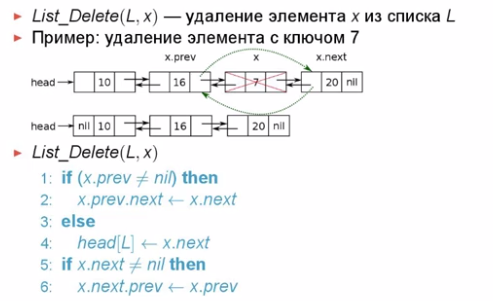
Связанные список – динамическое множество, элементы которого расположены в линейном порядке, определяемом указателями.

Поля элемента двусвязного списка:   
key – ключевое значение  
next – указатель на следующий элемент  
prev – указатель на предыдущий список  
Часто бывает hеad – указатель на первый элемент.  
  
Для циклических списков принято вводить пустой элемент (nil), чтобы точно знать, где голова (следующий элемент за этим)

Поиск в связном списке:   
  
Начинаем с элемента «головы» и идем так по списку, пока не найдем или не дойдем до nil. Если элемент отсутствует, то результатом поиска будет nil.  
Время работы: О(n);

Вставка элемента в список:   
Для этого нужно сначала создать элемент в списке, а затем уже добавлять данные.  
  
Элементы добавляются в начало списка и происходит переназначение головы.

Удаление из связного списка:  
При удалении, у предыдущего элемента меняем ссылку на следующий элемент после удаляемого. А для следующего после удаляемого меняем ссылку на предыдущий.

  
Это удаление именно удаления элемента, а не ключа. Если хотим удалить ключ – то его надо сначала найти, поэтому время работы удаления элемента по значению – O(1), а по ключу – O(n).

*Двоичное дерево.*

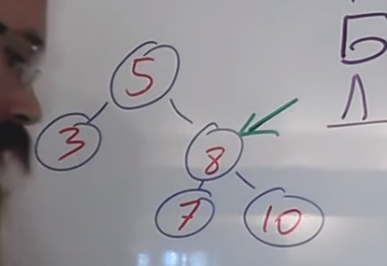
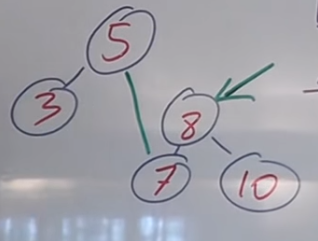
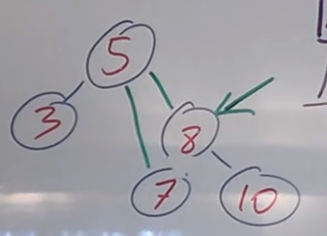
Двоичное дерево (бинарное дерево) – конечное множество узлов, которое:   
- либо пусто (пустое дерево)   
- либо состоит из трех непересекающихся множеств узлов: корневой узел – корень, двоичное дерево, называемое левым поддеревом и двоичное дерево, называемое правым поддеревом.

