Алгоритмы

Урок 1

Что такое алгоритм?

Алгори́тм — набор инструкций, описывающих порядок действий исполнителя для достижения результата решения задачи за конечное число действий (википедия).

Типичный пример: рецепт приготовления блюда. В отличие от алгоритма для компьютера, рецепт подразумевает некоторую свободу действий и пространство для импровизации. Компьютерный алгоритм должен быть четко описан и четко выполнен.

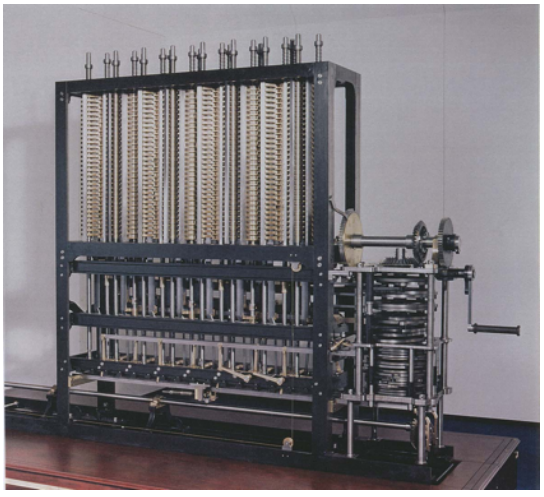
Программа ≠ алгоритм.
Программа существует и выполняется в компьютере и может использовать алгоритмы для реального решения задач.

Алгоритм – абстрактное понятие, описание идеи (вроде математической теории).

Нас интересует не сложность конкретной задачи, а рост сложности. Как сильно увеличится количество работы с увеличением входных данных?

Пример линейного роста: красить стену. В два раза больше площади – в два раза больше времени потребуется.

Пример нелинейного роста: поиск фамилии в телефонном справочнике. Если увеличить справочник в два раза, то время поиска не увеличится в два раза (если мы используем не глупый метод поиска от начала до конца).



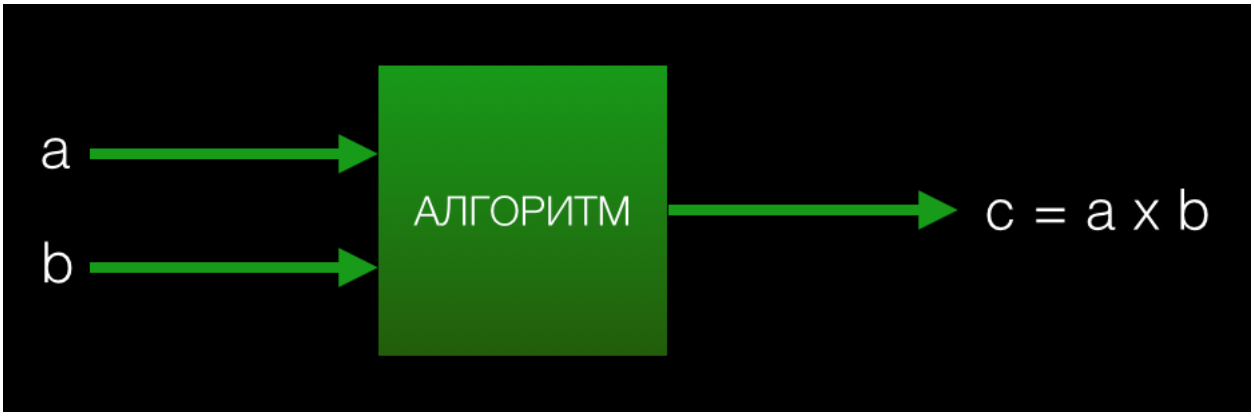
Немного истории

825 г., Абу Абдуллах Мухаммед ибн Муса аль-Хорезми, ученый из Хорезма. Слово “алгоритм” скорее всего произошло от его фамилии.

1834 г., Аналитическая машина Чарльза Бэббиджа. По задумке автора является программируемым устройством для решения целого класса вычислительных задач. Ада Лавлейс описала первые программы для аналитической машины. Графиня Лавлейс считается первым в мире программистом.

Пример алгоритма: умножение сложением
a x b = a + a + a … + a (b раз)

Абстракция – важное понятие в программировании и информатике. Алгоритм можно рассматривать как абстрактную коробку, которая принимает данные, производит какие-то операции и “возвращает” результат.




```
MULT-BY-ADD (a, b)

c ← 0
FOR each number from 1 to b
      c ← c+a

RETURN c
```

Сортировка

Сортировка – важная задача, которая решается повсюду.

Bogosort

Глупый алгоритм сортировки, полагающийся на удачу.

```
BOGOSORT (A)

while not isSorted(A)
      shuffle(A)
```

Сортировка пузырьком (Bubble sort)

Сравниваем все соприкасающиеся пары чисел, меняем числа местами если нужно. Самые большие числа “поднимаются” (как пузырьки в сосуде воды) в верх (конец) списка.

BUBBLE-SORT (array A of n numbers)

```
for i = (n–1) to 1
      for j = 0 to (i–1)
            if A[j] > A[j + 1]
                  swap A[j] with A[j+1]
```

(Видео: визуализация работы алгоритма сортировки пузырьком – https://www.youtube.com/watch?v=Cq7SMsQBEUw)

Анализ количества сравнений: для массива из n чисел:

$$(n-1) + (n-2) + \ldots + 1$$

сравнений. Это

$$n\,(n-1) / 2 = n^2/2 - n/2$$

n^2 это самый “тяжелый” элемент выражения, он больше всего влияет на сумму при изменении n. Поэтому мы используем его для грубой оценки производительности алгоритма – $O(n^2)$.

Сортировка вставками (Insertion sort)

Похож на сортировку пузырьком. Составляем отсортированный массив в начале списка, добавляем по одному элементу в этот массив и находим для него нужное место.

INSERTION-SORT (list A of n numbers)

```
for i = 1 to n-1
      j = i
      while j > 0 and A[j] < A[j-1]
            swap A[j] with A[j-1]
            j = j-1
```

Худший случай, числа отсортированы в обратном порядке:
O(n2) сравнений и перестановок

Лучший случай: числа отсортированы в правильном порядке:
O(n) сравнений и O(1) перестановок

Средний случай:
O(n2) сравнений и перестановок

(Видео: визуализация работы алгоритма сортировки вставками – https://www.youtube.com/watch?v=8oJS1BMKE64)

Модель вычислений RAM

- +, *, -, =, вызов, if – простая операция, 1 шаг
- циклы – комбинации простых операций
- обращение к памяти – 1 шаг

Дополнительно
Открытая лекция “Что такое алгоритмы” – http://habrahabr.ru/post/173385/

Ссылки
1. Визуализация алгоритмов сортировки (видео) https://www.youtube.com/watch?v=kPRA0W1kECg
2. Сравнительная визуализация алгоритмов сортировки http://sorting.at/
3. Еще одна визуализация с разными параметрами http://www.sorting-algorithms.com/
4. Статья про Чарльза Бэббиджа и его аналитическую машину https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%80%D0%BB%D1%8C%D0%B7_%D0%91%D1%8D%D0%B1%D0%B1%D0%B8%D0%B4%D0%B6
5. Аналитическая машина в музее компьютерной истории (+видео) http://www.computerhistory.org/babbage/

Урок 2

Идея “разделяй и властвуй”: разделить задачу на несколько более простых задач.

