Первый курс, весенний семестр 2015/16
Конспект лекций по алгоритмам

Собрано 24 июня 2016 г. в 02:55

Содержание

1. Двоичное дерево поиска 1
- 1.1. Хранение . . . 1
- 1.2. Добавление . . . 2
- 1.3. Операции . . . 2
- 1.4. Ускорение операций . . . 3
- 1.5. Удаление . . . 4
- 1.6. Обходы дерева . . . 4
- 1.7. Способы вывода дерева . . . 5
- 1.8. Set, Multiset и Map . . . 5

2. AVL-дерево 5
- 2.1. Опеределение и основные свойства . . . 6
- 2.2. Добавление . . . 6

3. Удаление из AVL и Неявный Ключ 8
- 3.1. Удаление из AVL дерева . . . 9
- 3.2. Implicit Key . . . 9
- 3.3. Случайное BST-дерево . . . 10

4. Декартово дерево 11
- 4.1. Декартово дерево (cartesian tree) . . . 11
- 4.2. Декартово дерево (treap) . . . 11
- 4.3. О равных ключах . . . 12
- 4.4. Split . . . 12
- 4.5. Merge . . . 12
- 4.6. Медленные Add и Delete . . . 13
- 4.7. Быстрые Add и Delete . . . 13
- 4.8. Неявный ключ . . . 14
- 4.9. Про Add для вещественных чисел . . . 14
- 4.10. Приоритеты не нужны . . . 14
- 4.11. Время работы . . . 14

5. Red-Black, AA, 2-3, 2-3-4, B+ Trees 14
- 5.1. 2-3 Tree . . . 15
- 5.2. AA tree . . . 18
- 5.3. 2-3-4 Tree . . . 19
- 5.4. Red-Black Tree . . . 19

5.5. B+ Tree . . . 20

6. Splay Tree **22**
6.1. Операции . . . 22
6.1.1. Zig-zig вращение . . . 22
6.1.2. Zig-zag вращение . . . 22
6.2. merge и split . . . 23
6.3. Асимптотика . . . 23
6.3.1. Бор . . . 24
6.3.2. 2-3-tree . . . 25

7. Дерево отрезков. **26**
7.1. Еще немножко о деревьях поиска. . . . 27
7.2. Самые обычные деревья отрезков. . . . 27
7.3 Динамическое ДО. . . . 29
7.4 Персистентное ДО. . . . 29
7.5 Многомерные ДО. . . . 29

8. Дерево отрезков **30**
8.1. ДО сортированных массивов . . . 31
8.1.1. Задача 1 . . . 31
8.2. Scanline . . . 31
8.2.1. Задача 2 . . . 31
8.2.2. Другое решение задачи 1 . . . 31
8.3. К-порядковая статистика за $\mathcal{O}(n \log n)$ . . . 32

9. Rope **33**
9.1. Определение . . . 33
9.2. Через Splay . . . 33
9.2.1. Split . . . 33
9.2.2. Merge . . . 33
9.3. Skip-List . . . 33
9.3.1. Find . . . 34
9.3.2. Add . . . 34
9.3.3. Delete . . . 34
9.3.4. Split . . . 34
9.3.5. Merge . . . 34
9.4. Offline персистентность . . . 35
9.5. Persistent RBST . . . 36

10. Фарах-Колтон и Бендер **36**
10.1. Что уже есть . . . 37
10.2. Sparse Table . . . 37
10.2.1. Super Sparse . . . 37
10.3. алгоритм Фараха-Колтона и Бендера . . . 38
10.3.1. Анонс . . . 38
10.3.2. RMQ $\to$ LCA . . . 38

10.3.3. LCA → RMQ±1 . . . 39
10.3.4. RMQ±1 . . . 39

11. RMQ ±1, LCA **39**
11.1. RMQ ±1 . . . 40
11.2. LCA . . . 40

12. LA **42**
12.1. LCA-Offline. Тарьян . . . 42
12.2. LA (Level Ancestor) . . . 42
12.3. Ladder-Decomposition . . . 42

13. Euler-tour/HLD/Link-cut **44**
13.1. Euler-Tour-Tree . . . 44
13.2. HLD – Heavy Light Decomposition . . . 44
13.3. Link-Cut Tree (без доказательства времени) . . . 44

14. RMQ и LCA **46**
14.1. RMQ offline. . . . 46
14.2. Сумма на пути дерева. . . . 46
14.3. Рандомизированный MST за линейное время. . . . 47

15. Минимум на пути в дереве **48**
15.1. Постановка задачи . . . 48
15.2. Обзор существующих медленных решений . . . 48
15.2.1 Centroid decomposition . . . 48
15.2.2 Двоичные подъёмы . . . 48
15.2.3 dfs с деревом отрезков . . . 48
15.3. Алгоритм с СНМ . . . 48

16. Минимальное паросочетание **49**
16.1. Определения, постановка задачи . . . 50
16.2. Двудольные графы . . . 50
16.2.1. Алгоритм поиска ДЧП . . . 51
16.2.2. Алгоритм Куна . . . 51

17. Max independent set, min cover **52**
17.1. Алгоритм за $\mathcal{O}(V+E)$ . . . 52
17.2. Корректность алгоритма . . . 52
17.3. Всяческие ускорения алгоритма Куна . . . 53

18. Обобщение Куна на произвольные графы **54**
18.1. Матрица Татта . . . 54
18.2. Алгоритм поиска совершенного паросочетания: . . . 54
18.3. Upgrade Куна почти всегда работающего на произвольном графе: . . . 54
18.4. Задача: Класифицировать рёбра на обязательно/возможно лежащие в max паросочетании для двудольного графа: . . . 54

19. Marriage Problem, покраска рёбер графа **56**
19.1. Marriage Problem (stable matching) . . . 56
19.2. Технические моменты . . . 56
19.3. Алгоритм . . . 56
19.4. Доказательство корректности . . . 57
19.5. Замечания . . . 57
19.6. Покраска рёбер графа . . . 57
19.7. Двудольные графы . . . 57

20. Раскараска рёбер двудольных графов **59**
20.1. Вспомним прошлую лекцию . . . 59
20.2. Паросочетание, покрывающее вершины степени D . . . 59
20.3. Покраска рёбер за $\mathcal{O}(E^2)$ . . . 59

21. Вершинные покраски **61**
21.1. Теорема Брукса . . . 61
21.2. Алгоритм покраски вершин . . . 61
21.3. Интересный факт про планарые графы . . . 61
21.4. Покраска планарного графа в 6 цветов . . . 61
21.5. Покраска планарного графа в 5 цветов . . . 62

После коллоквиума **62**

22. Венгерский алгоритм **63**
22.1. Задача и способы ее решения . . . 63
22.2. Идея венегерского алгоритма . . . 63
22.3. Псевдокод алгоритма . . . 64
22.4. Оптимизация до $\mathcal{O}(V^3)$ . . . 64

23. Потоки **65**
23.1. Основные определения . . . 65
23.2. Задача о максимальном потоке, теорема Форда-Фалкерсона . . . 68
23.3. Алгоритм Форда-Фалкерсона . . . 70
23.4. Существование максимального потока в произвольной сети. Алгоритм Эдмондса-Карпа . . . 71
23.5. Декомпозиция потока . . . 72

24. Сложные потоки **73**
24.1. Вспомнить все . . . 73
24.1.1. Разность потоков одинакового размера . . . 73
24.1.2. Эдмондс-Карп . . . 73
24.2. Масштабирование потока . . . 73
24.3. Алгоритм Диница . . . 73
24.3.1. +Scaling . . . 74
24.4. Хопкрофт-Карп . . . 74

25. Первая Теорема Карзанова. **75**

26. Потоки: LR. 76
26.1. Что хотим? 76
26.2. Понемногу двигаемся к цели 76
26.3. Максимизируем 76
26.4. Не забываем старые знания. 77

27. Минимальный глобальный разрез и Диниц 78
27.1. Диниц с Link-cut tree 78
27.2. Каргер-Штейн 78
27.2.1. $\mathcal{O}(n^4 \cdot \log C)$ 78
27.2.2. $\mathcal{O}(n^2 \cdot \log^2 n \cdot \log C)$ 79
27.3. Штор-Вагнер 79

28. Mincost flow 81
28.1. 4(четыре) задачи 81
28.2. Алгоритм Клейна для mincost circulation 81
28.3. Алгоритм через доп.пути 81

29. MincostFlow 81
29.1. Алгоритм первый, простейший. 82
29.2. Алгоритм второй, бинпоиск по ответу. 82
29.3. Первый быстрый алгоритм. 82
29.4. Второй быстрый алгоритм. Capacity Scaling. 82
29.5. Ускорение с помощью Дейкстры с потенциалами. 83

30. Строки. Поиск подстроки в строке. 84
30.1. Обозначения 84
30.2. C++. 84
30.3. КМП. 84
30.4. Z-функция. 85

31. Алгоритм Бойера-Мура, Хеширование 87
31.1. Алгоритм Бойера-Мура 87
31.2. Хеширование 89

32. Хеширование 91
32.1. Хеши, теория 91
32.2. Что есть в C++ 91
32.3. Строка Туэ-Морса 91
32.4. Алгоритм Капуна 92

33. Применение хешей 95
33.1. Алгоритм Рабина-Карпа 95
33.2. Сравение строк, LCP 96
33.3. Суффиксный массив 96
33.4. Применение суффиксного массива 96
33.5. Наибольшая общая подстрока двух строк 97

34. Суффиксный массив за $\mathcal{O}(n \log n)$ и LCP 98
34.1. Суффиксный массив за $\mathcal{O}(n \log n)$ . . . 98
34.2. Суффиксный массив и LCP . . . 98

35. Суффиксный массив: продолжение 100
35.1. Алгоритм Каркайнена-Сандерса ($\mathcal{O}(n)$) . . . 100
35.1.1. . . . 100
35.1.2. . . . 100
35.1.3. . . . 100
35.1.4. . . . 100
35.1.5. . . . 100
35.2. Поиск строки в тексте . . . 100
35.2.1. . . . 100
35.2.2. . . . 100

36. Алгоритм Ахо-Корасик 101
36.1. Бор . . . 102
36.2. Суффиксная ссылка . . . 102
36.3. Алгоритм Ахо-Корасик . . . 102
36.3.1. Способ первый - как префикс-функция . . . 102
36.3.2. Способ второй - полный автомат . . . 104

37. Суффиксное дерево. Алгоритм Укконена. 105
37.1. Суффиксный бор, дерево . . . 105
37.2. Элементарный алгоритм построения . . . 105
37.3. Алгоритм Укконена . . . 105
37.3.1. Реализация алгоритма Укконена за $\mathcal{O}(n^2)$ . . . 106
37.4. Улучшение до $\mathcal{O}(n)$ . . . 106

38. Хеширование 107
38.1. Введение . . . 107
38.2. Хеш-таблица . . . 107
38.3. Хеширование Кукушки . . . 107
38.4. Perfect Hashing . . . 108

39. Игры 109
39.1. Симметричные игры на графе . . . 109
39.1.1. Решение для ациклического орграфа . . . 109
39.1.2. Решение с циклами (ретроанализ) . . . 109
39.1.3. Прямая сумма игр . . . 110
39.1.4. Ним . . . 110

40. Быстрое проеобразование Фурье 111
40.1. Немного о многочленах . . . 111
40.2. Идея . . . 111
40.3. Мотивация . . . 111
40.4. Быстрое преобразование Фурье . . . 111

40.5. Медленная реализация БПФ . . . 112

41. Действительно Быстрое проеобразование Фурье 113
41.1. Фурье обращается просто . . . 113
41.2. Фурье погрешен . . . 113
41.3. Коротко о длинных числах . . . 114
41.4. Фурье оптимизируется и избавляется от ненужной рекурсии . . . 114
41.4.1. $2\mathbb{R} = 1\mathbb{C}$ . . . 114
41.4.2. cos и sin — долгая штука . . . 115
41.4.3. Нерекурсивная реализация . . . 115

42. Длинная арифметика 116
42.1. Простые операции . . . 116
42.2. Умножение длинных чисел . . . 116
42.3. Деление длинных чисел за $\mathcal{O}(nm)$ . . . 117
42.4. Двоичная арифметика . . . 118

43. Деление длинных чисел 120
43.1. Двоичный поиск по частному . . . 120
43.2. Двоичный поиск по цифре . . . 120
43.3. Деление за честный квадрат . . . 120
43.4. Деление за $\mathcal{O}(n \log^2 n)$ . . . 121

44. Классы сложности 122
44.1. Типы задач . . . 122
44.2. Классы сложности задач . . . 122
44.3. Сравнение задач . . . 122
44.4. Классы сложности задач (продолжение) . . . 123

45. Классы сложности 124
45.1. Примеры задач из разных классов . . . 124
45.1.1. SAT . . . 124
45.1.2. CNFSAT . . . 124
45.1.3. 3-CNFSAT . . . 124
45.1.4. Independent set . . . 125
45.1.5. Halting problem . . . 125
45.1.6. Другие задачи . . . 126

46. Теория чисел 126
46.1. Расширенный алгоритм Евклида . . . 127
46.2. Обратные в кольце по модулю . . . 127
46.3. Возведение в степень за $\mathcal{O}(\log n)$ . . . 128
46.4. Обратные для чисел от 1 до $k - 1$ по модулю за $\mathcal{O}(k)$ . . . 128
46.5. RSA . . . 128
46.6. Первообразный корень . . . 129

47. Теория чисел **129**
47.1. Китайская теорема об остатках . . . 130
47.2. Решето Эратосфена . . . 130
47.3. Вычисление мультипликативных функций . . . 131
47.4. Простые числа . . . 131
47.5. Факторизация . . . 132

48. Окончание лекции по теории чисел **134**
48.1. 3 задачи про разные степени . . . 134
48.2. Решения: . . . 134

49. Вероятностные алгоритмы **136**
49.1. Фильтр Блюма . . . 136
49.2. Вероятностные алгоритмы . . . 136
49.3. Понижение ошибки и прочее . . . 137

Лекция по алгоритмам #1
Двоичное дерево поиска

9 февраля

Def 1.0.1. *Двоичное дерево поиска, binary search tree (BST) – бинарное дерево, у которого оба поддерева вершины являются BST, причём значения всех вершин левого поддерева меньше значения вершины, а значения всех вершин правого поддерева больше значения вершины.*

В зависимости от реализации каждая вершина может иметь не более двух или ровно двух (с помощью добавления фиктивных вершин) детей.

Def 1.0.2. *Глубина вершины – расстояние до корня*

Def 1.0.3. *Глубина дерева – максимальная глубина по всем вершинам*

1.1. Хранение

Двоичные деревья поиска можно хранить:

1. **Как бинарные деревья:**

```
struct Node {
  Node *l, *r, *parent;
  int x;
}
```

   Хранить ссылку на отца не обязательно.

2. **Способом для произвольных деревьев:**

```
struct Node {
  Node *child, *next;
  int x;
}
```

   Дети каждой вершины хранятся списком. child указывает на первого из детей, next – на следующего брата.

3. **Через отцов:**

```
struct Node {
  Node *parent;
  int x;
}
```

   Такой способ хорош тем, что позволяет легко генерировать случайные деревья. Сделаем корнем вершину с номером 0, каждую следующую будем подвешивать к случайной уже добавленной. $p_0 = 0$ – корень, $0 \le p_i < i$ – все остальные.

1.2. Добавление

1. **Способ первый, простой:**

```
Node *add(Node *v, int x) {
  if (v == 0)
    return new Node(0, 0, x);
  if (x > v->x)
    v->r = add(v->r, x);
  else
    v->l = add(v->l, x);
  return v;
}
```

2. **Способ второй, персистентный:**

```
Node *add(Node *v, int x) {
  if (v == 0)
    return new Node(0, 0, x);
  if (x > v->x)
    return new Node(v->l, add(v->r, x), x);
  else
    return new Node(add(v->r, x), v->r, x);
}
```

Достоинство этого способа – персистентность. Он не изменяет переданное дерево, а создаёт новое, отличающееся от старого лишь добавленным значением.