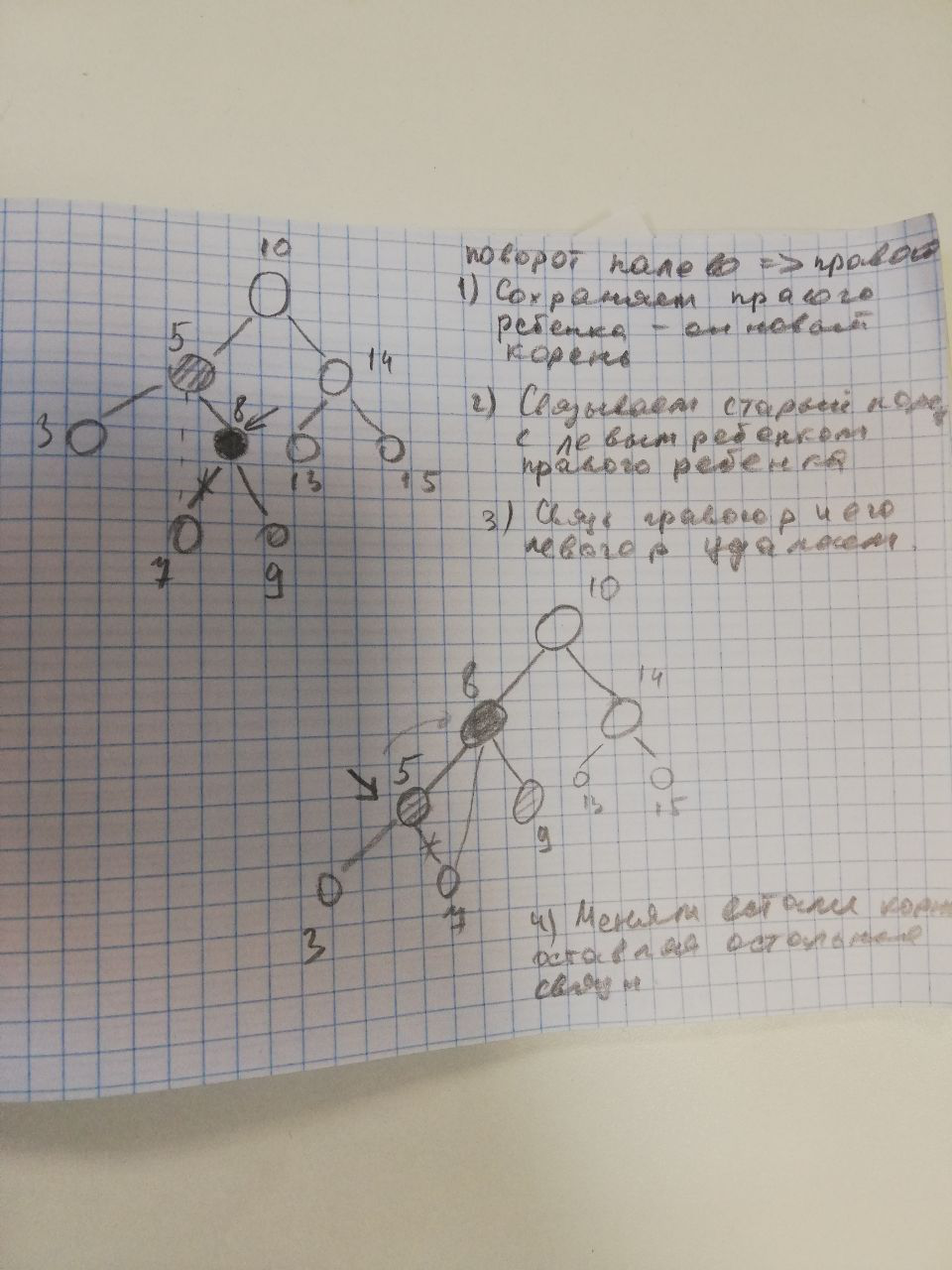
Любой узел y на пути от корня к узлу х – предок х.  
Если некоторый узел у является предком узла х, то узел х – потомок узла у   
Если левое(правое) поддерево узла х непустое, то его корень называется левым(правым) ребенком узла х   
Если узел х – ребенок узла н, то узел у – родитель узла х  
Узел, у которого отсутствуют оба ребенка называется листом  
Узел, не являющийся листом – внутренний узел.  
*Глубиной узла* называется число ребер на пути от корня к узлу  
*Высотой узла* называется число ребер в самом длинном пути от узла до листа  
*Высотой дерева* называют высотой корня

*Полное двоичное дерево* – если все листья имеют одну и туже глубину, а так же все внутренние узлы имеют по два ребенка.

**Лекция 5.**

Дополнительно: Поворот бинарного дерева

1. Задаем указатель на правый дочерний узел (если поворот влево)  
   
2. Затем меняем указатель правого поддерева на левое поддерево левого корня, т.е. изначальный корень у нас соединяется с левым потомком правого потомка  
      
   Мы точно знаем, что любое из значений справа от корня не меньше пятерки.
3. Мы можем, знаем, что 5 меньше, чем 8, меняем указатель нового корня  
   

Своими словами:   
1) Для поворота налево сохраняем правого ребенка корня, относительно которого поворачиваем, направо – левого ребенка.   
2) Соединяем левого(правого для поворота направо) ребенка нового корня и старый корень, а для левого(правого) ребенка нового корня устанавливаем наш старый корень.  
3) Новый корень ставим на место старого корня.  


**Алгоритмы, основанные на двоичной куче.**