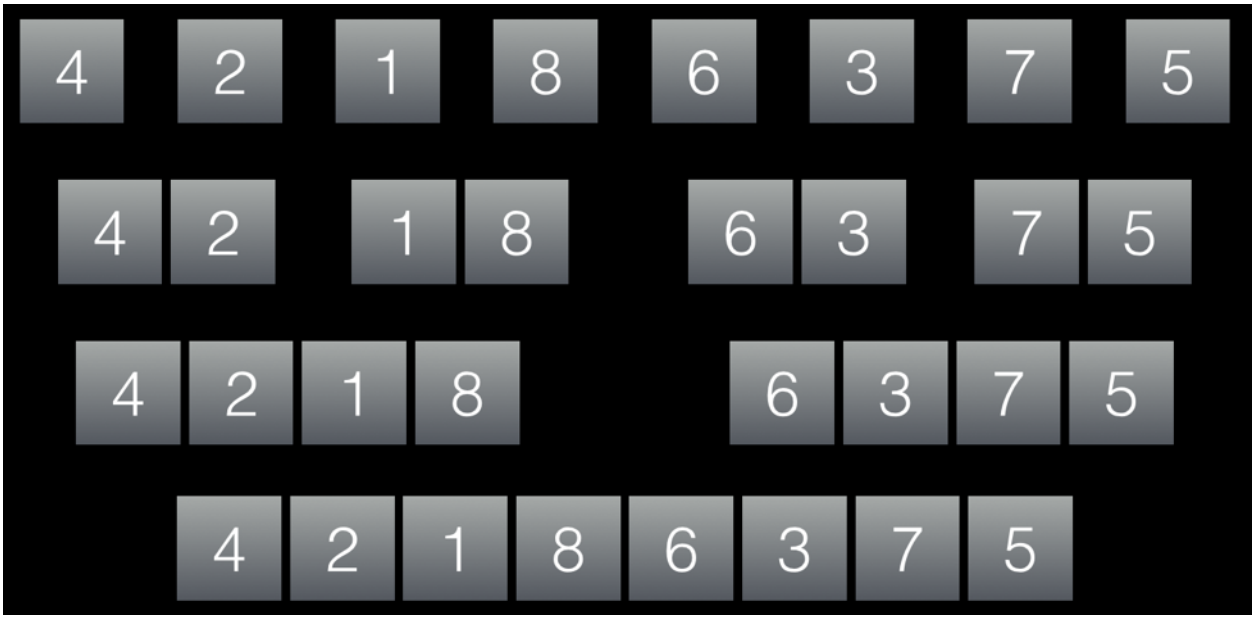
Линейный поиск: начать с начала и последовательно проверять числа.
Двоичный поиск (работает только в отсортированном массиве): разделить массив на две части, определить в какой из частей находится нужное число, перейти к нужной части, повторить операции.

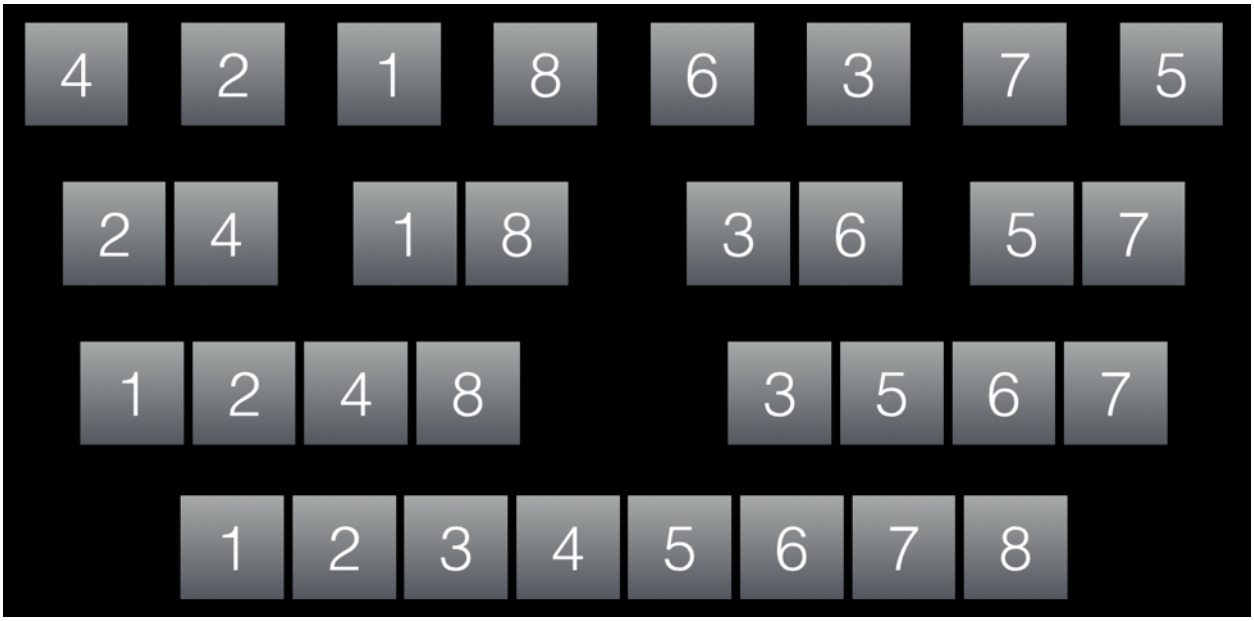
Линейный поиск – примерно n/2 сравнений.
Двоичный поиск – примерно $\log_2(n)$ сравнений.

Для массива из 100 элементов
линейный поиск – 50.5 сравнений
двоичный поиск – 7.72
$\log_2(100) = 6.64$

Для массива из 1000 элементов
линейный поиск – 500.5 сравнений
двоичный поиск – 10.976
$\log_2(100) = 9.96$

Сортировка слиянием (Merge sort)
1. Разделить массив на две части.
2. Продолжить делить каждую часть пополам пока не дойдем до n отсортированных массивов, каждый из которых состоит из одного элемента.
3. Сливать отсортированные массивы попарно.

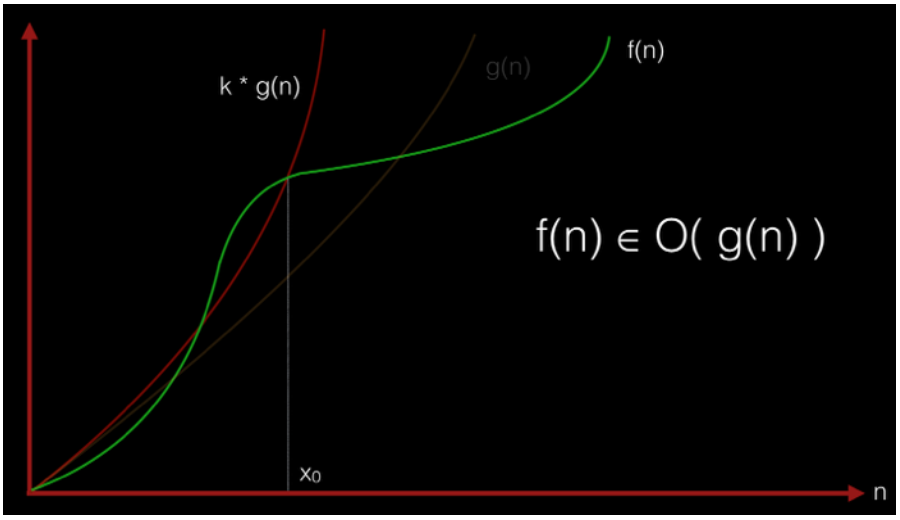





```
MERGE-SORT (array A of n numbers)
for i = 1; i < n; i = 2i
      for j = 0; j < (n – i); j = j + 2i
            Merge(A[j], i, 2i)


MERGE (A, end1, end2)
i = 0, j = end1, k = 0, temp = []

while i < end1 AND j < end2
      if (A[i] < A[j])
            temp[k] = A[i]
            i = i + 1, k = k + 1
      else
            temp[k] = A[j]
            j = j + 1, k = k + 1

while i < end1
      temp[k] = A[i]
      i = i + 1, k = k + 1

while j < end2
      temp[k] = A[j]
      j = j + 1, k = k + 1

for (i = 0; i < end2; i = i + 1)
      A[i] = temp[i]
```