3. **Способ третий, короткий:**

```
void add(Node *&v, int x) {
  if (v == 0)
    v = new Node(0, 0, x);
  else if (x > v->x)
    add(v->r, x);
  else
    add(v->l, x);
}
```

1.3. Операции

1. **Find**
   Поиск элемента в дереве. Спускаемся, пока не окажемся в вершине с искомым значением. На каждом шаге переходим к левому или правому сыну в зависимости от того, больше или меньше искомого текущее значение.
2. **Leftest, Rightest**
   Поиск наименьшего и наибольшего элемента в дереве. Переходим в левого или правого сына соответственно до тех пор, пока это возможно.

3. **Next, Previous**
   Поиск предыдущего и следующего элемента. Поскольку операции аналогичны, рассмотрим только поиск следующего. Возможно два случая.
   Пусть у данной вершины есть правый сын. Тогда любой предок текущей вершины либо больше правого поддерева (если рассматриваемая вершина находится в его левом поддереве), либо меньше самой рассматриваемой вершины (если она в его правом поддереве). Любая вершина, не являющаяся предком или потомком рассматриваемой, лежит в другом поддереве, поэтому она либо меньше данной, либо больше её правого поддерева. Осталось взять наименьшую вершину в правом поддереве.
   Если правого поддерева нет, будем переходить к отцу до тех пор, пока исходная вершина не окажется в его левом поддереве. Первый предок, для которого данная вершина находится в левом поддереве – это минимальный из предков этой вершины, превосходящий её. Вершины из правых поддеревьев этого и других предков, для которых наша вершина находится в левом, больше этих предков. Вершины из левых поддеревьев других предков меньше исходной, и поэтому они тоже не подходят. Таким образом, ответ – это первый предок, для которого данная вершина лежит в левом поддереве.

   Lm 1.3.1. Цикл, обходящий дерево в порядке возрастания с помощью операции next

```
1       for (v = leftest; v != 0; v = next(v))
```

   Работает за линейное время.

   Доказательство первое, простое.
   Маршрут v в дереве почти совпадает с эйлеровым обходом. Каждое ребро, кроме рёбер на пути из корня в наибольший элемент будет пройдено два раза. Рёбра же на пути из корня в самую правую вершину будут пройдены один раз.
   Доказательство второе, строгое.
   Индукция.
   База: листья обрабатываются за $\mathcal{O}(1)$. Это укладывается в $\mathcal{O}(n)$.
   Переход: Проход по дереву состоит из проходов по поддеревьям его корня. Оба поддерева обрабатываются за линейное от их размера время. В сумме получается линейное от размера дерева время.

Описанные операции работают за $\mathcal{O}(height)$, однако их можно ускорить до $\mathcal{O}(1)$.

1.4. Ускорение операций

1. **Find**
   Добавляем HashMap $x \to v$.
2. **Leftetst, Rightest**
   Храним значения крайних левого и правого элементов.
3. **Next, Previous**
   Поддерживаем двусвязный список вершин. Такая модификация называется прошивкой дерева. Каждая вершина содержит указатели на следующую и предыдущую в порядке возрастания значения ключа.

```
struct Node {
  Node *l, *r, *prev, *next;
  int x;
}
```

В качестве примера использования прошивки дерева можно привести обход set.

```
  for (auto it = s.begin(); it != s.end(); it++)
```

Этот код делает ровно то же самое, что и

```
  for (auto &x : s)
```

1.5. Удаление

Если вершина, которую мы хотим удалить, является листом, просто уберём её из дерева.

Если у вершины есть один сын, поставим его на её место.

Если же у вершины есть два поддерева, выберем минимальную (самую левую) вершину в правом поддереве и перенесём её на место удаляемой. Поскольку новая вершина меньше всех остальных вершин правого поддерева и больше всех вершин левого поддерева, условие двоичного дерева поиска сохранится.

1.6. Обходы дерева

1. **Симметричный обход** Выводит все вершины дерева в виде упорядоченного массива.

```
  void out(v) {
    if (!v) {
      return;
    }
    out(v->l);
    cout << v->x;
    out(v->r);
  }
```

2. **Прямой обход**

```
  void out(v) {
    if (!v) {
      return;
    }
    cout << v->x;
    out(v->l);
    out(v->r);
  }
```

Lm 1.6.1. Вывод прямого обхода позволяет однозначно восстановить дерево.

Действительно, мы знаем, что первая вершина – это корень дерева. После неё идёт набор чисел меньше неё. Это левое поддерево. Затем идёт набор чисел больше неё. Это правое поддерево. Таким образом, мы умеем выделять корень и поддеревья. Переходя от дерева к поддеревьям, мы будем спускаться от корня к листьям.

1.7. Способы вывода дерева

1. **В строку**
   Каждое поддерево берётся в скобки. На выходе имеем одну строку.

```
void Out( Node* v ) {
  if (!v) return;
  printf("(");
  Out(v->l);
  printf("%d", v->x);
  Out(v->r);
  printf(")");
}
```

2. **С отступом**
   Каждая вершина выводится на своей строке, с отступом от начала строки, пропорциональным её глубине. Вывести на экран значение вершины v с отступом d можно с помощью строки:

```
printf("%*s%d\n", 2 * d, "", v->x);
```

   Пишется дольше, но вывод читается лучше.

1.8. Set, Multiset и Map

На двоичных деревьях поиска возможно реализовать такие структуры, как множество, мультимножество и словарь. Множество представляется просто BST, в которое значения добавляются как ключи вершин. Словарь получается из множества добавлением к каждой вершине значения при сравнении только по ключу. Мультимножество получается, если добавлять к входным значениям дополнительную часть ключа.

Лекция по алгоритмам #2
AVL-дерево

9 февраля

2.1. Опеределение и основные свойства

Def 2.1.1. *Высота поддерева – наибольшее из расстояний от корня до листьев этого поддерева. Высота вершины – высота поддерева этой вершины.*

Def 2.1.2. *AVL-дерево – сбалансированное двоичное дерево поиска, в котором для каждой вершины выполняется инвариант: разница между высотами левого и правого поддеревьев не больше 1.*

Lm 2.1.3. AVL-дерево имеет высоту $O(\log n)$.

Доказательство. Оценим размер каждого поддерева. Сверху мы ограниченны размером полного бинарного дерева (т.к. это максамальное по размеру бинарное дерево с фиксированной высотой). Соотвественно для минимального размера наши сыновья должны быть также минимального размера и иметь разные высоты (это св-во рекурсивно распространяется вниз по дереву). Т.е. $SIZE_h \geq SIZE_{h-1} + SIZE_{h-2}$. Заметим, что эта формула соответствует числам Фибоначчи, а значит $SIZE_h \geq FIB_h \geq 1.6^h$ (оценка на значение h-го числа Фибоначчи). Получили, что $1.6^h \leq SIZE_h \leq 2^h$.
С помощью оценкци размера поддерева можно получить оценку глубины (прологорифмировав левое и правое неравенство поотдельноести):
$\log_2 size \leq h \leq log_{1.6} size$

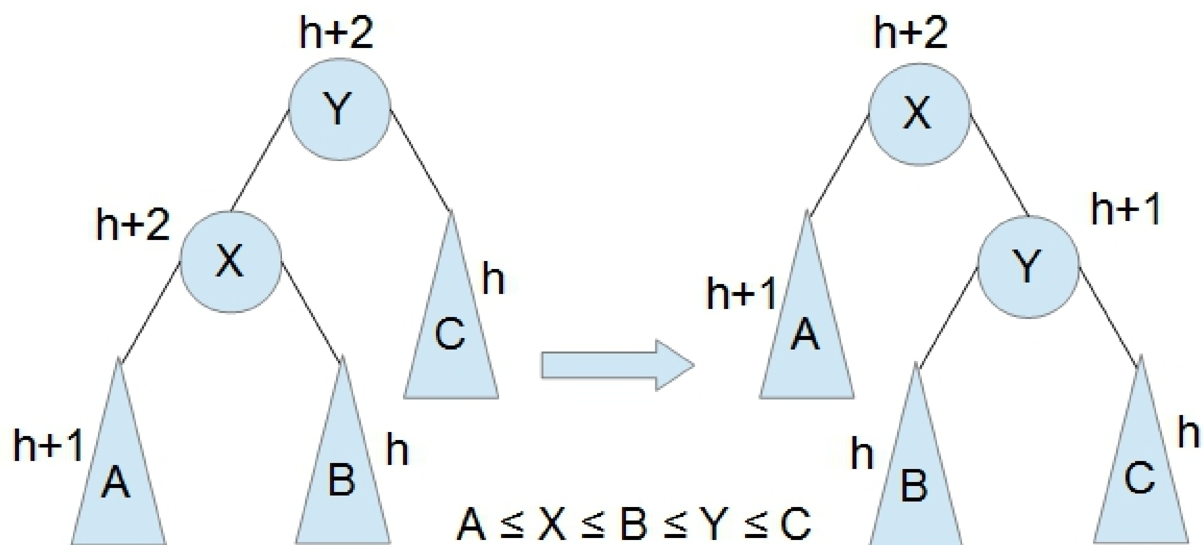
■

2.2. Добавление

При добавлении находим место в дереве, куда мы будем добавлять новый элемент x. Для этого спускаемся вниз по дерево как при поиске, пока не придем в пустую/фиктивную вершину. Вставляем в это место новую вершину. После добавления у нас могли измениться глубины, а значит в некоторых вершинах мог сломаться инвариант. Будем рассматривать вершины на пути от добавленной до корня (это можно сделать или циклом поднимаясь по предкам, или разворачия рекурсию, если операция добавления реализована рекурсивно). Пусть мы стоим в некоторой вершине v. Сначала проверяем, выполняется ли инвариант в текущем поддереве. Если да, то пересчитываем высоту вершины. Если она не изменилась, то заканчиваем исправление (т.к. высоты вершин выше тоже не могли измениться, а значит там ничего не сломано). Иначе продолжаем исправление и переходим к вершине выше.

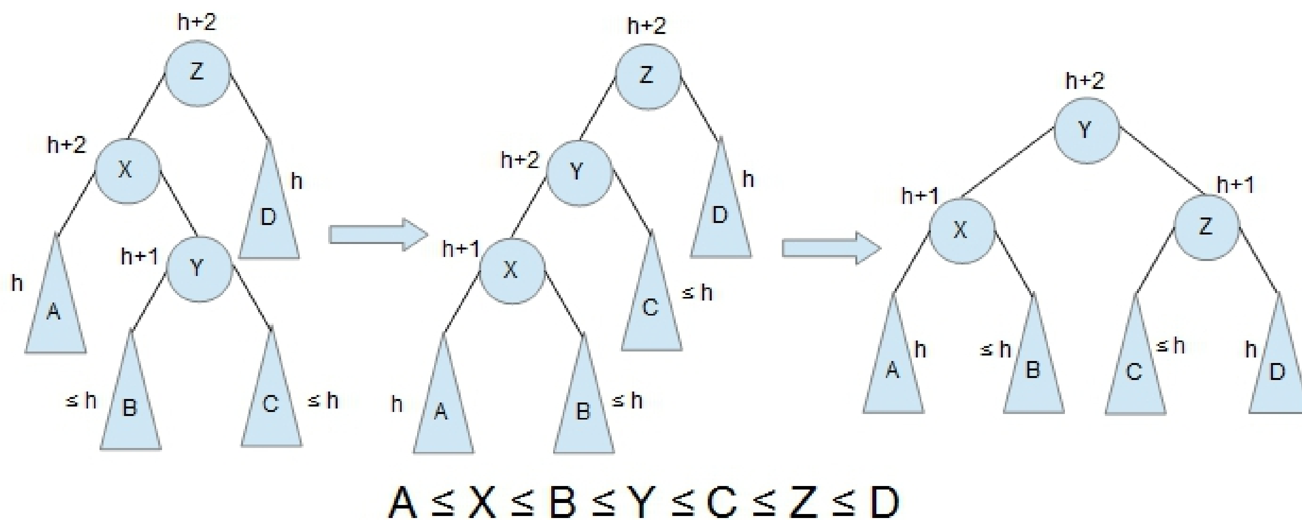
Теперь рассмотрим ситуацию, когда для текущей вершины инвариант не выполняется. Т.е. один из сыновей вершины имеет высоту на 2 больше, чем второй. Для удобства будем считать, что это левый сын (для случая правого сына все операции аналогичны). Обозначим его высоту за $(h + 2_)$, тогда у правого высота h. У левого сына один из его сыновей имеет высоту $(h + 1)$. Заметим, что это всегда только 1 из сыновей (после текущего добавления мы могли увеличить высоту только одного из них, а тогда текущаяя вершина стала высоты $(h + 2)$ еще раньше). Рассмотрим 2 случая: если это левый сын сына (нужно больше сыновей), и если это правый.

В первом случаи нам на помощь придет правое (аналогично ему есть левое) малое вращение (в англ. вики его называют просто вращением). Название его соответсвтует тому, что оно делает: мы поворачиваем часть поддерева и переподвешиваем вершины, тем самым изменяя высоты.



На рисунке показано, как происходит правое вращение. Левое вращание выполняется аналогично и является обратным к правому (из правого поддерева мы получим левое). Вследствии неравенства после поворота наше дерево останется корректным двоичным деревом поиска ($A \leq X$ обозначает, что все ключи в поддереве не больше X). После выполнения поворота на не нужно увеличивать высоту текущей вершины (на рисунке видно, что она не изменилась), а значит мы закончили исправления.

Во втором случаи на поможет большое правое вращение. На самом деле оно будет просто состоять из двух малых. Мы ставим на место корня текущего поддерева „плохого“внука. Для этого мы поднимаем его сначала малым левым, а потом малым правым вращениями.



Корни поддеревьев B и C до поворота могли иметь только высоты h или $(h-1)$. Поэтому они не будут ломать инвариант в своих поддеребьях (их „братья"имеют высоту h). Аналогично 1-му случаю нам не нужно увеличвать высоту корня поддерева, а значит мы закончили исправление дерева. Благодарю неравенству наше дерево осталось корректным деревом поиска.

Лекция по алгоритмам #3
Удаление из AVL и Неявный Ключ

12 февраля

3.1. Удаление из AVL дерева

1. Делаем так же, как и при обычном удалении из BST.
2. На обратном ходе рекурсии делаем ребаланс. Заметим, что высота у ребенка уменьшилась не больше чем на 1 $\Rightarrow$ достаточно сделать один из поворотов.

Lm 3.1.1. $Time(\texttt{AVL::add}) = \Omega(\log n)$.

Доказательство. От противного:

1. Добавим n неупорядоченных элементов в дерево.$(o(n \cdot \log n))$.
2. Обойдем дерево. $(\mathcal{O}(n))$.

Получили сортировку за время $o(n \cdot \log n)$. Противоречие. ■

3.2. Implicit Key

- **k-ый элемент за $\mathcal{O}(\log n)$**

Для каждой вершины теперь дополнительно храним $size$ — размер поддрева.

```
1  struct Node
2      Node* left, right;
3      size_t size;
4  Node* Node::null = new Node();
5
6  inline size_t getKey(Node* v)
7      return v->left->size; // Элементы нумеруются с нуля
8
9  inline void refreshSize(Node* v)
10     v->size = v->left->size + v->right->size + 1;
11
12 Node* find(Node* v, size_t key)
13     if (v == Node::null) return Node::null;
14     if (getKey(v) == key) return v;
15     if (getKey(v) < key) return find(v->right(), key - (v->left->size + 1);
16     if (getkey(v) > key) return find(v->left(), key);
```

- **Массив при помощи AVL.**

Хотим уметь делать следующие операции:

1. `get`(i) – получить элемент на i-той позиции.
2. `set`(i, x) – присвоить элементу на i-ой позиции значение x.
3. `add`(i, x) – добавить элемент со значением x после i-того.

Заметим, что в AVL ключ нам нужен только при спуске вниз и никак не влияет на ребаланс. Значит, можно использовать неявный ключ ничего не сломав. Тогда:

1. `get`(i) := AVL.find(i)->data;
2. `set`(i, x) := AVL.find(i)->data = x;
3. `add`(i, x) := AVL.add(i)->data = x;

Здесь AVL.find(key) – поиск по неявному ключу, описанный выше, а AVL.add(key) – добавление как в обычном AVL, но с поправкой на неявный ключ, то есть, со спуском, как в AVL.find(key). Очевидно, что время работы каждой операции $\mathcal{O}(\log n)$.

3.3. Случайное BST-дерево

Рассмотрим несбалансированное BST. Очевидно, что случайная перестановка задает случайное дерево.

Lm 3.3.1. Вспомогательная лемма $\sum\limits_v size_v = \sum\limits_v depth_v$

Доказательство. Зафиксируем v.

1. v лежит в поддереве любой вершины, находящейся на пути от корня до v.
2. Нет других вершин, у которых v лежит в поддереве.

■

Lm 3.3.2. Матожидание средней глубины вершины равно $2 \cdot \ln n$.

Доказательство. В доказателстве `QuickSort` мы использовали формулу подсчета времени $T(n) = T(x) + T(n - x) + n$. Эта же формула справедлива для подсчета суммы количества вершин в поддеревьях, поэтому сошлёмся на док-во `QuickSort`. ■

Lm 3.3.3. Матожидание максимальной глубины вершины не больше $4 \cdot \ln n$;

Доказательство. Эта лемма была без доказательства. ■

Лекция по алгоритмам #4
Декартово дерево

12 февраля

4.1. Декартово дерево (cartesian tree)

В обычном дереве поиска каждой вершине соответствует пара из ключа и информации. Теперь в каждой вершине так же будет хранится ее приоритет (поле y). Декартово дерево - это дерево поиска, в котором доплнительно выполнен следующий инвариант: приоритет родителя меньше (или меньше или равен, если приоритеы могут совпадать), чем у его детей. Другими словами, $v{-}{>}y$ меньше $v{-}{>}l{-}{>}y$ и $v{-}{>}r{-}{>}y$. Получается, что cartesian tree – это дерево поиска по ключам(x) и куча по приоритетам(y). Так как каждая вершина представляет собой пару (x, y), то удобно рисовать декартово дерево на плоскости.

4.2. Декартово дерево (treap)

Treap – это cartesian tree, только все приоритеты выбираются случайным образом. Другими словами, это обычное дерево поиска, только каждой вершине сопоставлен случайный приоритет, который у сыновей должен быть больше, чем у родителей. Так же будем считать, что все приоритеты различны (на практике обычно у нас порядка 10^5 вершин, а приоритеты выбираются из промежутка $[1..10^9]$, поэтому вероятность совпадения крайне мала).

Lm 4.2.1. Декартово дерево единственно.

Доказательство. Покажем это рекурсивно. Понятно, что вершина с самым маленьким приоритетом u должна быть корнем (так как приоритеты разные, такая вершина определяется однозначно). Теперь все вершины разделились на 2 группы - те, у которых ключ меньше $u{-}{>}x$, и те, у которых ключ больше или равен, чем $u{-}{>}x$. При этом в каждой группе все приоритеты меньше, чем у u, поэтому можно просто рекурсивно построить деревья для этих групп, а потом сделать их корни левым и правым сыном u. ■

Lm 4.2.2. Декартово дерево – случайное дерево.

Доказательство. Корнем дерева может быть любая вершина с одинаковой вероятностью (вероятность того, что ее приоритет больше остальных). Заметим, что в определении случайного дерева корень тоже выбирался равновероятно. Далее построение дерева слева и справа от корня идет независимо, при этом, например, слева корнем снова станет случайная вершина из оставшихся, так же, как и при построении случайного дерева. В общем, видно, что процедура построения декартова дерева идентична построению случайного дерева, если сказать, что мы рассматриваем вершины в порядке увеличения приоритетов. Тогда каждая перестановка вершин равновероятна, и мы получаем случайное дерево. ■

Пока что мы только показали, что если строить декартово дерево по алгоритму из доказательства первой леммы, то мы получим случайное дерево, а значит, его средняя глубина будет не больше $2\ln n$, а максимальная глубина не превосходит $4\ln n$. Так же можно заметить, что время построения treap оценивается рекурсивным соотношением $T(n) = T(x) + T(n-x-1) + n$,

которое полностью идентично тому, что мы получали при оценке времени работы *qsort*. Поэтому, построение декартова дерева работает $O(n \ln n)$.

4.3. О равных ключах

В общем случае считается, что в cartesian tree в левом поддереве лежат вершины с меньшим ключом, а в правом - с большим или равным. На практике в зависимости от задачи можно хранить пары из ключа и времени его добавления, можно в вершинке хранить вектор всех вершин с таким ключом, можно позволить существовать нескольким вершинам с разным ключом. Я буду придерживаться последнего варианта.

4.4. Split

Функция, принимающая на вход дерево T и ключ x. Возвращает пару деревьев l, r такую, что множество вершин из T - это множество вершин из объединения l и r, при этом ключи вершин из l меньше x, а ключи вершин из r больше или равны x. Если нарисовать дерево на плоскости, то мы проводим вертикальную черту и делим дерево на две части, лежащие по разные стороны от этой черты. Алгорим рекурсивный. Пусть мы стоим в вершине v. Тогда, если ее ключ сейчас меньше x, то она и ее левое поддерево останутся слева от x и не изменятся. Поэтому мы вызываемся рекурсивно от правого сына и получаем его разбиение на l_1, r_1 (то есть, мы разделили правого сына v на две части, одна из которых меньше x и останется сыном v, а другая часть лежит справа от x, и поэтому она уже не будет связана с v). Теперь нужно в правого сына v положить l_1 и вернуть наверх пару из v и r_1. Случай, когда v справа от x, разбирается так же.

```
1  pair<node*, node*> split(node *v, int x) {
2    if (v == NULL)
3      return make_pair(NULL, NULL);
4    if (v->x < x) {
5      l1, r1 = split(v->r, x);
6      v->r = l1
7      return make_pair(v, r1);
8    } else {
9      l1, r1 = split(v->l, x);
10     v->l = r1;
11     return make_pair(l1, v);
12   }
13 }
```

4.5. Merge

Функция, обратная split. Принимает на вход два дерева, между которыми можно провести черту (то есть все ключи одного меньше всех ключей другого) и объединяет их в одно. Условие про ключи важно, нельзя склеивать два дерева, у которых непонятно как соотносятся ключи. Алгоритм также рекурисвный. Пусть мы склеиваем вершины l и r. Понятно, что вершина с меньшим приоритетом станет корнем, пусть, например, r. Тогда ее правый сын не поменяется, а левый сын получится склеиванием l и предыдущего левого сына $r{-}{>}l$. Случай, когда приоритет l меньше, разбирается так же.

```
1  node* merge(node *l, node *r) {
2    if (l == NULL)
3      return r;
4    if (r == NULL)
5      return l;
6    if (l->y > r->y) {
```

```
7      r->l = merge(l, r->l);
8      return r;
9    } else {
10     l->r = merge(l->r, r);
11     return l;
12   }
13 }
```

4.6. Медленные Add и Delete

Все операции с декартовым деревом обычно выражаются через split, merge и спуск по дереву. Например, чтобы вставить вершину с ключом x, достаточно сначала разрезать дерево по x, получить, что новая вершина лежит между двумя кусками дерева, после чего просто склеить сначала левый кусок с новой вершиной, потом получившуюся часть с правым куском.

```
1 node* add(node *T, node *v) {
2   l, r = split(T, v->x);
3   T = merge(l, v);
4   T = merge(T, r);
5   return T;
6 }
```

Delete выражается аналогично. Разрезаем дерево по x, потом по $x + 1$. Дерево теперь имеет вид левый кусок, все вершины с ключом x, правый кусок. Далее нужно склеить правый и левые куски, игнорируя вершины между ними. Здесь важно понимать, что при таком способе удаляться все вершины с ключом x. Если надо удалить только одну, то можно из дерева, в котором все вершины имеют ключ x, удалить корень, а оставшиеся части вновь склеить вместе с левым и правым куском.

```
1 node* delete(node*T, int x) {
2   l, r = split(T, x);
3   l1, r1 = split(r, x + 1);
4   return merge(l, l1->l, l1->r, r1);
5 }
```

Это версия с удалением одной, если надо удалить все, можно вернуть $merge(l, r1)$.

Полученные оперции работают от 3 до 5 вызовов split/merge, поэтому они не очень быстрые.

4.7. Быстрые Add и Delete

Можно реализовать Add за один split, а Delete за один merge. Идеи одинаковые, рассмотрим, например, Add. Если бы у добавляемой вершины был самый маленький приоритет, было бы достаточно разделить дерево одним splitом на две части и подвесить их к новой вершине. Поэтому общий алгоритм такой: пока приоритеты вершин маленькие, спускаемся вниз. Как только дошли до того, что у сыновей все приоритеты большие, делим это поддерево одним splitом и подвешиваем к новой вершине.

```
1 node* add(node *T, node *v) {
2   if (v->y < T->y) {
3     v->l, v->r = split(T, v->x);
4     return v;
5   }
6   if (v->x < T->x)
7     return add(T->l, v);
8   else
```

```
9        return add(T->r, v);
10   }
```

Как видно, такой Add работает уже за один *split*.

Операция Delete похожа, достаточно спуститься до удаляемой вершины, сделать merge ее детей и вернуть наверх.

4.8. Неявный ключ

Идея точно такая же, как и в AVL-дереве. В каждой вершине храним ее размер. Тогда при поиске k-ой вершины смотрим на размер левой компоненты и спускаемся либо налево, либо направо. Так же надо добавить функцию $recalc(v)$, которая пересчитывает размер поддерева, как только у вершины поменялись дети. Соответственно, ее надо добавить после всех операций вида $v{-}{>}l{=}u$. Легко заметить, что для замены явного ключа на неявный достаточно добавить recalc в нужные места и заменить условие на спуск в split.

4.9. Про Add для вещественных чисел

Заметим, что если ключи вещественные, то для вставки надо писать 2 разных *split* - один отрезает поддеревья с условиями $(\leqslant x, > x)$, другой $(< x, \geqslant x)$. Чтобы не писать код дважды, можно сделать дерево по неявному ключу и вместо второго split просто откусывать один элемент.

4.10. Приоритеты не нужны

На самом деле, можно не хранить приоритеты, а в те места, где мы выбираем, какую из двух вершин делать корнем (например, в merge) выбирать левую с вероятностью $\frac{l.size}{l.size+r.size}$. Утверждается, что от этого асимптотика не изменится (без доказательства). Такое дерево называется RBST (Randomized Balanced Search Tree).

4.11. Время работы

Мы доказали, что при построении декартова дерева оно получается случайным, и потому его глубина логарифмическая. Доказательство того, что дерево остается случайным после всех операций split/merge, требует некоторых выкладок, которых мы не делали.

Лекция по алгоритмам #5
Red-Black, AA, 2-3, 2-3-4, B+ Trees

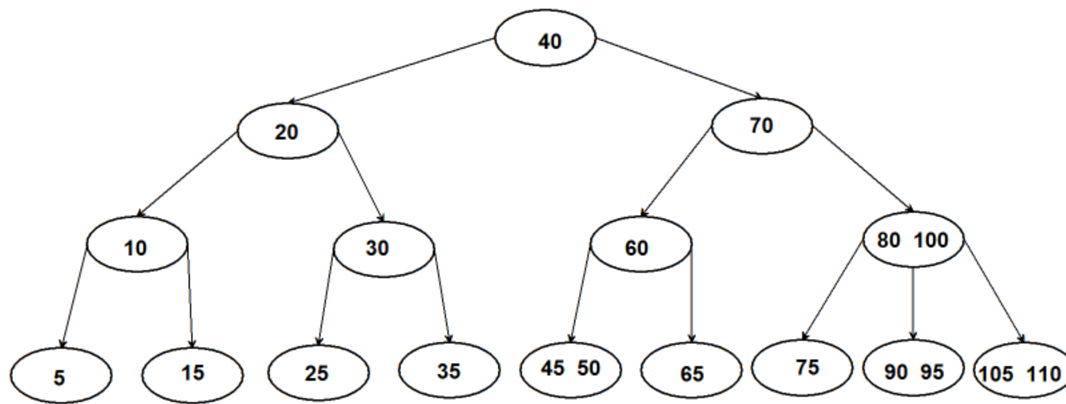
19 февраля

Все описываемые деревья являются сбалансированными деревьями поиска. Далее будет о том, что они из себя представляют и как связаны друг с другом.

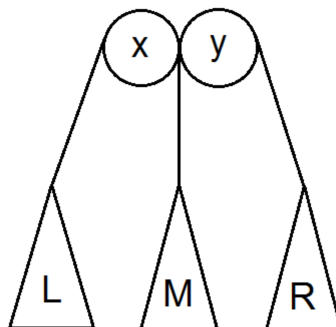
5.1. 2-3 Tree

В дереве существуют вершины двух видов: 2-вершины и 3-вершины. 2-вершины такие же, как и обычном дереве поиска, но обязательно имеют два ребёнка. 3-вершины содержат в себе сразу два ключа и имеют трёх детей. Таким образом любая вершина либо лист, либо имеет двух детей, либо трёх. Вершин с одним ребёнком в дереве нет.

Пример дерева:



Пример двойной вершины:



Для ключей выполнено неравенство дерева поиска:
$L < x < M < y < R$ (в предположении, что все ключи различны)

Инвариант: все листья дерева находятся на одинаковой глубине.
Из этого, в свою очередь, следует, что $\log_3 n \le h \le \log_2 n$, где h – высота дерева.

- **Добавление в дерево**

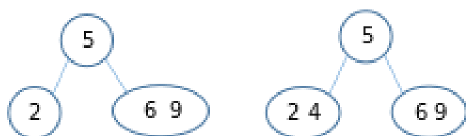
Найдём стандартной процедурой лист, в который должен быть помещён новый ключ. Добавим его в эту вершину. В результате чего в этой вершине могло стать три ключа. Это плохо, нужно исправлять. Возьмём средний ключ из трёх и "вытолкнем"его наверх, в предка. Теперь осталась вершина с двумя ключами и четырьмя детьми, разделим её на две, отдав первой меньший ключ и два левых сына, а правой второй ключ и оставшихся двух детей. Теперь на этом уровне всё стало в порядке, однако проблема могла появиться в предке. Поднимемся и сделаем для него аналогичные действия. Поднимаясь ближе к корню, либо в какой-то момент мы остановимся в вершине, в которой стало два ключа, либо, если на пути от корня до листа изначально все вершины имели по два ключа, из корня вытолкнется вверх один ключ, создав новый корень и увеличив высоту дерева на 1.

Пример добавления:

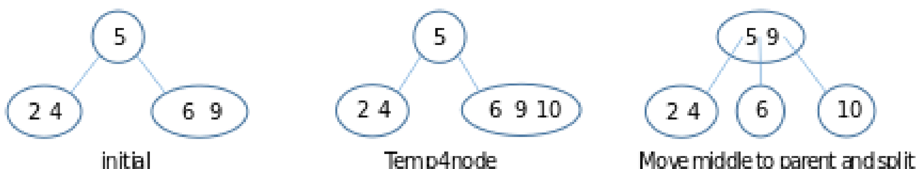
Изначально есть дерево с ключами 2, 5, 6, 9.

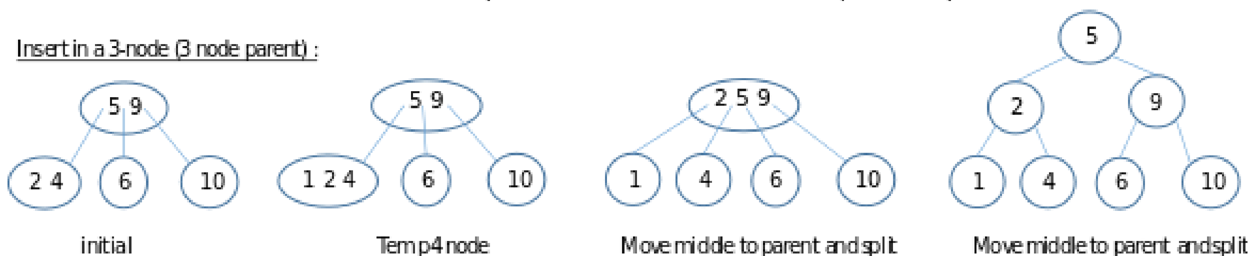
Последовательно добавляются три ключа: 4, 10, 11.