Big O

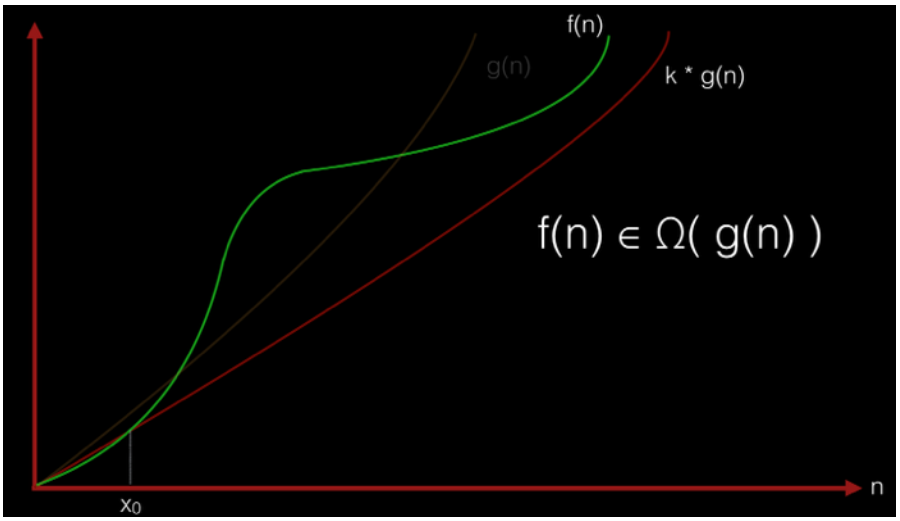
- Нас волнует масштабируемость
- Нужно описать увеличение работы с увеличением информации
- Будем использовать функции от n