Insert in a 2-node :



Insert in a 3-node (2 node parent) :



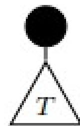


- **Удаление из дерева**

Найдём нужный ключ в дереве. Как и в обычном BST найдём следующий по значению ключ, он будет в листе. И обменяем местами ключи в этом листе и удаляемой вершине. Теперь этот ключ нужно удалить уже из листа.

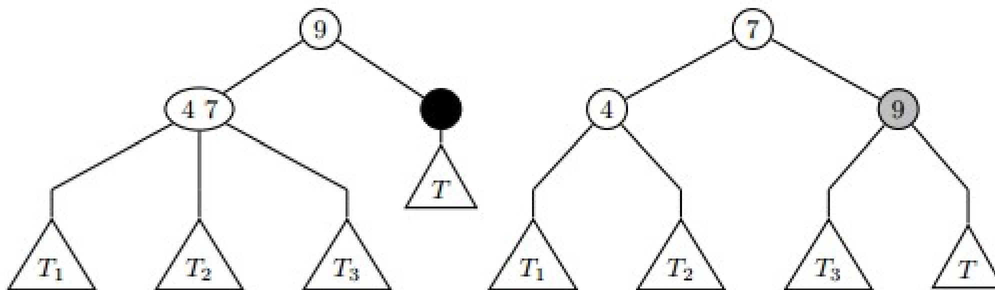
Самый простой случай, когда в этом листе содержится два ключа. Тогда просто удалим нужный ключ, в вершине останется один ключ и структура дерева не нарушится.

Так как некоторые случаи проталкивают "проблему"наверх, то и рассматривать случаи будем в общем случае. Для удобства введём понятие "пустая вершина". Это вершина с нулём ключей и одним сыном. Для пустой вершины-листа, очевидно, сын будет пустым. Заменим вершину, которую нам нужно удалить, на пустую.

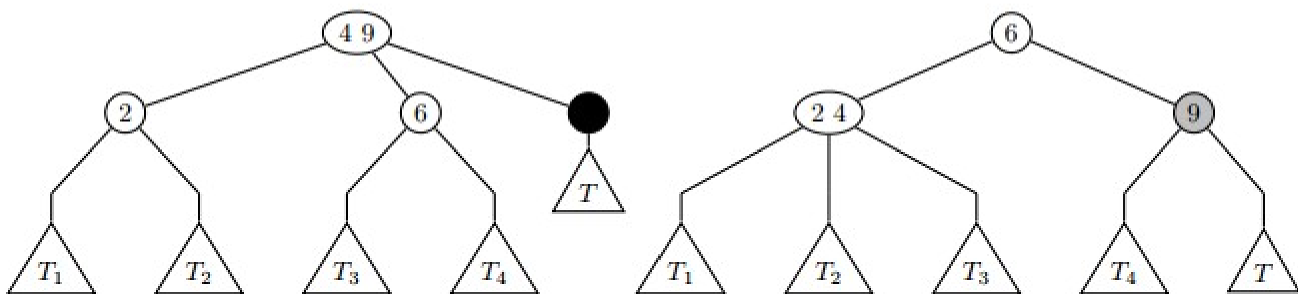


Теперь образовался один из случаев:

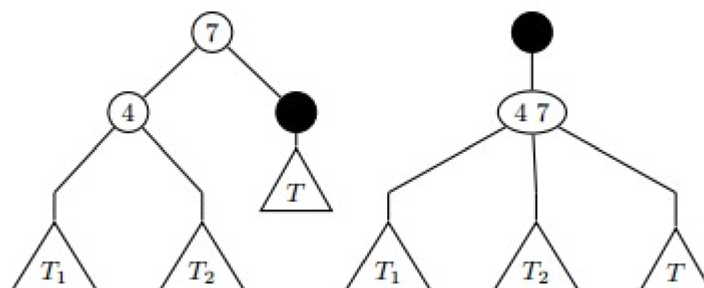
1. У пустой вершины есть брат с двумя ключами. Заберём у него один ключ. А точнее переместим его в отца, а ключ отца заберём себе.



2. У пустой вершины два брата, то есть у отца два ключа. Заберём один ключ себе, а из двух братьев сделаем одного.



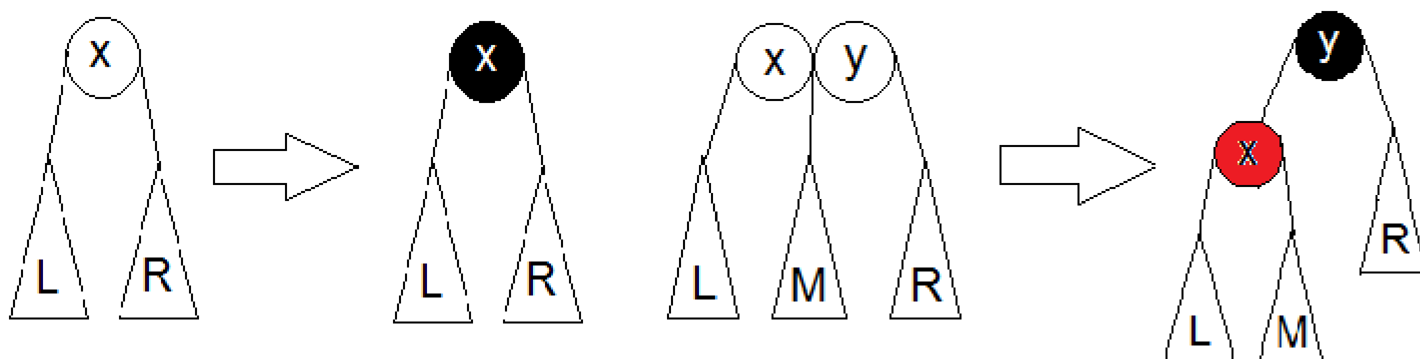
3. У пустой вершины один брат с одним ключом и отец, соответственно, с одним ключом. Ни у кого ничего не заберёшь. Тогда склеим отца и брата, получим вершину с двумя ключами. Поместим пустую вершину на место отца, а новую вершину подвесим за неё. Сына пустой вершины подвесим за эту новую вершину. Теперь на данном уровне всё в порядке, но на уровень выше – нет. Поднимемся и решим задачу уже там.



4. И простой случай, который может образоваться только один раз, когда мы дошли до корня. А именно: пустая вершина оказалась на месте корня. В таком случае мы просто удаляем её, а её сын становится новым корнем.

5.2. AA tree *Названы в честь придумавшего их Арне Андерссона.*

AA дерево – бинарное дерево, вершины которого бывают чёрными и красными. Так как только что было описано устройство 2-3 дерева, то описать AA дерево будет проще всего, сказав, что оно получается из 2-3 дерева заменой обычных вершин на чёрные, а двойные вершины на цепочку из чёрной вершины и подвешенной к ней красной.



Из такого определения сразу следует несколько свойств AA дерева:

1. Назовём *чёрной глубиной вершины* количество чёрных вершин на пути от вершины до корня. В AA дереве чёрная глубина всех листьев одинакова.
2. Не бывает рёбер между красными вершинами.
3. Корень всегда чёрный.
4. Красная вершина всегда является левым сыном своего отца.

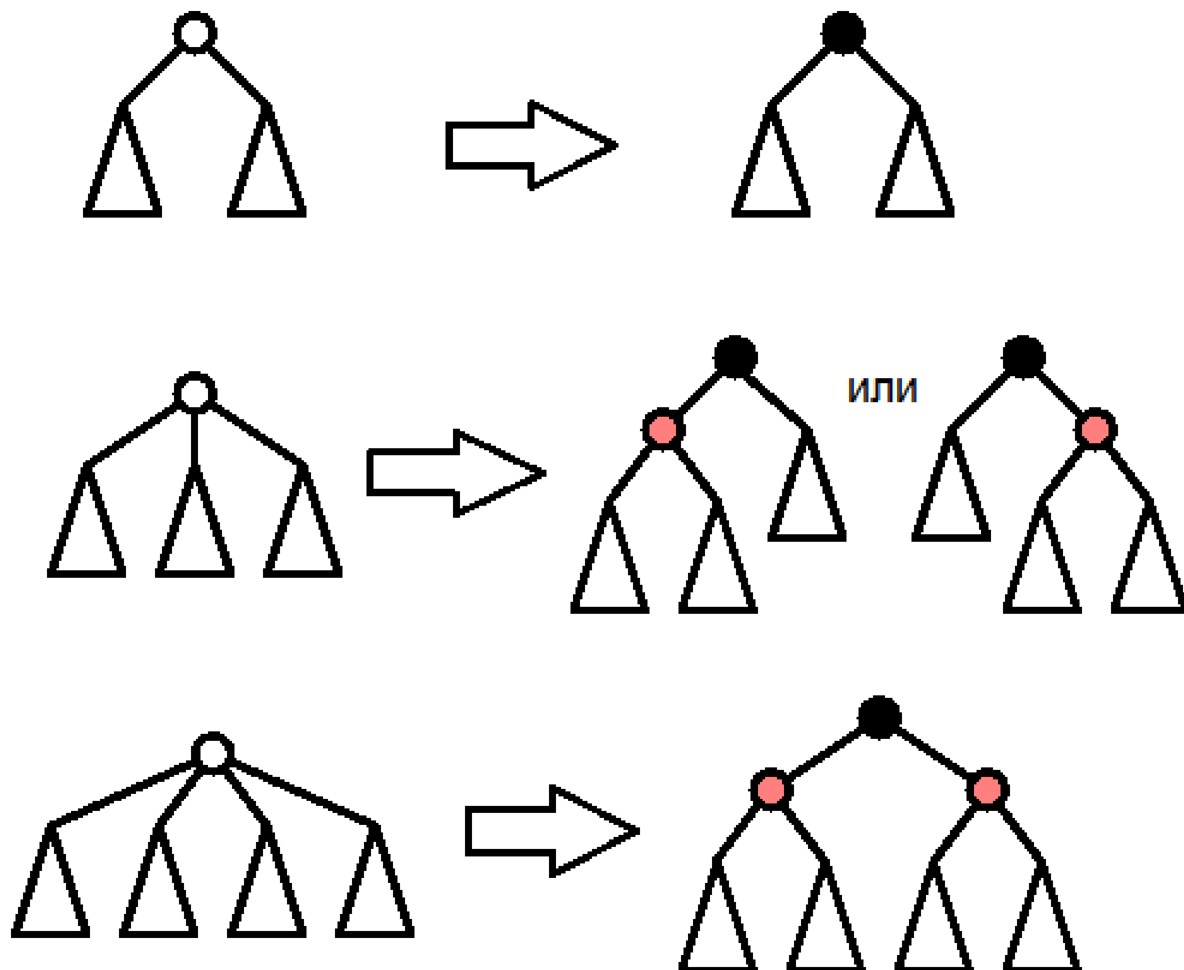
5.3. 2-3-4 Tree

2-3-4 дерево – это то же 2-3 дерево, только с добавлением вершин с тремя ключами и четырьмя детьми. Как и у 2-3 дерева, глубина всех листьев равна. Операции добавления и удаления также аналогичны 2-3 дереву, только надо рассмотреть побольше случаев. Какой интерес для нас представляет это дерево, если оно почти ничем не отличается от 2-3 дерева? Сейчас узнаем.

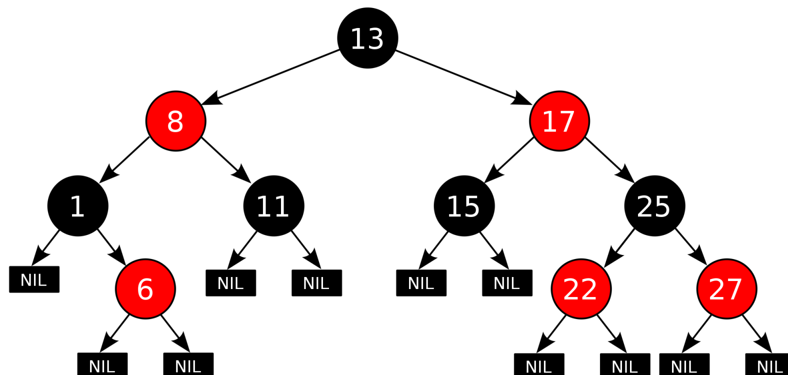
5.4. Red-Black Tree

Не случайно в этой главе дано столько деревьев и все они вместе. Мы уже встречались с аналогиями и этот случай не будет исключением.

Как можно понять из названия, красно-чёрное дерево содержит красные и чёрные вершины, оно бинарное. Очень похоже на AA дерево, а если точнее, то это AA дерево без ограничения на то, что красные вершины могут быть только левыми сыновьями. И раз уж AA дерево можно получить из 2-3 дерева, RB дерево можно получить из 2-3-4 дерева. Для этого нужно применить следующие замены:



Таким образом, мы представили 2-3-4 дерево в удобном бинарном формате. Работают эти деревья аналогично, но как раз благодаря тому, что RB дерево бинарное, его намного проще реализовывать и на практике используют именно его. Стоит упомянуть, что в целях упрощения реализации принято считать (а в последствии и программировать), что все листья дерева – пустые чёрные вершины. А именно, во всех местах дерева, где у какой-то вершины мог бы быть сын, но его нет, создаётся фиктивная пустая чёрная вершина. На практике обычно создаётся ровно одна такая фиктивная вершина, на которую и ссылаются все "пустые дети".

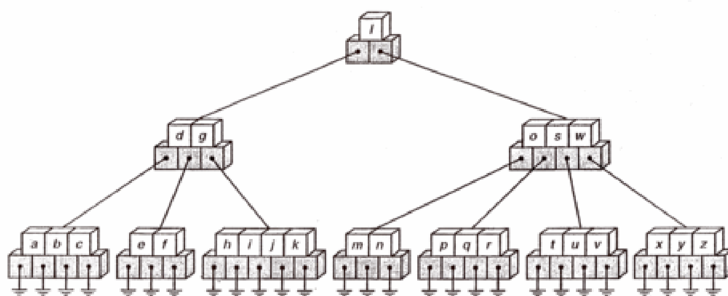


5.5. B+ Tree

Обобщение 2-3 и 2-3-4 деревьев – B деревья. Сейчас мы рассмотрим один из вариантов B деревьев – B+ дерево.

B+ дерево порядка t (со степенью ветвления t) – дерево поиска со следующими свойствами:

1. Корень дерева содержит от 1 до $2t-1$ ключей (0 или от 2 до $2t$ детей).
   Все остальные узлы содержат от $t-1$ до $2t-1$ ключей (от t до $2t$ детей).
2. Пусть v – какая-то вершина, не лист, $k_1, k_2, ..., k_n$ – ключи в v, $T_1, T_2, ..., T_{n+1}$ – дети v.
   Тогда $\forall x \in T_1 : x < k_1 . \forall\ 2 \le i \le n : \forall x \in T_i : x \in (k_{i-1}, k_i) . \forall x \in T_{n+1} : x > k_n$.
   Или проще – как и в 2-3, и в 2-3-4, выполнен порядок на ключах и на ключах в поддеревьях детей.
3. Глубина всех листьев одинакова.
4. Все данные хранятся в листьях, в промежуточных же вершинах содержаться только ключи.
5. В каждом листе есть ссылка на соседний лист.



Если первые три пункта понятны, но непонятно, зачем они нужны, то последние два – совсем непонятны.

А дело в том, что структура была придумана и используется для хранения данных во внешней памяти. Выигрыш перед остальными структурами данных заключается в том, что обращение к памяти, как правило, идёт не к паре байт, а сразу к нескольким сотням байт, и операция эта очень дорогая. Поэтому, грубо говоря, чем меньше обращений к памяти делает структура, тем быстрее она работает. И можно по свойствам B+ заметить, что если t довольно большое, то обращений будет действительно мало. А именно, если t взять, к примеру, равным 1001 и хранить в дереве 10^9 каких-то объектов, то поиск нужного потребует всего лишь 3 обращения к памяти. На практике t выбирается так, чтобы одна вершина дерева помещалась в один блок памяти и при этом содержала как можно больше ключей.

Лекция по алгоритмам #6
Splay Tree

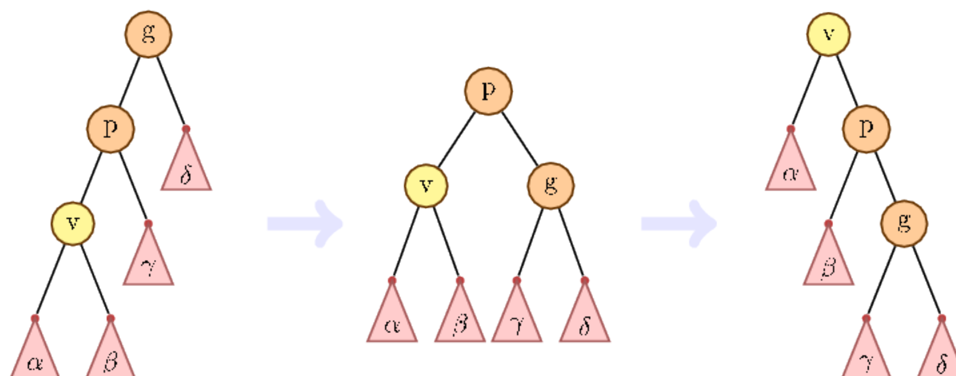
19 февраля

Splay-дерево — это самобалансирующееся бинарное дерево без каких-либо дополнительных условий. В худшем случае у него может быть линейная глубина, но в среднем все нормально. Оно хорошо тем, что позволяет быстро находить элементы, к которым недавно обращались.

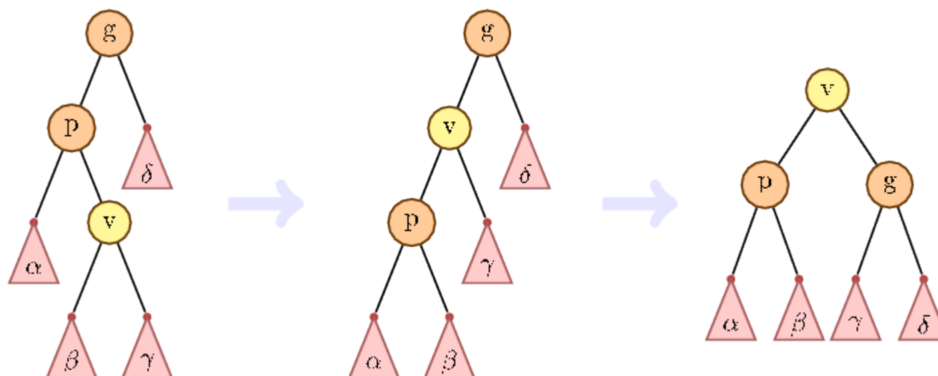
6.1. Операции

Все основные операции работают так же, как и в обычном BST, но для того, чтобы дерево сохраняло логарифмическую высоту после каждой "обычной" операции выполняется операция splay, которая поднимает данную вершину в корень с помощью вращений, балансируя дерево.

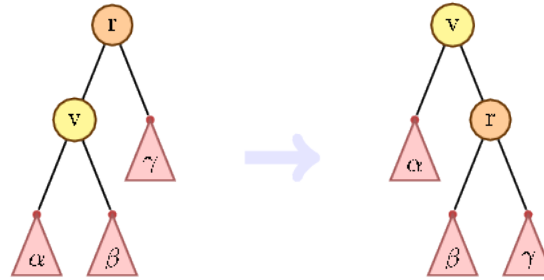
6.1.1. Zig-zig вращение



6.1.2. Zig-zag вращение



Если дедушки нет, то делаем такое вращение:



Нетрудно заметить, что zig-zag — это одно из вращений AVL-дерева.

6.2. merge и split

Для merge берем самый большой элемент из левого дерева, выполняем splay, получаем дерево, у корня которого нет правого сына. Подвесим туда правое дерево.

Для split по x запускаем splay(x) (если x нет, то можно, например, добавить), возвращаем поддеревья корня.

6.3. Асимптотика

Lm 6.3.1.

$$\texttt{Пусть } (a+b)=1; a,b>0$$
$$\log a+\log b\to\max$$
$$\Rightarrow a=b=\tfrac{1}{2}; \log a+\log b=-2$$

Доказательство.

$$b=1-a$$
$$(\log a+\log(1-a))'=0$$
$$\frac{1}{a}-\frac{1}{1-a}=0\Rightarrow a=\frac{1}{2}$$

■

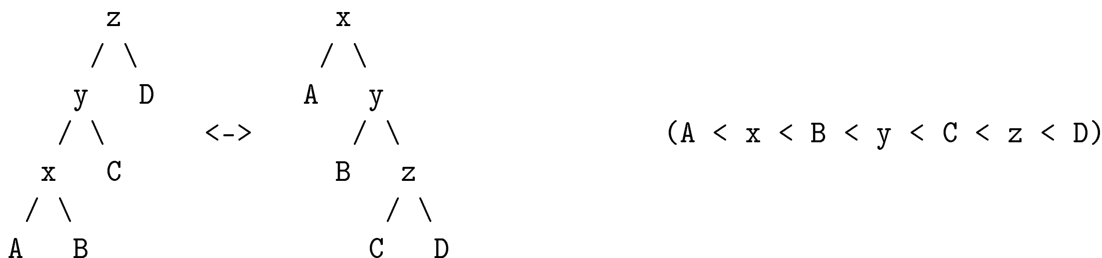
Теорема 6.3.2. Выполнили n операций. Тогда:

$$\sum_{i=1}^{n} t_i\le 4n\log n$$

Теорема 6.3.3. Если при операции `splay(v)` мы подняли ее в u, тогда:

$$\texttt{Пусть } R_v=\log(\text{size}_v)$$
$$a_i\le 3(R_u-R_v)$$
$$a_i=t_i+\Delta\phi$$
$$\phi=\sum R_v\le n\log n$$

Доказательство. Для zig-zig Покажем, что если после вращения мы поднялись из v_{i-1} в v_i, то $a_i \le 3(R_{v_i} - R_v)$. Тогда после всех вращений почти все сократится и получится $a_i \le 3(R_u - R_v)$. Для zig-zig (x):



$$2 + R'_x - R_x + R'_y - R_y + R'_z - R_z = 2 - R_x + R'_y - R_y + R'_z$$

(так как $R'_x = R_z$)

$$2 - R_x + R'_y - R_y + R'_z \le 2 - R_x + R'_x - R_x + R'_z$$
$$= 2 + R'_x - 2R_x + R'_z$$

Пусть S_v — размер поддерева v. Тогда по лемме $R'_x - 2R_x + R'_z = \log(S_x/S'_x) + \log(S'_z/S'_x) \le -2$

Подставим минус это вместо 2 в полученном выражении. Таким образом:

$$(2R'_x - R_x - R'_z) + R'_x - 2R_x + R'_z = 3(R'_x - R_x)$$

■

Для zig-zag доказывается так же, а для простого вращения доказывается, как ни странно, просто. Вот здесь хорошие доказательства: http://www.cs.cornell.edu/courses/cs3110/2011sp/recitations/rec25-splay/splay.htm.

Теорема 6.3.4 (О статический оптимальности)**.**

Хотим найти i k_i раз. Тогда

$$\sum a_i \le \sum k_i \log(n/k_i)$$

Иными словами, даже если мы заранее знаем запросы, то мы не можем построить никакого BST, которое будет асимптотически быстрее.

Без доказательства.

6.3.1. Бор

В каждой вершине храним `map<char, int> next[N]`. Если map — это “обычное” BST, то $t_i = |s| \cdot \log|\Sigma|$, но если это splay-дерево, то $t_i = |s| + \log|\Sigma|$, где Σ — алфавит. Идея в том, что каждое splay-дерево будет давать разность логарифмов и в итоге все сократится. Похожая идея используется в Link-Cut Tree.

6.3.2. 2-3-tree

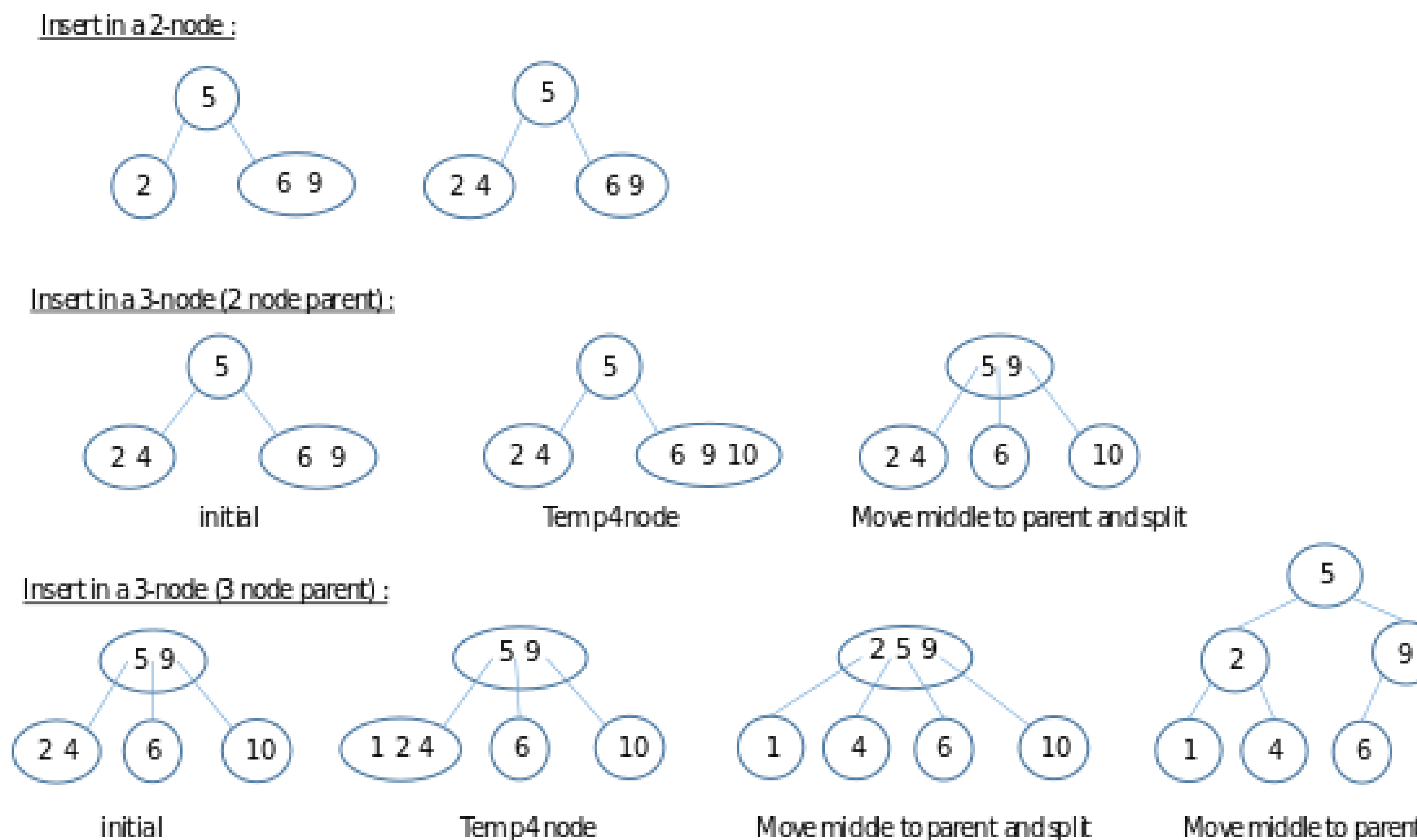
Данное дерево имеет два типа вершин: 2-вершины и 3-вершины. Кроме того, оно является сбалансированным, то есть все листья находятся на одной глубине. Благодаря этому дерево имеет логарифмическую высоту.

2-вершины обычные вершины обычного бинарного дерева поиска.

3-вершины хранят сразу 2 элемента и имеют трех сыновей. $L < x_1 < M < x_2 < R$

Search работает интуитивно понятным образом.

Insert находим подходящий лист и вставляем туда. Есть 3 случая, один из них хороший, два не очень. Все они разобраны на картинке:



Delete Как и в большинстве бинарных деревьев, удалять откуда-то из центра совершенно неудобно. Поэтому ищем подходящий лист, делаем swap с вершиной, которую мы хотим удалить и удаляем уже в листе. Проблема: может получиться так, что в вершине ничего нет, а в нашем случае это недопустимо (дерево должно быть сбалансированным). Будем чинить вершину и подниматься вверх (потому что мог сломаться родитель), пока не дойдем до корня. Есть четыре случая, три их них простые.

1. Есть сын 3-вершина. Тогда заимствуем у него один (вместе с поддеревом).
2. Есть предок 3-вершина. Тогда заимствуем один у него (снова с поддеревом).
3. Мы стали корнем. Удалим себя.
4. Поблизости нет никаких 3-вершин. Объеденим себя с корнем и пропушим вверх как в insert.

Более детально и с картинками можете почитать здесь: http://www-bcf.usc.edu/~dkempe/CS104/11-19.pdf. Осторожно, в 3 случае там опечатка, должно быть only one вместо two.

Лекция по алгоритмам #7
Дерево отрезков.

26 февраля

7.1. Еще немножко о деревьях поиска.

Раньше, если нам хотелось посчитать какую-нибудь функцию на отрезке значений, лежащих в дереве поиска, мы умели вырезать этот отрезок при помощи двух операций split, брать значение функции в корне и склеивать дерево двумя merge'ами обратно. У этого способа много проблем — split и merge возможны далеко не всегда, к тому же, 4 вызова этих функций ради одного отрезка — дорогое удовольствие.

Напишем простой код для этой задачи, считая, что функция — сумма на отрезке, а потом поймем, почему он хорошо работает:

```
1 int get (Node v, Query q) {
2   if (q in v)
3     return v.sum;
4   if (q not in v)
5     return 0;
6   return  get(v.l, q) + v.value + get(v.r, q);
7 }
```

Если вершина целиком в запросе или вне него — сразу вернем ответ. Если же она пересекается с запросом, просто вызовемся от ее детей.

Рассмотрим все вершины, лежащие на одной глубине. Если рассматривать их слева направо, то они лежат так:

1. вне запроса
2. пересекаются с запросом (не более 1 вершины)
3. в запросе
4. пересекаются с запросом (не более 1 вершины)
5. вне запроса

На нулевой глубине лежит ровно одна вершина — корень. Из написанного выше списка видно, что на каждом следующем уровне будет задействованно не более 4 вершин (новые вызовы порождаются не более чем 2 вершинами). Итого, если глубина дерева h, суммарно запрос отработает за $\mathcal{O}(4h) = \mathcal{O}(h)$.

7.2. Самые обычные деревья отрезков.

Дерево отрезков — структура данных, строящаяся над массивом или его подобием и позволяющая делать кучу всяких полезных вещей. Например, считать на отрезке значение любой ассоциативной функции.

Что оно вообще из себя представляет? Будем считать, что мы работаем с массивом $a[]$, $size(a) = 2^k = n$, если это не так — добьем до степени двойки нейтральными элементами функции, которую хотим считать. Для дерева отрезков понадобится второй массив $tree[]$, $size(tree) = 2n$. В нем вершины $[n..2n)$ будут листьями и будут соответствовать оригинальному

массиву. Как и в бинарной куче, для вершины с индексом i детьми будут $2i$ и $2i+1$. Родителем, соответственно, будет $\frac{i}{2}$. Каждая вершина отвечает за отрезок листьев, находящихся в ее поддереве.