$f(n) \in O(g(n))$ если есть такие $k, n_0 > 0$, что

$f(n) \le k*g(n)$

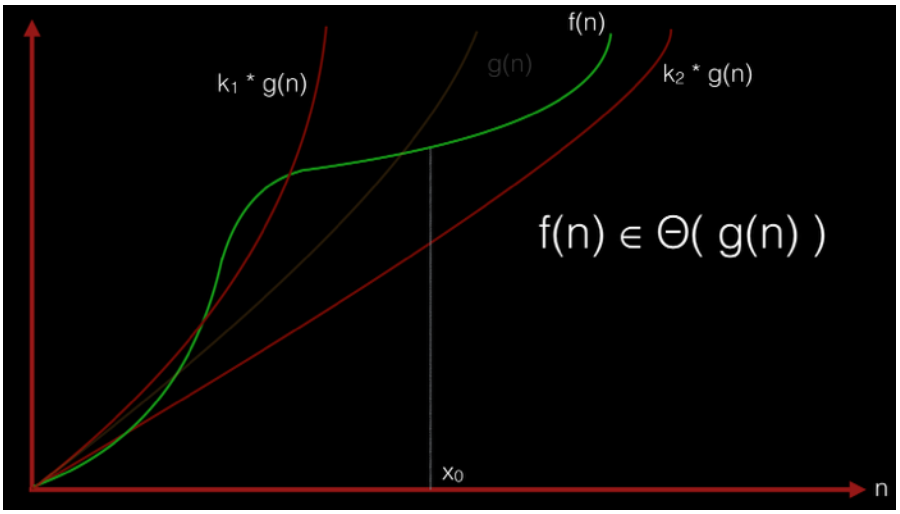
для всех $n > n_0$



$f(n)$ Ω $O(g(n))$ если есть такие $k, n_0 > 0$, что

$f(n) \ge k*g(n)$

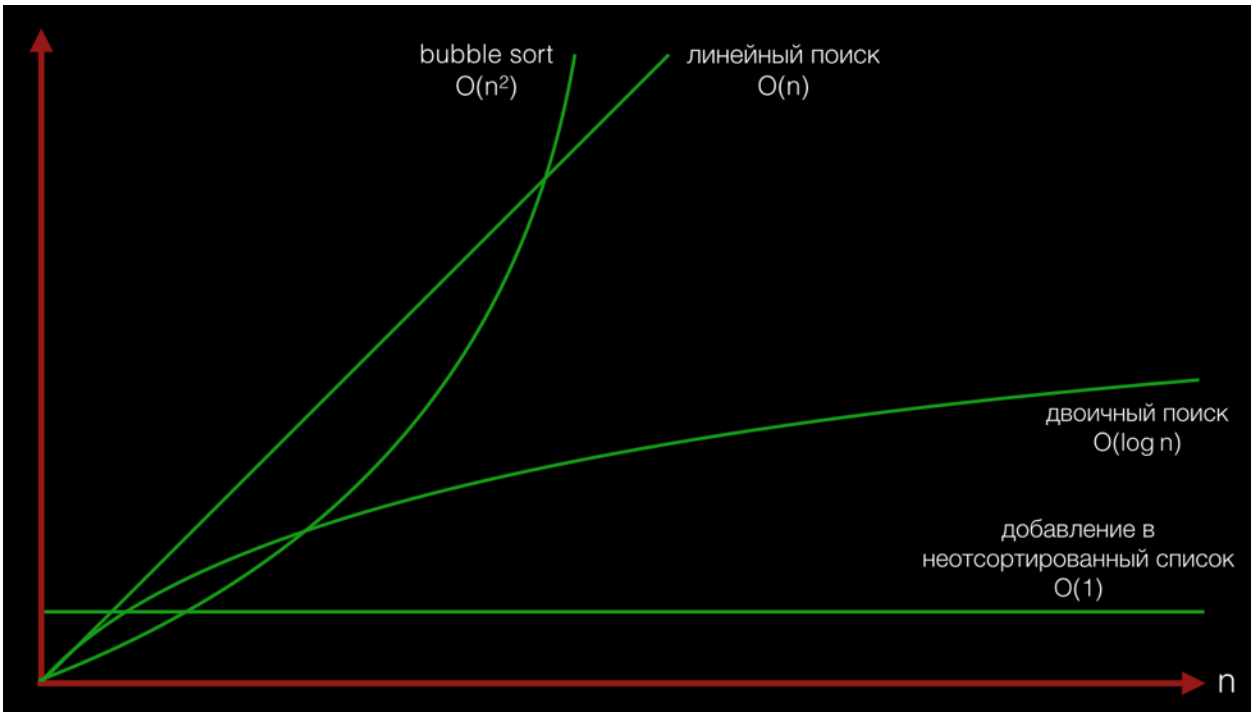
для всех $n > n_0$



$f(n)$ Ω $\Theta(g(n))$ если есть такие $k_1, k_2, n_0 > 0$, что

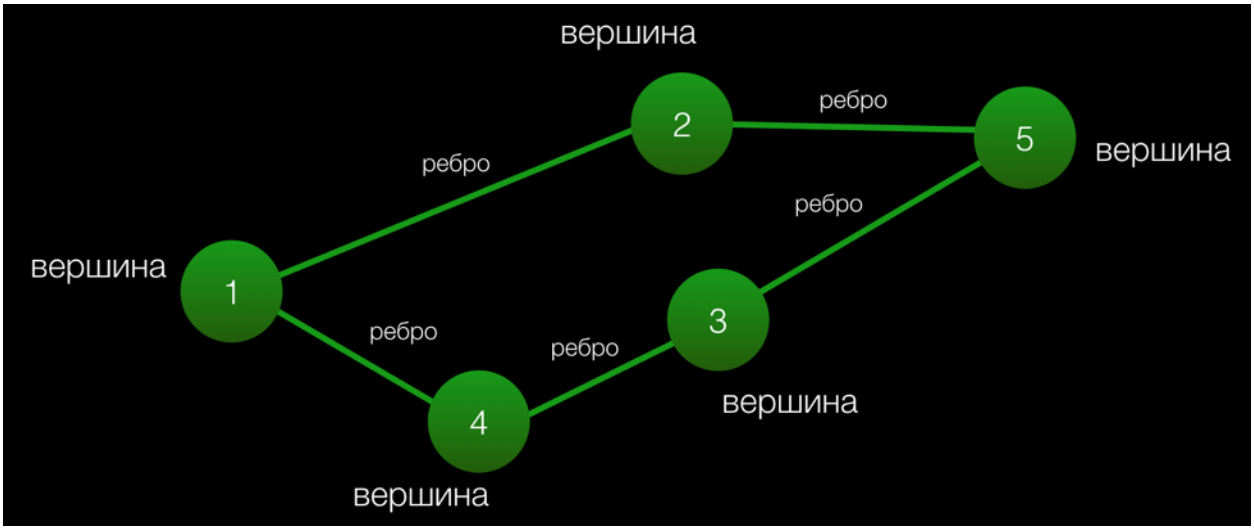
$k_1*g(n) \ge f(n) \ge k_2*g(n)$

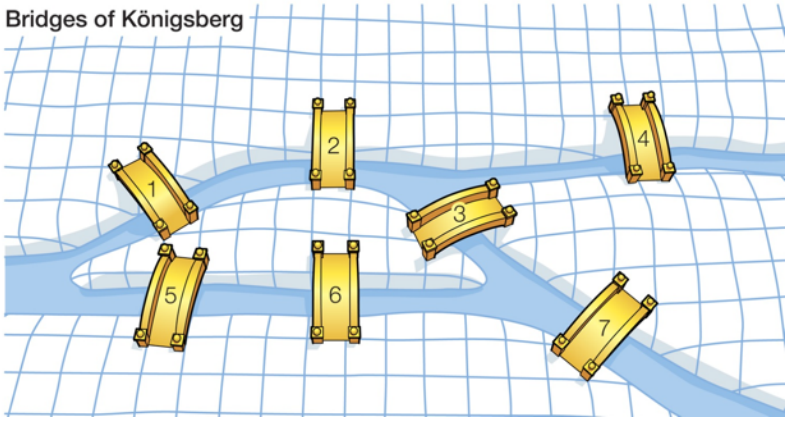
для всех $n > n_0$



- Big O не позволяет предугадать или оценить производительность алгоритмов
- Big O определяет границу роста
- Big O позволяет классифицировать алгоритмы

Графы



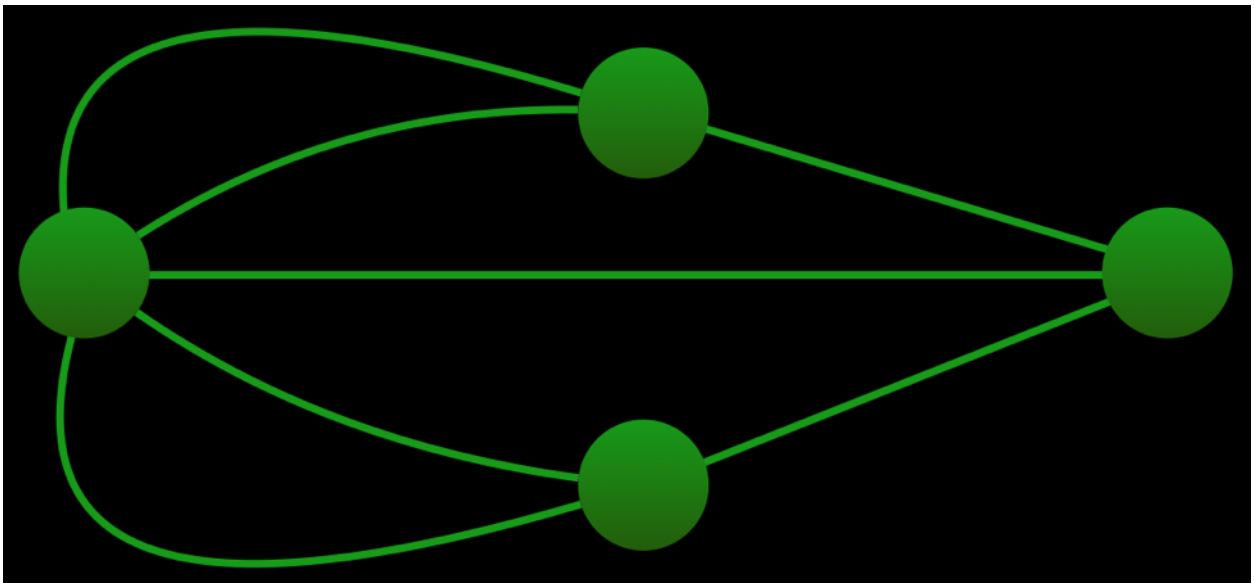


Проблема семи мостов Кенигсберга

Как можно пройти по всем семи мостам Кёнигсберга, не проходя ни по одному из них дважды?

Леонард Эйлер решил задачу в 1736 г. с помощью созданной им теорией графов.

Конструкцию можно смоделировать с помощью графа:

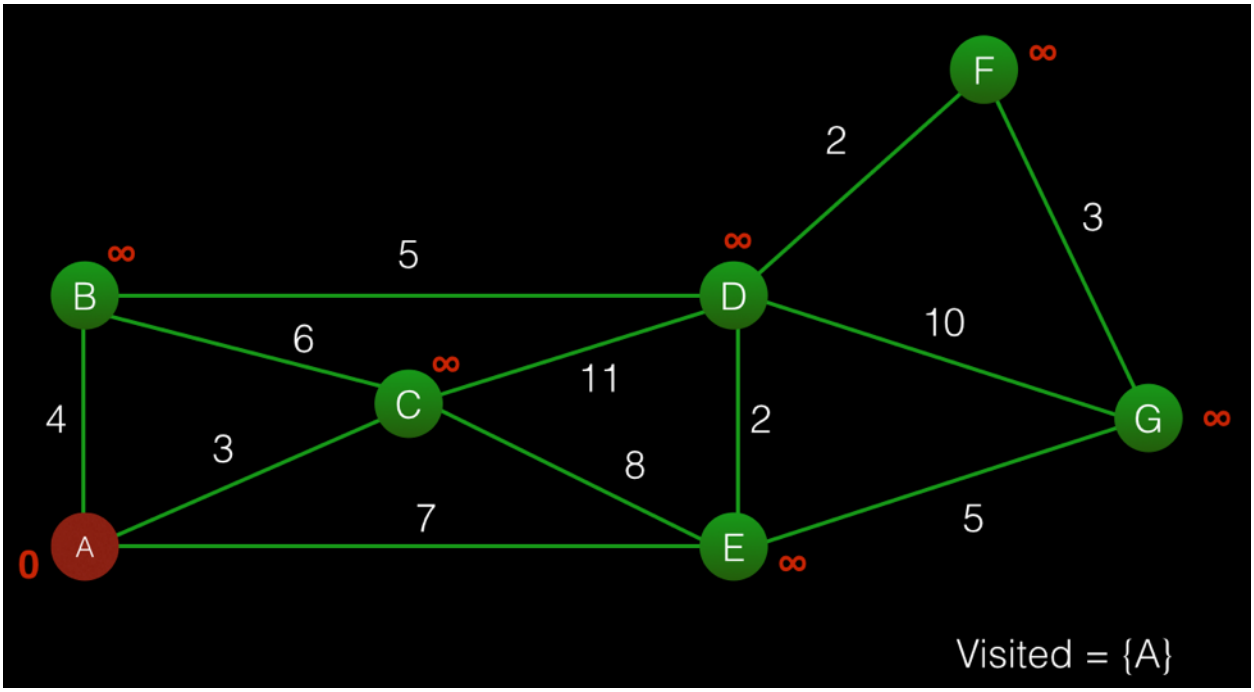


Эйлер пришел к следующим выводам:

- Число нечётных вершин (вершин, к которым ведёт нечётное число рёбер) графа должно быть чётно. Не может существовать граф, который имел бы нечётное число нечётных вершин.
- Если все вершины графа чётные, то можно, не отрывая карандаша от бумаги, начертить граф, при этом можно начинать с любой вершины графа и завершить его в той же вершине.
- Граф с более чем двумя нечётными вершинами невозможно начертить одним росчерком.