Приведем код построения ДО (опять функцией для простоту выберем сумму, а элементами — целые числа):

```
void init(int* a, int* tree, int n) {
  tree = new int[2 * n];
  copy(a, a + n, tree + n);
  for(i = n - 1; i > 0; --i) {
    tree[i] = tree[2 * i] + tree[2 * i + 1];
  }
}
```

Построить дерево смогли, как делать в нем запросы на отрезке, мы уже знаем. Теперь хочется поменять какой-нибудь элемент. Для изменения отдельного элемента есть два подхода — сверху и снизу. Сверху — запускаемся от корня, спускаемся в лист, изменяем в нем значение и на обратном ходе рекурсии пересчитываем значение в вершинах. Снизу — поднимаемся (без рекурсии, просто в цикле) от листа к корню и пересчитываем значения в вершинах. Второй способ короче с точки зрения кода, побыстрее работает, но не позволяет делать операций изменения на отрезке, что немаловажно.

Сверху (вершина v отвечает за отрезок $[L..R)$, хотим в pos записать x):

```
void change(int v, int L, int R, int pos, int x) {
  if (L + 1 == R) {
    tree[v] = x;
    return;
  }
  int middle = (L + R) / 2;
  if (pos < middle)
    change(2 * v, L, middle, pos, x);
  else
    change(2 * v + 1, middle, R, pos, x);
  tree[v] = tree[2 * v] + tree[2 * v + 1];
}
```

Снизу:

```
void change(int pos, int x) {
  tree[pos] = x;
  pos /= 2;
  while (pos > 0) {
    tree[pos] = tree[2 * pos] + tree[2 * pos + 1];
    pos /= 2;
  }
}
```

get сверху уже описан сверху (sic!). Напишем get снизу (те же преимущества, что и у изменения снизу):

```
int get(int L, int R) {
  int sum = 0;
  for(L += n, R += n; L <= R; L /= 2, R /= 2) {
    if (L $%$ 2 == 1) sum += tree[L++];
    if (R $%$ 2 == 1) sum += tree[R--];
```

```
  }
  return sum;
}
```

На этом обычные деревья отрезков заканчиваются и начинаются интересности.

7.3 Динамическое ДО.

Пока что массив был достаточно коротким, чтобы помещаться в память целиком. А что, если хотим работать с массивом длины 10^9 или 10^{18}, в котором не очень много элементов, отличных от нейтрального? Опять появляются два подхода.

Если решаем задачу offline, то можно сжать координаты — отсортировать все позиции, которые будут встречаться в запросах и работать только с ними.

Если решаем задачу online, то на помощь приходит динамическое дерево отрезков. Изначально в нем есть только корень, отвечающий за весь массив. Когда поступают запросы, мы при необходимости создаем новые вершины. Т.е., если мы хотим пойти в вершину, которой нет, создаем ее. Теперь дерево приходится писать не на массиве, а на указателях. Время работы тоже меняется, теперь это не $\mathcal{O}(\log n)$, а $\mathcal{O}(\log C)$, где C — максимальное значение, которое нам может встретиться.

С динамическим ДО, в принципе, можно работать снизу, но это не особо интересно даже в теории, разве что как игра ума.

7.4 Персистентное ДО.

Не отличается от других персистентных структур. Вместо изменения вершины, создаем ее копию и работаем с ней. Теперь уж точно приходится писать ДО на указателях и работать с ним сверху. Позволяет реализовать персистентный массив, а он в свою очередь — персистентное много что.

В общем случае код функции будет выглядеть как-то так:

```
Node* foo(Node* u, ..) {
  Node *v = new Node(u);
  ...
  if (...)
    v->left = foo(v->left, ..);
  else
    v->right = foo(v->right, ..)
  ...
  return v;
}
```

7.5 Многомерные ДО.

Пока что в вершинах лежали какие-то значения. Но этим можно не ограничиваться — в них можно сохранить все что угодно. Если сохранить в них какую-нибудь другую структуру данных, например еще одно ДО или ДД, то получится двумерное ДО. Вложенность можно делать какую угодно, главное понимать зачем.

Для примера сохраним в вершине отсортированный массив значений ее отрезка. Строится такое ДО точно так же, как и обычное, благодаря встроенному merge:

```
1 merge(tree[2 * i].begin(), tree[2 * i].end(), tree[2 * i + 1].begin(), tree[2 * i
      + 1].end(), tree[i].begin());
```

Такое дерево может, например, считать количество чисел со значениями из $[A..B]$ на отрезке индексов $[L..R)$. Для этого приходя в вершину считаем искомое число двумя бинпоисками на ее массиве.

Лекция по алгоритмам #8
Дерево отрезков

26 февраля

8.1. ДО сортированных массивов

8.1.1. Задача 1

Дана последовательность $a_1, a_2, \cdots, a_n$
online Q запросов: $\langle L_j, R_j, x_j, y_j \rangle$, посчитать число таких i, что
$L_j \le i \le R_j, \ x_j \le a_i \le y_j$

Замечаем что в ДО можно хранить произвольную структуру данных (Декартово дерево, Дерево отрезков, Sparse Table $\cdots$).

В нашем случае храним ДО сортированных массивов. Постоение за $\mathcal{O}(n \log n)$: во время built c

Ответ на запрос: для всех вершинах ДО, затротронутых запросом делаем:

```
1    $ans += upper_bound(all(T[v]), x_j) - lower_bound(all(T[v]), x);
```

$\mathcal{O}(n \log^2 n)$

К-порядковая статистика $\mathcal{O}(n \log^3 n)$: бинарный по поиск ответу в задаче 1.

8.2. Scanline

8.2.1. Задача 2

Offline.
Даны прямогольники.
Q запросов посчитать, скольки прямоугольникам принадлежит точка.
Разобьём запросы и прямоугольники на события:

1. Прибавить 1 на полосе.

2. Вычесть 1 на полосе.

3. Вычислить значение в точке.

Отсортируем события одной координате. Проходим по событиям в порядке сортировки. Запросы 1, 2 – прибавть на отрезке, 3 – посчитать сумму в точке.

8.2.2. Другое решение задачи 1

Offline.
Геометрически задача 1 выглядит так: Q запросов посчитать число точек $\langle a_i, i \rangle$ на прямоугольнике.
Разобьём запросы и точки на события:

1. Посчитать сумму на полосе, и добавить к ответу на запрос.

2. Посчитать сумму на полосе, и вычесть из ответа на запрос.

3. Прибавить 1 в точке.

Отсортируем события одной координате. Проходим по событиям в порядке сортировки. Запросы 1, 2 – посчитать сумму на отрезке, 3 – прибавить в точке.

Online: оставим только события типа 3, сделаем scanline персистеным.

Уже умеем задачу 1 за $\mathcal{O}(n \log n) \implies$ K-порядковую статистику на отрезке научились искать за $\mathcal{O}(n \log^2 n)$.

8.3. K-порядковая статистика за $\mathcal{O}(n \log n)$

Избавимся в решении за $\mathcal{O}(n \log^2 n)$ от бинарного поиска с помощью параллельного спуска. Обратитмся к версия соответствующим l - 1 и r и будем параллельно в их спускаться аналогично поиску K-ой единицы с помощью ДО.

```
1     int get(int v1, int v2, int tl, int tr, int k) {
2         int tm = (tl + tr) / 2;
3         if (tr - tl <= 1) return tl;
4         int lCnt = T[T[v2].l].cnt - T[T[v1].l].cnt;
5         if (lCnt > k) {
6             return get(T[v1].l, T[v2].l, tl, tm, k);
7         } else {
8             return get(T[v1].r, T[v2].r, tm, tr, k - lCnt);
9         }
10    }
```

Лекция по алгоритмам #9
Rope

20 марта

9.1. Определение

Rope – интерфейс для структуры данных, позволяющий резать строку на подстроки и склеивать их в другом порядке.

Как мы умеем решать эту задачу?
Заметим, что строку можно интерпретировать как массив, на массиве умеем строить дерево, а значит через операции Split и Merge мы можем осуществить то, что нам нужно. Т.е. просплитить в нужном месте, а затем склеить в другом порядке.

Split и Merge мы умеем делать в следующих структурах:

1. Treap. Декартово дерево по неявному ключу.
2. Через Splay. Ниже будет приведено описание Split и Merge.
3. Через Skip-List.
4. Также утверждается, что в AVL и 2-3 Tree также можно реализовать Split и Merge.

9.2. Через Splay

9.2.1. Split

Пусть мы хотим сделать split по элементу x. Можно найти его $\text{lowerbound}(x)$ и вытолкнуть его в корень. Второй способ это добавить его в дерево, он автоматически будет корнем в новом дереве, тогда левое и правое поддеревья – есть ответ функции Split. Понятно, что оба способа работают за логарифм.

9.2.2. Merge

Нам даны дерево L и R. Считаем, что они оба корректны. Найдем самый правый элемент в левом поддереве и поднимем его в корень. Это мы сделали, чтобы операция Merge вообще просто работала за $O(1)$, проводя ссылку в корень дерева R. Таким образом это тоже работает за $O(\log n)$.

9.3. Skip-List

Рандомизированная структура данных. Не является деревом, но умеет делать такие же операции со строками.
Собственно, что это такое: пусть мы храним отсортированный массив как список. Сделаем второй уровень: каждый элемент на втором уровне окажется с вероятностью $\frac{1}{2}$. Аналогично сделаем на последующих уровнях. Если высота уровня равна m, то вероятность, что конкретный элемент там окажется – $\frac{1}{2^m}$. Для удобства реализации, можно договориться, чтобы левый элемент всегда переходил на вверх.

Сколько мы делаем уровней? Для понимания условимся, что будем хранить $\log n$ уровней, однако утверждается, что если мы будем переходить на следующий уровень, пока туда хоть кто-то может перейти – такая версия также будет быстро работать.

9.3.1. Find

В принципе, можно идти втупую, но давайте лучше делать это умнее.
Предполагаем, что x есть в нашем Skip-List

```
1 Find(x)
2   v = LeftTop
3     while (v->x != x)
4       if (v->r->x <= x)
5           v = v->r
6         else
7           v = v->down
```

Мега объяснение почему это работает за $O(\log n)$.
Интуитивное понимание: это очень похоже на бинарный поиск.
Сколько раз мы можем перейти вниз? Понятно, что не более логарифма. Посчитаем матожидание от количества шашов на текущем уровне. Сколько раз мы можем перейти направо? Вероятность того, что на текущем уровне мы сделали не более одного шага – $\frac{1}{2}$. Вероятность того, что на текущем уровне сделали не более двух шагов – $\frac{1}{4}$ и так далее. Пусть мы сделали k шагов. Т.е. вероятность k-ого шага равна $\frac{1}{2^k}$. Тогда матожидание будет: $E = \sum \frac{k}{2^k}$. Как мы знаем из матана, этот ряд сходится и он равен $O(1)$. Значит, Find работает за $O(1)$. Теперь строже стало.

9.3.2. Add

С помощью Find умеем найти место куда вставить x уже за $O(\log n)$. Вызовем его. Запомним позиции на каждом уровне такие, на которых Find был последний раз, т.е. прежде чем спуститься вниз. Т.е. $O(\log n)$ позиций. На самую нижнюю из них мы вставим x. На втором уровне вставим его с вероятностью $\frac{1}{2}$. Если мы вставили его на втором уровне, вставим его на третий уровень с вероятностью $\frac{1}{4}$. И так далее.

9.3.3. Delete

Аналогично запускаем Find и выпиливаем нужный "столбец".

9.3.4. Split

Через Find также находим столбец, обрубаем все связи на нем. Если в реализации предусмотрен фиктивный столбец слева(для удобства), добавляем его в получившуюся правую часть.

9.3.5. Merge

Удаляем фиктивный столбец в правой части. Далее просто проводим ссылки.

9.4. Offline персистентность

Вообще, далее на лекции были несколько задач и их разбор. Я напишу про 2 из них, если в будущем что-то еще появится в билетах, этот раздел дополнится.

Задача: даны запросы к СНМ вида, обратиться в i-ой версии и, например, объединить элементы a и b, тем самым создав новую версию, или просто проверить, лежат ли a и b в одном множестве в i-ой версии СНМ. Это запросы первого и второго типа соответственно.

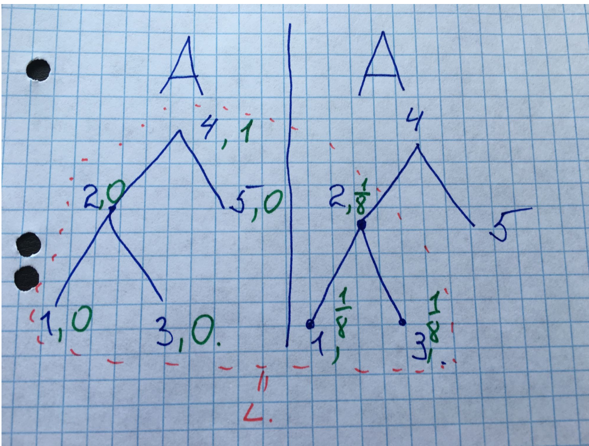
Решим в оффлайн за $O(n+m)$. (n – количество элементов в СНМ, m – количество запросов).

Решение: Всегда, когда мы говорим о персистентной структуре данных, образуется некое дерево версий, где корень – изначальная структура данных. В такой интерпретации понятно, что запрос изменения к структуре – порождает нового сына в дереве. То же верно и для persistent DSU. Построим дерево версий. В потом запустим DFS обход из корня вершины. Когда мы заходим в произвольную вершину, обрабатываем все запросы второго типа, поскольку, очевидно, что эти запросы производятся к уже существующей версии. Потом рекурсивно вызываем DFS от сына – следующей версии. Но перед тем как это сделать, нужно запомнить состояние в котором были, чтобы после обхода очередного сына – откатить все назад. Таким образом все работает за нужную асимптотику.

9.5. Persistent RBST

Def 9.5.1. *RBST(Randomized Binary Search Tree) – BST, в котором каждая вершина, может стать корень равновероятно* $\Leftrightarrow$ *наша версия без y, а именно y-ки использовались в merge, теперь там строчка :* $p = \frac{L.size()}{L.size()+R.size()}$. *С вероятностью p, корнем merge будет корень левого поддерева, иначе будет корень правого поддерева.*

Мы хотим сделать его персистентным, будем каждый раз создавать новое дерево при любом изменении. Однако, если – дерево, то merge(A, A), выдаст дерево, по определению не являющимся RBST.



Пусть 4 в правом дереве , будет корнем результирующего дерева(фиксируем ее), тогда элементы левого дерева и левого поддерева 4, окажутся в левом поддереве результирующего дерева. По определению, вершина левого поддерева должна выбираться равновероятно. Смотрим, на левое поддерево 4. Каждая вершина выбирается с вероятностью $\frac{1}{3}$, т.к. это корректное произвольное BST. Значит при выборе корня в левом поддереве, каждая вершина из этого поддерева возьмется с вероятностью равно $\frac{3}{8} \cdot \frac{1}{3} = \frac{1}{8}$. Значит, с этим поддеревом все понятно. Теперь посмотрим на левое поддерево . Здесь должно было бы получиться аналогично: каждая вершина должна была бы выбираться с вероятностью $\frac{1}{8}$, но на самом деле корнем будет 4 в левом дерева . Всегда. Поскольку мы зафиксировали 4 в правом дереве и эта 4 всегда будет отцом, потому что деревья одинаковые. Т.е. 4 возьмется с вероятностью 1, а все остальные вершины будут с вероятностью 0. Т.е. получили, что по нашему определению, это не будет произвольным RBST. Однако утверждается, что при таком merge мы все равно получим RBST(без док-ва).

Лекция по алгоритмам #10
Фарах-Колтон и Бендер

2 марта

10.1. Что уже есть

Мы умеем решать задачу поиска минимума на отрезке (RMQ) с помощью дерева оторезков и получать следующую ассимптотику:

1. `init` – построение, $\mathcal{O}(n)$
2. `get(L, R)` – минимум на отрезке [L, R], $\mathcal{O}(\log n)$
3. `change(i, X)` – изменить i-й элемент массива на X, $\mathcal{O}(\log n)$

Теорема 10.1.1. *Lower Bound*
Нельзя сделать get, change за $o(\log n)$.

Доказательство. Пусть можно. Тогда мы получили сортировку за $o(n \log n)$. ■

10.2. Sparse Table

Это структура данных, умеющая init за $\mathcal{O}(n \log n)$ и get за $\mathcal{O}(1)$, но не умеющая change. Пусть был исходный массив a. Тогда от каждой точки i для каждой длины 2^k будем хранить минимум $m[k, i]$. Посчитать его можно динамикой:

$$\begin{cases} m[0, i] = a[i] \\ m[k, i] = min(m[k-1, i], m[k-1, i+2^{k-1}]) \end{cases}$$

get будет покрывать [L, R] двумя отрезками, для которых посчитаны минимумы. Для этого просто берётся максимальная не превосходящая R-L+1 степень двойки. Один отрезок такой длины начинается в точке L, второй отрезок такой длины заканчивается в точке R. Наименьший из минимумов на этих отрезках и есть минимум на [L, R].

get(L, R) $= min(m[k, L], m[k, R-2^k+1])$, где предподсчитанное $k : 2^k \le R - L + 1 < 2^{k+1}$

10.2.1. Super Sparse

Соединим Sparse Table с Деревом Отрезков, чтобы стало ещё быстрее. Две модификации:

1. init за $\mathcal{O}(n \log \log n)$, get за $\mathcal{O}(1)$
2. init за $\mathcal{O}(n)$, get за $\mathcal{O}(\log \log n)$

В обоих случаях сначала разобьём исходный массив a на кусочки длины $\log n$ и заведём массив b – минимумы на этих кусочках. Размер массива b равен $k := \lceil n / \log n \rceil$. Теперь построим Sparse Table от b. Это займёт $\mathcal{O}(k \log k) = \mathcal{O}(n)$ времени. Осталось только научиться обрабатывать частичные префиксы/суффиксу – расстояния от L/R до ближайшей внутренней границы кусочка (или от L до R, если между ними нет границы). Обозначим минимум на префиксах как $p[i]$ (аналогичный массив s для суффиксов), где i – L или R соответственно.

$$\begin{cases} p[i] = +\infty, \text{ если } i \text{ – граница} \\ p[i] = min(a[i], p[i+1]), \text{ если } i \text{ – не граница} \end{cases}$$

Если между L и R есть граница – позвращаем $min(min$ из Sparse Table от $b, p[L], s[R])$. Иначе нужно возвращать минимум напрямую на [L, R]. Для этого придётся заранее на каждом кусочке завести структуру данных, которая будет быстро выдавать минимум на [L, R], если тот оказался внутри этого кусочка. Это может быть либо Дерево Отрезков, влекущее get за $\mathcal{O}(\log\log n)$, либо Sparse Table, влекущее init за $\mathcal{O}(n\log\log n)$.

10.3. алгоритм Фараха-Колтона и Бендера

10.3.1. Анонс

Теперь научимся решать RMQ с init за $\mathcal{O}(n)$ и get за $\mathcal{O}(1)$. Для этого сведём RMQ к LCA (поиск наименьшего общего предка в подвешенном за корень дереве), а её – к RMQ±1 (частный случай RMQ, в котором соседние числа отличаются ровно на 1). Каждое сведение будет за $\mathcal{O}(n)$ предподсчёта, а RMQ±1 уже научимся решать с init за $\mathcal{O}(n)$ и get за $\mathcal{O}(1)$.

Стоит упомянуть, что приведённая здесь версия на практике не лучше, чем Super Sparse через Дерево Отрезков. Однако существует более новая версия этого алгоритма, использующая $\mathcal{O}(n)$ бит памяти.

10.3.2. RMQ → LCA

Вспомним Декартово Дерево (Cartesian Tree). Давайте построим его от нашего массива a. Пары <значение, ключ> ($<x,\ y>$) в дереве – пары $<a[i],\ i>$. Внезапно оказывается, что если составить массив пар, где i-я пара – $<a[i],\ i>$, то в нём элементы будут возрастать по ключу. Осталось только научиться строить Декартово Дерево от такого массива за $\mathcal{O}(n)$.

Строим дерево последовательно и со стеком. На k-м шаге у нас есть уже готовое дерево от первых k элементов, причём мы помним, где из них самый правый, и в стеке храним путь от корня до этого самого правого элемента. Поскольку путь до самого правого элемента идёт строго направо, наш стек – это возрастающий массив. Как же сделать следующий шаг и вставить новый элемент?

Для начала, в качестве корня выберем фиктивную минус бесконечность. Кладём её на стек. В дальнейшем будем подвешивать к ней. Пусть теперь мы хотим добавить новую вершину $<a[k],\ k>$. Последовательно убираем вершины со стека (запоминая последнюю убранную), пока значение верхней из оставшихся на стеке больше, чем $a[k]$. Фактически, мы поднимаемся по дереву. При этом стек не окажется пуст, потому что фиктивный корень всегда меньше. Когда остановились – удаляем ребро между верхней на стеке вершиной и той, которую убрали последней. Заметим, что это ребро "вправо". На место этого ребра "вклиниваем" нашу новую вершину, то есть подвешиваем $<a[k],\ k>$ к верхней на стеке вершине как правого потомка и добавляем $<a[k],\ k>$ на стек, а потом, если есть последняя убранная со стека вершина, подаешиваем её к $<a[k],\ k>$ как левого потомка.

<u>Lm</u> 10.3.1. $\mathcal{O}(n)$.

Доказательство. Каждую вершину положили на стек 1 раз, а убрали – не больше. ■

<u>Lm</u> 10.3.2. Корректность дерева.

Доказательство. Старые элементы не поменяли свой порядок (левее/правее) друг относительно друга, а новый – правее всех и больше всех по изначальному условию. ■

Lm 10.3.3. Равносильность RMQ и LCA.

Доказательство. В получившемся дереве каждая вершина соответствует отрезку, причём корень соответствует всему массиву. При спуске от корня к L и к R последняя общая вершина – последняя вершина, соответствующая отрезку, содержащему весь [L, R]. При этом значения соответствующих отрезкам вершин – минимальные элементы исходного массива на этих отрезках. Значит, ответ на RMQ – наименьший общий предок, то есть LCA. ■

10.3.3. LCA → RMQ±1

Построим от дерева с корнем Эйлеров обход – упорядоченный набор вершин, соответствующий циклу по всему дереву, проходящему по каждому ребру в каждую сторону по одному разу. Если несколько видов такого обхода. Точнее, несколько способов хранить его в массиве.

1. Последовательно сохранять номера всех посещённых вершин.
2. Последовательно сохранять номера всех посещённых впервые вершин (просто dfs).
3. Последовательно сохранять все посещённые ориентированные рёбра, предварительно разделив каждое неориентированное ребро на 2 ориентированных в разные стороны.

Второй способ интересен тем, что поддеревья в дереве являются отрезками в массиве обхода, то есть вершины любого поддерева расположены в массиве обхода без "дырок" из других вершин.

Но в данной задаче нам нужен первый способ. Если при обходе всегда поворачивать в самого левого из непосещённых потомка (если таковые есть), а вместо самих вершин сохранять их глубину ($height[v]$), то мы получим всё ещё однозначно задающий дерево массив, соответствующий "условию ± 1" – разность соседних элементов равна 1.

Lm 10.3.4. RMQ±1 на Эйлеровом обходе через $height$ = LCA на исходном дереве.

Доказательство. Договоримся, что в запросе lca(v, u) v не правее u. Почему LCA на отрезке [v, u] в $height$? Потому что чтобы прийти из v в u нужно обязательно пройти через какого-то общего предка, а для этого нужно пройти через ближайшего общего предка. Почему неит других общих предков? Потому что при переходе в ближайшего общего предка по пути из v алгоритм бы не поднялся, если бы не были проверены все дети, а значит и все потомки потомки LCA (в том числе и u), поэтому, если он вернулся в другого общего предка, то u алгоритм уже посетил последний раз, из чего следует, что все остальные общие предки лежат вне отрезка [v, u]. Почему LCA – минимум? Потому что алгоритм не поднимется выше LCA ни до него, ни после, ведь иначе LCA – не общий предок.

Проще говоря, это правда, потому что наш обход – честный и порядочный dfs. ■

10.3.4. RMQ±1

?

Лекция по алгоритмам #11
RMQ ±1, LCA

?? ??

11.1. RMQ ±1

$[\mathcal{O}(n), \mathcal{O}(1)]$

1. Разделяем массив на куски по $\frac{1}{2}\log n$. По минимумам на каждом куске строим $SparseTable$.

2. Если при запросе L, R попадают в разные блоки, берем минимум из блоков, попавших целиком (тут поможет $SparseTable$), осталось только рассмотреть случай, когда L, R - в одном и том же блоке.

3. Заметим, что вариантов того, как может быть устроен этот маленький блок не так уж и много, а именно не больше, чем $2^{(\frac{1}{2}\log n)} = \sqrt{n}$. Здесь мы пользуемся тем, что соседние элементы отличаются на ±1, поэтому можно массив заменить на массив из плюс-минус-единиц. Конечно, потому что нам важно только расположение минимума в данном случае.

4. Мало вариантов внутреннего устройства подотрезка, поэтому мы делаем предподсчет, тем самым добиваясь ответа на вопрос за $\mathcal{O}(1)$.

5. На лекции предлагалось в реализации хранить подотрезки по маске, заминив `+1` на `0`, `-1` на `1`, и делать запрос к предподсчету по элементу $f[(x_i \gg L) \& (2^{R-L} - 1)]$. Последняя часть, чтобы откинуть лишние биты. Утверждается, что это можно сделать и это даже будет работать для префиксов, в чем не трудно убедиться самостоятельно.

11.2. LCA

`Способы решать задачу:`

1. Сводим к $RMQ, SparseTable$. Online $[\mathcal{O}(n\log n), \mathcal{O}(1)]$, offline $[\mathcal{O}(n), \mathcal{O}(\log n)]$

2. Двоичные подъемы. $[\mathcal{O}(n\log n)]$

   $go[k, v] = go[k-1, go[k-1, v]]$

   $go[0, v] = parent[v]$

   $parent[root] = root$

   $dep[v]$ - глубины вершин.

   Собственно, функция LCA (работает за $\mathcal{O}(\log n)$):

```
1
2     LCA(a,b) {
3         if(dep[a] > dep[b])
4             Jump(a, dep[a] - dep[b])
5         else
6             Jump(b, dep[b] - dep[a])
```

```
7
8        FOR k = log2(n) .. 0
9            if (go[a,k] != go[b,k])
10               a = go[a,k]
11               b = go[b,k]
12
13       return a == b ? a : parent[a]
14   }
```

3. Времена входа-выхода.

```
1
2        // O(n)
3        dfs(v) {
4            in[v] = T++
5            for(int x : g[v])
6                dfs(x)
7            out[v] = T++
8        }
9
10       // O(1)
11       bool is_ancestor(int a, int b) {
12           return in[a] <= in[b] && out[a] >= in[b];
13       }
14
15       int LCA2(int a, int b) {
16           for k = log2(n) .. 0
17               if (!is_ancestor(go[a,k], b])
18                   a = go[a,k];
19               return is_ancestor(a,b) ? a : parent[a];
20       }
```

Лекция по алгоритмам #12
LA
11 марта

12.1. LCA-Offline. Тарьян

Мы хотим обрабатывать запросы "найти наименьшего общего предка"в offline. Для этого для каждой вершины храним все связанные с ней запросы. Делаем dfs по дереву с СНМ. Когда выходим из вершины, join-им ее с отцом. Для каждой компоненты в СНМ храним самого высокого представителя. Пусть мы обрабатываем вершину a. Переберем все запросы, связанные с ней. Если вторая вершина b из запроса еще не посещена, пропустим этот запрос. Иначе ответ на запрос - это сама высокая вершина-предок b, из которой мы еще не вышли, т.е. самый высокий представитель в компоненте b.

```
1    void dfs(int v, int p) {
2      //обработка запросов
3      u[v] = true;
4      for (int x: g[v])
5        dfs(x, v);
6      join(v, p);
7    }
```

Работает за $\mathcal{O}((n+m)A^{-1}(n)$.

12.2. LA (Level Ancestor)

Мы хотим уметь подниматься от вершины v на k вверх. $LA[v][k] = LA[p[v]][k-1]$
Различные решения:

1. **LA Offline** $\mathcal{O}(n+m)$

   Запустим dfs, будем хранить массив вершин на пути от вершины до корня. Для каждой вершины сохраним все связанные с ней запросы. Ответ на запрос - k-ая с конца вершина в массиве.

2. **Двоичные подъемы** $[\mathcal{O}(n\log n),\ \mathcal{O}(\log n)]$

3. **Алгортм Вишкина**

   Храним эйлеров обход, высота соседних вершин отличается на ± 1. Пусть $last[v]$ – последнее вхождение v в эйлеров обход. Тогда ответом на запрос будет первая вершина u на суффиксе, начиная с $last[v]$, у которой высота $d[u]$ равна $d[v]-k$. Это мы умеем считать деревом отрезков. Работает за $[\mathcal{O}(n), \mathcal{O}(\log n)]$. В качестве упражнения оставлено, что методом 4-х русских можно превратить $\mathcal{O}(\log n)$ в $\mathcal{O}(1)$ за счет того, что высоты соседних вершин отличаются на ± 1.

4. **Ladder-Decomposition** $[\mathcal{O}(n), \mathcal{O}(1)]$

12.3. Ladder-Decomposition

Рассмотрим подробнее последний способ. Предподсчет будет состоять из трех шагов.

- **Longest-Path-Decomposition**

  Разобьем все вершины на пути следующим образом. Обойдем дерево dfs-ом. Пусть мы стоим в вершине v. Обойдем всех ее детей, добавим v в путь, идущий в самое глубокое поддерево, т.е. в котором находится вершина с самой большой глубиной.

  Для каждой вершины сохраним номер пути, в который она входит.

- **Ladder-Decomposition**

  Увеличим каждый путь в два раза вверх. Для каждого нового пути сохраним все входящие в него вершины. Для каждой вершины сохраним ее позицию в пути с тем номером, который мы для нее сохранили до этого.

- **Двоичные подъемы**

  Насчитаем обычные двоичные подъемы $p_k[v]$.

Пусть нам пришел запрос найти $LA(v, k)$.

1) Сделаем один максимально большой прыжок вверх с помощью насчитанных двоичных подъемов.

   $i = [\log k]$

   $v = p_i[v]$

2) Рассмотрим путь, на котором лежит вершина v до удвоения. Он длины хотя бы 2^i, так как мы точно знаем, что существует вершина-потомок v, расстояние до которой ровно 2^i (это вершина, из которой мы только что пришли). Значит, после удвоения этот путь стал длины хотя бы 2^{i+1}, причем хотя бы 2^i вершин в нем – предки v. Это означает, что вершина, которую мы ищем, находится на этом пути (иначе бы мы могли до этого прыгнуть еще на 2^i вверх). Так как мы знаем позицию v в этом пути, то нужную вершину мы можем найти за $\mathcal{O}(1)$.

Таким образом, наш алгоритм работает за $[\mathcal{O}(n \log n), \mathcal{O}(1)]$. Утверждается, что с помощью алгоритма 4-х русских мы можем получить $[\mathcal{O}(n), \mathcal{O}(1)]$.

Лекция по алгоритмам #13
Euler-tour/HLD/Link-cut

11 марта

13.1. Euler-Tour-Tree

Решаем задачу "добавлять и удалять рёбра, проверять связность вершин, в каждый момент времени граф не содержит циклов". Храним в декартовом дереве по неявному ключу эйлеров обход в виде последовательности из $2(n-1)$ ребра. Для каждой компоненты отдельное дерево.

1. Link(a, b):
   1. делаем a и b корнями соответствующих деревьев
   2. Merge(Tree(a), $a \to b$, Tree(b), $b \to a$)
2. Cut(a, b):
   1. у ребра $a \to b$ есть два вхождения в ДД — удалим их и вырежем то, что было между ними. В результате получаем поддерево под ребром
   2. соединяем префикс и суффикс — получили два дерева
3. isConnected(a, b) $\Leftrightarrow$ root(a) $==$ root(b). Храним у вершин дерева предков: теперь остается подняться до корня за $\log n$ и проверить

13.2. HLD – Heavy Light Decomposition

Решаем задачу "давать ответы на запросы на пути в дереве".