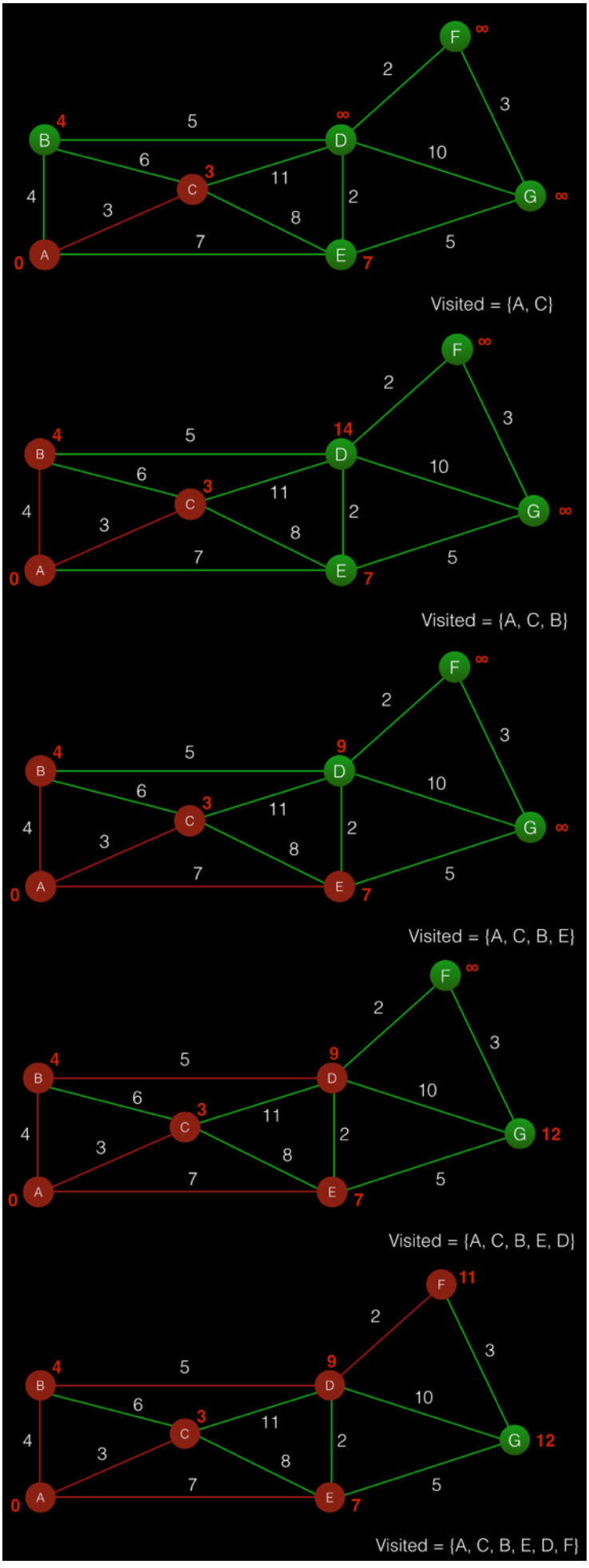
Так как в графе, моделирующем семь мостов Кенигсберга, четыре нечетные вершины, то **невозможно** пройти по всем мостам не проходя ни по одному из них дважды.

Алгоритм Дейкстры
для нахожления кратчайшего пути между двумя точками на взвешенном графе.



Нужно найти кратчайший (самый дешевый) путь от точки A до точки F.
Начинаем в точке A. Каждой вершине приписываем цену. Пока мы не посетили ни одну вершину, всем приписываем бесконечную цену. В памяти держим множество посещенных вершин.

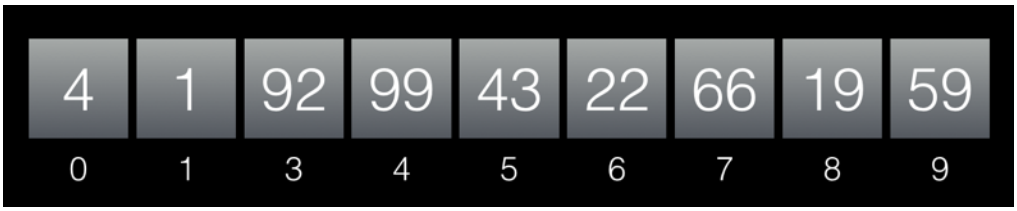
Считаем цену движения к каждой из ближайших не посещенных вершин (B, C, E). Сравниваем подсчитанную цену с уже записанной для этих вершин. Везде стоит бесконечная цена, поэтому заменяем ее на новую (4, 3 и 7, соответственно). Выбираем самую дешевую вершину и переходим к ней. Повторяем.


F ∞
2
3
4
5
∞
B
D
6
10
3
11
C
4
2
G ∞
3
8
7
5
0
A
E 7
Visited = {A, C}
F ∞
2
3
4
5
14
B
D
6
10
3
11
C
4
2
G ∞
3
8
7
5
0
A
E 7
Visited = {A, C, B}
F ∞
2
3
4
5
9
B
D
6
10
3
11
C
4
2
G ∞
3
8
7
5
0
A
E 7
Visited = {A, C, B, E}
F ∞
2
3
4
5
9
B
D
6
10
3
11
C
4
2
G 12
3
8
7
5
0
A
E 7
Visited = {A, C, B, E, D}
F 11
2
3
4
5
9
B
D
6
10
3
11
C
4
2
G 12
3
8
7
5
0
A
E 7
Visited = {A, C, B, E, D, F}

Урок 3

Структуры данных.

Массив
- упорядоченный набор элементов
- доступ к элементам – по индексу



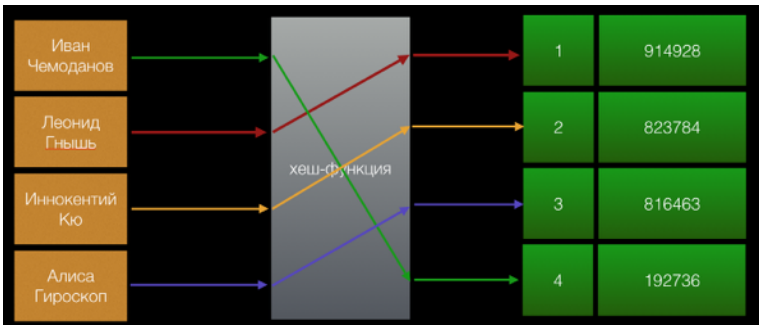
Добавление элемента: O(1)
Поиск элемента: O(n)
Удаление элемента: O(n)

Отсортированный массив
Добавление элемента: O(n)
Поиск элемента: O(log n) *двоичный поиск*
Удаление элемента: O(n)

Ассоциативный массив
Коллекция пар “ключ -> значение”. Операции:
- INSERT (key, value)
- FIND (key)
- REMOVE (key)

Примеры:
- штрих коды: код (ключ) соответствует информации о товаре (значение)
- паспорт: серийный номер паспорта (ключ) соответствует записи в базе данных граждан (значение)

Хеш-таблица
Реализация ассоциативного массива. Хеш-функция генерирует индекс массива по ключу.



Добавление элемента: O(1)
Поиск элемента: O(1)
Удаление элемента: O(1)

Распределение ключей с помощью хеш-функции
Что если использовать простой генератор случайных чисел и распределять ключи случайно?

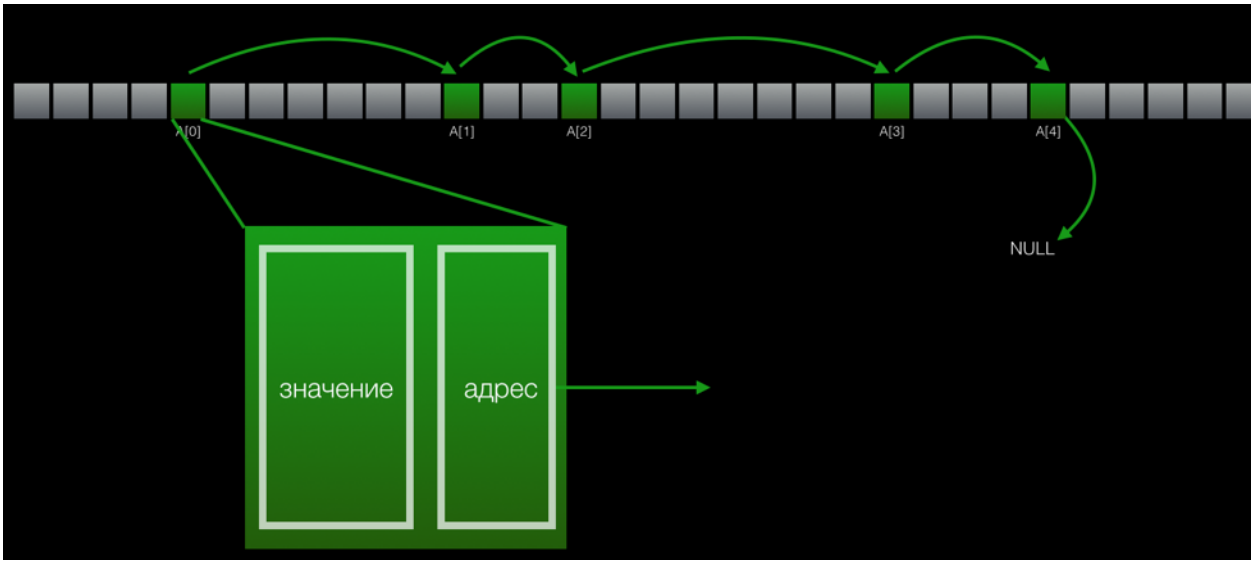
Пример: 2500 элементов и 1 000 000 ячеек. При случайном распределении вероятность что два элемента окажутся в одной ячейке = 95% (см. парадокс дней рождения https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%BE%D0%BA%D1%81_%D0%B4%D0%BD%D0%B5%D0%B9_%D1%80%D0%BE%D0%B6%D0%B4%D0%B5%D0%BD%D0%B8%D1%8F)

Мы хотим минимизировать совпадения (столкновения или коллизии). Нам нужна хорошая хеш-функция.

Хорошая хеш-функция
- равномерное распределение ключей для минимизации коллизий
- для некоторых задач: нужно избегать кластеризации (скопления занятых ячеек)

Идеальная хеш-функция
- нет коллизий вообще
- поиск за O(1) в худшем случае

Связный список
Вместо линейного использования памяти мы храним элементы распределенно, при этом каждый элемент должен хранить ссылку (адрес) на следующий элемент.

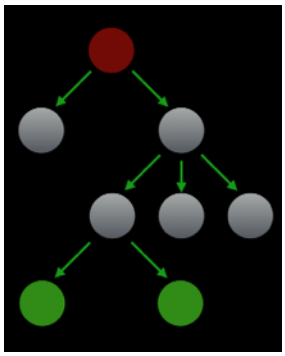


Добавление элемента: O(1)
Поиск элемента: O(n)
Удаление элемента: O(n)

Плюсы и минусы связных списков
+ нет проблем с памятью как у массивов
+ простое добавление/удаление
- сложность поиска
- дополнительные затраты на указатели

Проблема медленного поиска в связном списке
- “move-to-front heuristic” – перемещать найденный элемент в начало списка
- индексы: дополнительная структура данных с указателями

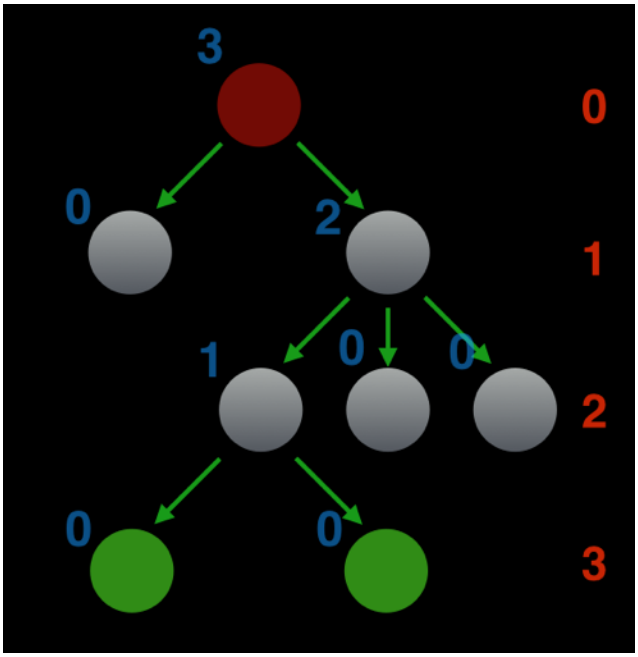
Деревья
- Дерево (tree) – набор связанных узлов
- Узел (node) – содержит данные и указатели на своих потомков



- Корень (root) – узел без родителя
- Внутренний узел – имеет как минимум одного потомка
- Листовой узел (leaf) – узел, не имеющий потомков

Дерево – рекурсивная структура данных.

- N узлов -> N-1 указателей (стрелок, ребер)
- Глубина (**depth**) узла – длина пути от корня до узла (количество ребер)
- Высота (**height**) узла – количество ребер в самом длинном пути от узла до листового узла
- Высота дерева = высота корня

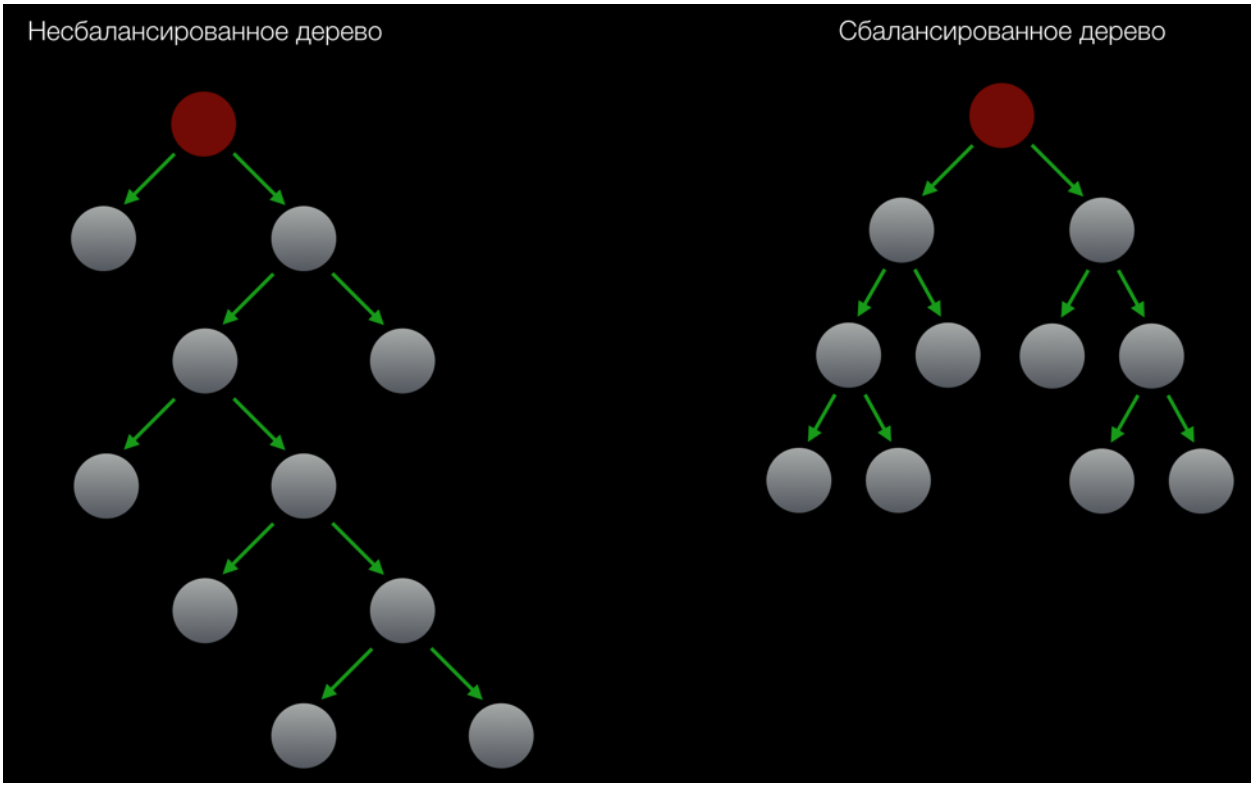


Двоичное дерево
У каждого узла не более двух потомков

- Strictly binary tree (full binary tree) – все узлы кроме листовых имеют ровно два потомка
- Complete binary tree – все узлы кроме листовых заполнены, дерево растет слева

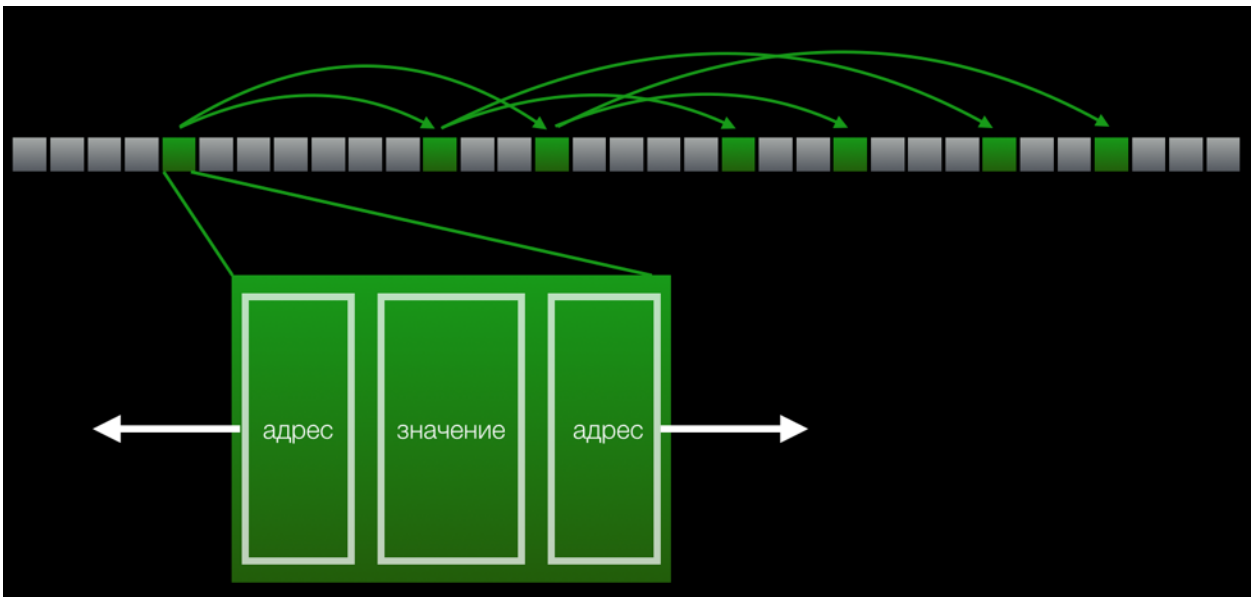
- Максимальное количество узлов на уровне i это = 2^i
- Максимальное количество узлов в дереве высотой h это $2^0+2^1+\ldots+2^h$ = **2^h+1-1**
- Высота **h = log(n+1)-1**

Мысли:
- Для большинства операций важным параметром является высота дерева
- Высоту можно минимизировать если делать дерево “плотным”, заполнять его максимально
- Сбалансированное дерево – разница в высоте между правым и левым поддеревом не превышает k (обычно k=1)



Как хранить двоичные деревья в памяти?

1. Аналогично связным спискам, использовать ссылки (указатели) на потомков.



2. Хранение в массиве (для full binary tree)

Для узла по индексу i:
индекс левого потомка – 2i+1
индекс правого потомка – 2i+2

Двоичное дерево поиска (binary search tree; BST)
Все узлы слева – меньше, все узлы справа – больше

Двоичный поиск: O(log n)
Добавление: O(log n)
Удаление: O(log n)

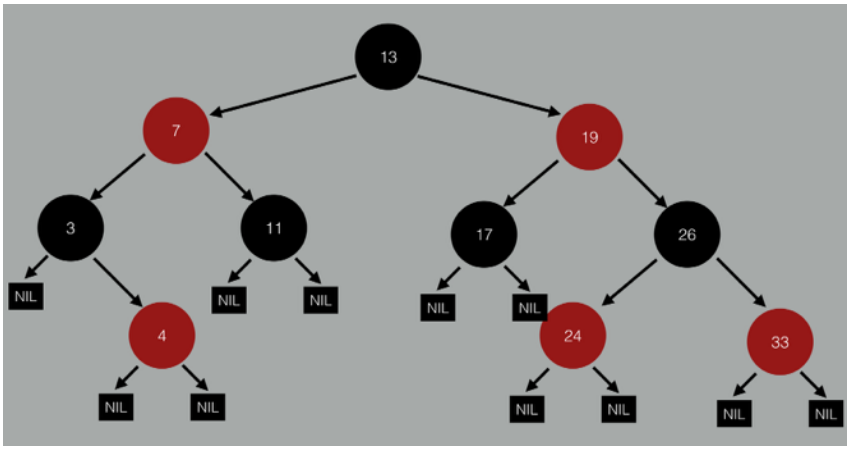
В худшем случае:
Двоичный поиск: O(n)
Добавление: O(n)
Удаление: O(n)

	Неотсортированый массив	Связный список	Отсортированный массив	Двоичное дерево поиска
Поиск	**O(n)**	**O(n)**	**O(log n)**	**O(log n)**
Добавление	**O(1)**	**O(1)**	**O(n)**	**O(log n)**
Удаление	**O(n)**	**O(n)**	**O(n)**	**O(log n)**

Balanced search tree
- Условие: высота всегда должна быть O(log n)
- В этом случае все операции будут O(log n) и в среднем и в худшем случае
- Необходимо: всегда поддерживать дерево в сбалансированном виде

Red-Black tree
Само-балансирующееся двоичное дерево поиска (self-balancing binary search tree)

1. Один дополнительный бит в каждом узле для указания цвета (красный / черный).
2. Корень должен быть черным
3. У красного узла должны быть черные потомки
4. Любой путь от корня до конца должен содержать одинаковое число черных узлов.

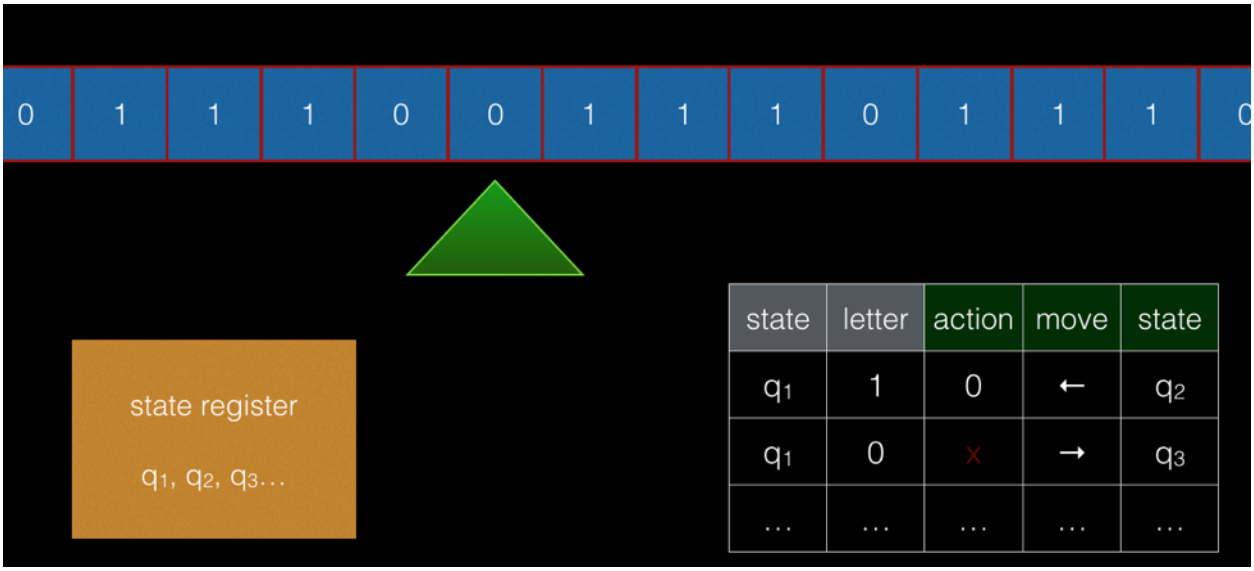


Урок 3

P vs NP

Машина Тьюринга – абстрактная вычислительная машина, состоящая из четырех элементов:

- Бесконечная лента с ячейками, в которые могут быть записаны символы конечного алфавита (например, 1 и 0 или буквы английского алфавита). Одна ячейка = один символ. Ячейка может быть пустой (обычно для пустоты используется специальный символ)
- Головка, которая может считывать символы из ячеек и записывать символы в ячейки. Головка может двигаться влево и вправо на одну ячейку за один раз.
- Регистр состояний, в котором хранятся “состояния” машины.
- Таблица правил, в которой описаны действия машины при определенном состоянии и текущем символе.



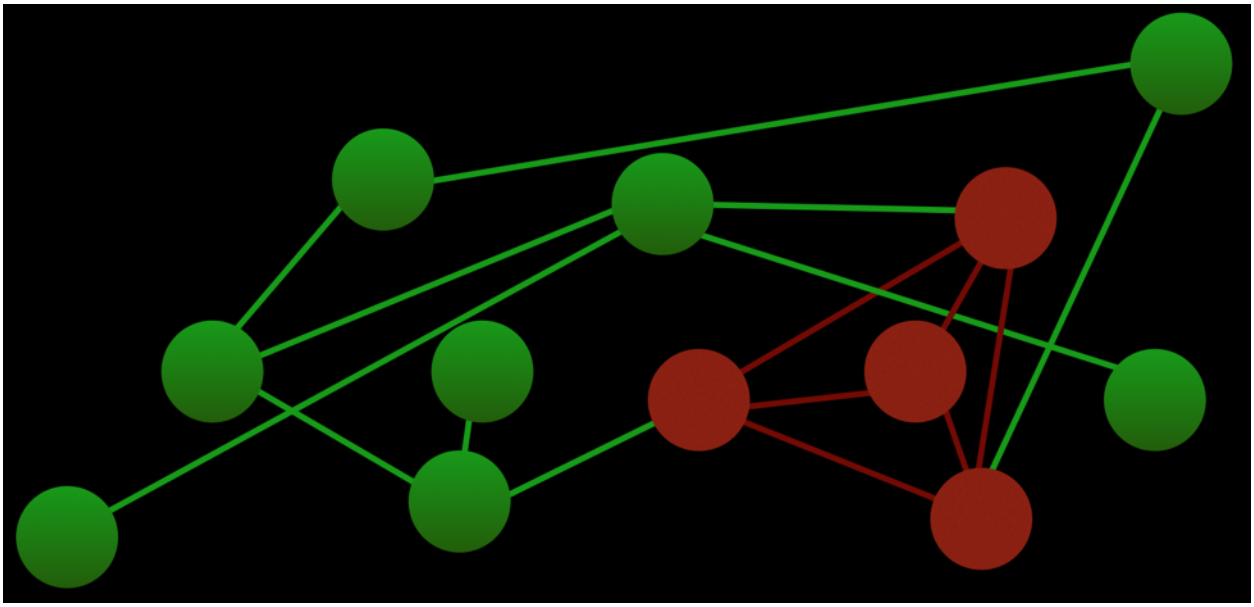
Это – **детерменированная машина Тьюринга**. На каждой “развилке” она может пойти только по одной из дорог. То есть, для каждой комбинации “состояние + символ” машина может использовать только одно правило из таблицы.

Недетерминированная машина Тьюринга это машина с бесконечной параллелизацией. На развилках она может идти одновременно по нескольким дорогам. Такой машины в реальности не существует (современные компьютеры имеют конечное число ядер или процессоров, а недетерминированная машина Тьюринга аналогична компьютеру с бесконечным числом ядер).

Важный вопрос: **Если решение проблемы можно быстро проверить, означает ли это что проблему можно быстро решить?**

Что такое быстрое решение? Это решение, которое можно найти за полиномиальное время на детерминированной машине Тьюринга. Все задачи, которые мы рассматривали до этого (сортировка, двоичный поиск, кратчайший путь в графе) – это простые задачи, их можно решить за полиномиальное время $O(n^k)$. Мы называем класс таких задач **P** (от polynomial).

Рассмотрим другую задачу: clique problem. Найти полностью соединенный подграф с максимальным количеством вершин.



Или другая задача: факторизация. Дано число N. Нужно найти такие числа X и Y, которые при умножении дадут N.

Это сложные задачи и единственный известный вариант решения это перебор всех (или почти всех) вариантов. Из-за особенностей задачи, количество вариантов растет экспоненциально, и в какой-то момент перебор оказывается настолько сложным, что может занять миллионы лет на современных компьютерах.

Особенность этих задач в том, что если нам дано решение и нужно проверить его корректность, то мы можем сделать это быстро, за полиномиальное время.

Такие задачи (полиномиальное время проверки решения на детерминированной машине Тьюринга, полиномиальное время поиска решения на недетерминированной машине Тьюринга) мы относим к классу **NP** (nondeterministic polynomial).

Почему решение таких задач является сложным? Потому что оно сводится к поиску в огромном множестве (brute force).

Вопрос “P=NP?” аналогичен вопросам:

- Если решение проблемы можно быстро проверить, означает ли это что проблему можно быстро решить?
- Можно ли решить задачу поиска без поиска?

Если P=NP, то

- Если задачу можно решить за полиномиальное время на недетерминированной машине Тьюринга, то ее можно решить за полиномиальное время на детерминированной машине Тьюринга
- Полный поиск не обязателен (иголку в стоге сена можно найти с помощью магнита, а не перебором всего стога)

Если P≠NP, то

- Если задачу можно решить за полиномиальное время на недетерминированной машине Тьюринга, то ее нельзя решить за полиномиальное время на детерминированной машине Тьюринга
- Полный поиск обязателен (волшебного магнита не существует)

Вопрос равенства P и NP относится к самым важным задачам современной математики.

Задачи тысячелетия
- Равенство классов P и NP
- Гипотеза Ходжа
- Гипотеза Пуанкаре (доказана)
- Гипотеза Римана
- Теория Янга — Миллса
- Существование и гладкость решений уравнений Навье — Стокса
- Гипотеза Бёрча — Свиннертон-Дайера

NP-полная задача
задача, к которой можно привести любую другую NP-задачу за полиномиальное время. Любую задачу из класса NP можно преобразовать в любую задачу класса NP-complete. Это означает, что если решить одну любую NP-complete задачу, то это автоматически решит все задачи класса NP.

Что изучать дальше?

- Математический аппарат для анализа алгоритмов
  - дискретная математика, алгебра, комбинаторика, теория чисел, теория графов
  - Concrete Mathematics: A Foundation for Computer Science, R. L. Graham, D. E. Knuth, O. Patashnik
- Вероятностный анализ и рандомизация
- Динамическое программирование
- Жадные алгоритмы
- Сложные структуры данных
- Многопоточные алгоритмы
- Вычислительная геометрия

Рекомендуемые книги

- Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (http://www.amazon.com/Introduction-Algorithms-Thomas-H-Cormen/dp/0262033844)
- Art of Computer Programming, Donald Ervin Knuth (http://www.amazon.com/dp/0201485419/)
- The Algorithm Design Manual,Steven S. Skiena (http://www.amazon.com/Algorithm-Design-Manual-Steven-Skiena/dp/1848000693)

Пожалуйста, пройдите опрос после завершения курса

http://simpoll.ru/run/survey/45697da8