1. Разбиваем подвешенное дерево на пути. Путь строим в сторону наибольшего поддерева
2. На каждом пути строим дерево отрезков. Также про каждую вершину нужно знать, на каком она пути находится и ее позицию на этом пути. Еще нужно насчитать для каждого пути знать путь-предка

Ответ на запрос:

1. Разбиваем запрос на два вертикальных пути при помощи LCA
2. Для каждого выполняем такую последовательность действий: пока не в корне, делаем запрос к ДО текущего пути, а затем поднимаемся в путь-предка

Время: $\mathcal{O}(k \log n)$, где k — число переходов между путями. Оценим их число:
Смена пути эквивалентна проходу по легкому ребру. Значит, размер поддерева вершины, в которую мы переходим, как минимум в два раза больше, чем у текущей. Так как массив ограничен n, то число переходов составляет $\mathcal{O}(\log(n))$. По тяжелым ребрам мы никогда не проходим, так как все они пропускаются прыжком в путь-предок. В итоге все обрабатывается за $\log^2(n)$

13.3. Link-Cut Tree (без доказательства времени)

Теперь мы хотим "HLD", который можно менять, удаляя и добавляя ребра. Рассмотрим на примере операции Distance(a, b).

Работаем за $\log^2$. При использовании Splay-деревьев асимптотика улучшается до log.
Путь — декартово дерево по неявному ключу, в котором мы храним вершины.
Вводим операцию Expose(a) — соединяет a с корнем так, что они еще и лежат в одном декартовом дереве. Поднимаемся до корня, как в HLD. По пути отрезаем ответвления и прицепляемся сами. Кроме того, отрежем от получившегося пути всё, что ниже a. Теперь a и корень – два конца одного пути.

Lm 13.3.1. $\sum k_i \leqslant (m+n) \cdot \log n$, где k_i – число прыжков между путями во время i-го Expose.

Выразим операции через Expose:

```
MakeRoot(a) {
    Expose(a);
    Reverse(a, Root(a));
}

Distance(a, b) {
    MakeRoot(a);
    Expose(b);
    return NodesBetween(a, b);
}

Link(a, b) {
    MakeRoot(b);
    parent[b] = a;
}

Cut(a, b) {
    MakeRoot(a);
    Expose(b);
    Split(b, parent[b]);
    parent[b] = -1;
}
```

Лекция по алгоритмам #14
RMQ и LCA

15 марта

14.1. RMQ offline.

`RMQ.offline = RMQ` $\to$ `LCA + LCA.offline`

Решаем задачу на массиве $a[1..n]$
$m_0 = 0$
$m_{i+1} = \texttt{minpos}(m_i..n]$
$a[m_1] \le a[m_2] \le a[m_3] \le \dots$
Храним в СНМ множества $[1..m_1](m_1..m_2](m_2..m_3]\dots$.
СНМ поддерживает две операции: `get` и `join`.
Отсортируем все запросы по правой границе за линейное время.
Получили для каждой правой границы R список левых концов $list[R]$.
Решение:

```
CHM.init()
a[0] = −∞
m.push_back(0)
for R=1..n:
    for L ∈ list[R]:
        ответ для запроса (L,R) = min[CHM.get(L)]
    while a[R] < a[m.back()]:
        CHM.join(m.back(), R), m.pop_back()
    m.push_back(R)
```

При этом “`min[корня множества]`” пересчитывает `CHM.join(a,b)`.

14.2. Сумма на пути дерева.

Веса на ребрах меняются. Храним Эйлеров обход на ребрах, т.ч. при спуске к листьям записываем вес с плюсом, при подъеме — с минусом. Для каждой вершины запомним, когда мы из нее вышли окончательно.

Как теперь с этим работать. Поменять вес на ребре = в ДО на обходе за логарифм поменять два элемента (проход вниз и вверх по ребру). Сумма на вертикальном пути = сумма на отрезке. Сумма на любом другом пути = разбиение на две суммы на вертикальных путях, точнее сумма на пути от A до LCA со знаком минус (ведь мы идем вверх, а не вниз по нему теперь), сложенная с суммой на пути от LCA до B.

Если веса не меняются, то проще будет ткая формула: $S(a, b) = S(a) + S(b) - 2S(LCA)$.

14.3. Рандомизированный MST за линейное время.

Пусть на очередном шаге алгоритма у нас n вершин и m ребер.*Порядок действий:*

1. Сделать 3 шага Борувки: $n- > \frac{n}{8}$. Это делается за $\mathcal{O}(n+m)$.
2. Берем рандомные $\frac{m}{2}$ ребер. Запускаем на них рекурсивный MST($\frac{n}{8}$, $\frac{m}{2}$). Этот MST вернет нам набор A_1 из не более чем $\frac{n}{8} - 1$ ребер, которые он посчитает хорошими, то есть есть вероятность, что они могут попасть в наш остов. Набор B_1 оставшихся ребер плохой, в итоговом остове их точно не будет (это шло без доказательства).
3. Переберем фориком оставшиеся $\frac{m}{2}$ ребер и проверим их. Если ребро может улучшить имеющийся сейчас остов, то есть оно меньше максимума на пути между двумя вершинами, что оно соединяет, то заменяем этот максимум на него и складываем ребро в множество A_2. Искать максимум на пути мы за единицу пока не умеем, но так можно. Изначально считаем, что граф полный и веса "отсутствующих"по факту ребер INF. Если же ребро никак улучшить ситуацию не может, то кидаем в B_2. Утверждение: ребра из B_2 в остов не входят.
4. По итогам получаем два множества кандидатов в остов: A_2 и A_1. Запускаем рекурсивный $MST(\frac{n}{8}, |A_1| + |A_2|)$. Заметим, что резмер множества A_1 и матожидание размера A_2 не больше, чем $\frac{n}{8} - 1$.
5. Сколько же это работает? $T(n,m) \leq (n+m) + T(\frac{n}{8}, \frac{m}{2}) + T(\frac{n}{8}, \frac{n}{4})$. То есть вся эта прелесть работает за $\mathcal{O}(n+m)$.

<u>Lm</u> 14.3.1. Пусть p – вероятность, с которой брали ребро на втором шаге (в описанном выше алгоритме это $\frac{1}{2}$). Тогда матожидание размера множества A_2 не больше, чем $(\frac{1}{p} - 1)(n-1)$.

Доказательство. Представим себя Краскалом. Упорядочим ребра по весу, начинаем их перебирать. Если встречаем ребра, которое надо бы добавить в остов, то с вероятностью p добавляем, а с 1-p пропускаем. Рассмотрим первый такой момент, когда добавили ребро в остов. Посчитаем матожидание количества "бросков монетки"нужное для того, чтобы ребро добавилось. $E_k = 1 + (1-p) \cdot E_k$. Отсюда $E_k = \frac{1}{p}$. Что в себя включает это E_k? Одно ребро из A_1 и $E_k - 1 = \frac{1}{p} - 1$ ребер из A_2, они нас и интересуют.Это мы добавили одноребро в остов. Надо повторить процесс n-1 раз. Отсюда и матожидание $(\frac{1}{p} - 1)(n-1)$. ■

Лекция по алгоритмам #15
Минимум на пути в дереве

15 марта

15.1. Постановка задачи

Для построения минимального остовного дерева за линейное время нужно научиться искать минимум на пути в дереве за $\mathcal{O}(1)$.

Пусть дано дерево из n вершин и m запросов. Хочется ответить на все запросы за $\mathcal{O}(m+n)$ в offline.

15.2. Обзор существующих медленных решений

15.2.1 Centroid decomposition

В каждом поддереве centroid decomposition посчитаем минимумы от каждой вершины до корня (центроида). Для ответа на запрос достаточно узнать LCA в дереве centroid decomposition. Сложность решения — $\mathcal{O}(m \cdot \texttt{LCA} + n \log n)$. Если искать LCA алгоритмом Фарах-Колтона и Бендера, получим $\mathcal{O}(m + n \log n)$.

15.2.2 Двоичные подъёмы

Та же идея, что и при поиске LCA методом двоичных подъёмов. Но для каждой вершины v помимо предка, находящегося на 2^k рёбер выше (`p[k][v]`), подсчитаем ещё и минимум на пути `v` $\rightsquigarrow$ `p[k][v]`. Тогда вместе с `LCA(u, v)` мы сможем подсчитать и минимумы на путях `u` $\rightsquigarrow$ `LCA` и `v` $\rightsquigarrow$ `LCA`. Асимптотика — $\mathcal{O}(m \log n + n)$, online.

15.2.3 dfs с деревом отрезков

Решим задачу в offline. Разобьём каждый запрос на два вертикальных ($\mathcal{O}(m \cdot \texttt{LCA})$). Запомним для каждой вершины, являющейся нижним концом вертикального запроса, все связанные с ней запросы. Будем обходить дерево dfs-м, поддерживая стек весов рёбер на пути до корня. В этом стеке нужно узнавать минимум на куске верхних элементов. Для этого заменим стек на дерево отрезков. Асимптотика — $\mathcal{O}(m \cdot \texttt{LCA} + (m+n) \log n)$, offline.

15.3. Алгоритм с СНМ

Как обычно, разобьём каждый запрос на два вертикальных. Теоретически для этого будем использовать алгоритм Фарах-Колтона и Бендера, а на практике — алгоритм Тарьяна.

В данном алгоритме мы не можем пользоваться СНМ как чёрным ящиком, нужно взаимодействовать со внутренней структурой. В СНМ с эвристикой сжатия путей будем хранить вершины дерева. Каждое множество будет представлять собой связное множество вершин дерева, корнем множества в СНМ всегда будет корень связного множества в дереве. Изначально каждая вершина — отдельное множество. Для каждой вершины `v` будем поддерживать `w[v]` — минимум на пути в дереве от неё до её родителя в СНМ (этот путь будет в дереве вертикальным).

Операция `Get(v)` возвращает минимум на пути в дереве от `v` до представителя (в СНМ) множества, содержащего `v`. Псевдокод приведён ниже. `p[v]` — родитель в СНМ, `w[v]` — минимум на пути `v` ⇝ `p[v]` в дереве.

```
Get(v) {
  if (v == p[v])
    return w[v];
  Get(p[v]); // w[p[v]] has changed
  w[v] = min(w[v], w[p[v]]);
  p[v] = p[p[v]]; // path compression
  return w[v];
}
```

Заметим, пока мы нигде не делали Join, это будет происходить в процессе обработки запросов. Для каждой верхней вершины вертикального запроса запомним все связанные с ней нижние вершины: `ListD[U]`. Отсортируем все вершины снизу вверх (например, в порядке, обратном порядку посещения dfs-м, или по убыванию глубины, главное, чтобы для каждого поддерева корень находился в отсортированном массиве после всех остальных вершин этого поддерева). Разумеется, сортируем за линейное время.

Ниже приведён псевдокод обработки запросов. `Sorted` — отсортированный снизу вверх массив вершин, `parent[v]` — родитель `v` в исходном дереве.

```
for (U in Sorted) {
  for (D in ListD[U]) {
    Answer = get(D);
  }
  p[u] = parent[u]; // Join
}
```

В данном варианте кода предполагается, что взвешены вершины, а не рёбра, и `w[v]` инициализируется весом вершины v.

Если взвешены рёбра, то `w[v]` изначально равно $+\infty$, и в код обработки запросов добавится ещё одна строчка:

```
for (U in Sorted) {
  for (D in ListD[U]) {
    Answer = get(D);
  }
  p[u] = parent[u]; // Join
  w[u] = weight[u];
}
```

`weight[u]` — вес ребра из `u` в `parent[u]` в исходном дереве.

Как же работает этот код? Легко видеть, что сохраняется такой инвариант: в момент, когда мы пришли в вершину `U`, всё поддерево с корнем `U` объединено в одно множество в СНМ, и `U` — представитель этого множества. Поэтому операции `Get(D)` отработают корректно.

Поскольку в СНМ используется лишь одна эвристика, время работы алгоритма — $\mathcal{O}((n+m)\log n)$, но на практике работает очень быстро. Утверждается, что можно получить линейную асимптотику, если добавить ранговую эвристику (решив возникшие при этом проблемы отложенным Join-м) и метод четырёх русских.

Лекция по алгоритмам #16
Минимальное паросочетание

28 сентября

16.1. Определения, постановка задачи

Def 16.1.1. *(M)atching - множество несмежных рёбер.*

Def 16.1.2. *Perfect Matching (совершенное паросочетание) - паросочетание размер которого равен размеру наименьшей доли.*

Def 16.1.3. *(I)ndependent Set - множество попарно неинцедентных вершин.*

Def 16.1.4. *Vertex (C)over - такое множество вершин, что хотя бы один конец каждого ребра в нём лежит.*

Задача, которую будем решать состоит в максимизации M и I и минимизации C.

Lm 16.1.5. $I = V \setminus C,\;\; C = V \setminus I,\;\; \max I = V \setminus \min C$

Замечание. В произвольном графе задача поиска $\max I$ является NP-трудной. Для поиска максимального паросочетания существуют алгоритм сжатия соцветий за $\mathcal{O}(V^3)$ и алгоритм Вазирани за $\mathcal{O}(E\sqrt{V})$.

Lm 16.1.6. Задача поиска максимальной клики эквивалентна задаче поиска $\max I$ в инвертированном графе.

16.2. Двудольные графы

Def 16.2.1. *Чередующийся путь - путь в котором рёбра чередуются в смысле принадлежности паросочетанию.*

Def 16.2.2. *Дополняющий чередующийся путь - чередующийся путь первая и последняя вершина которого не принадлежат паросочетанию.*

Lm 16.2.3. Для того, чтобы паросочетание было совершенным необходимо и достаточно, чтобы не существовало дополняющего чередующегося пути.

Доказательство. Если такой путь существует, то инвертируем все рёбра на нём и получим большее паросочетание.

Докажем теперь, что если такого пути нет, то паросочетание максимальное. Рассмотрим симметрическую разность максимального паросочетание и нашего. Степень каждой вершины в ней не более двух, значит она является объединением циклов и путей. Если есть нечётный путь, в котором первое и последнее ребро принадлежит нашему паросочу, то максимальный можно было бы увеличить по этому пути и вышло бы противоречие. Если есть нечётный путь, в котором первое и последнее ребро принадлежит макс. паросочу, то в нашем графе есть ДЧП, что противоречит предположению.

Значит симм. разность является объединением циклов и чётных путей, а значит в нашем и максимальном паросоче одинаковое количество рёбер, что и требовалось доказать. ■

16.2.1. Алгоритм поиска ДЧП

```
1 dfs(int v)
2   used[v] = 1
3   for(int x : c[v])
4     if mt[x] == -1 or (!u[mt[x]] and dfs(mt[x])):
5       mt[x] = v
6       return 1
7   return 0
```

С помощью этого алгоритма можно искать паросочетание за $\mathcal{O}(V(V+E))$:

```
1 for v/2
2   used <- 0
3   for v
4     if !covered[v]
5       dfs(v)
```

16.2.2. Алгоритм Куна

```
1 for v in V
2   u <- 0
3   if dfs(v)
4     size++
```

Для того, чтобы убедиться в корректности этого алгоритма нужно понять, что если мы один раз не смогли найти ДЧП из одной вершины, то больше никогда не сможем. Если рассмотреть изменение в графе после инвертирования ребёр, то несложно понять, что если ДЧП есть сейчас, то она была и раньше.

Кун*:

```
1 for v in V
2   if dfs(v)
3     size++
```

Для дальнейшего ускорения Куна можно добавить перед его запуском жадник, который найдёт паросочетание с размером не менее половины максимального:

```
1 for e in E
2   if we can add e to M
3     add e to M
```

Лекция по алгоритмам #17
Max independent set, min cover

24 марта

• **Обозначения:**

M – matching, паросочетание (множество попарно не смежных рёбер).

C – cover, вершинное покрытие или контролирующее множество (множество вершин таких, что для любого ребра хотя бы один конец лежит в множестве).

I – independent set, независимое множество (множество вершин таких, что никакие две из них не соединены ребром).

17.1. Алгоритм за $\mathcal{O}(V+E)$

Считаем, что нам уже дано максимальное паросочетание M. Задача: найти максимальное независимое множество, минимальное вершинное покрытие. Запустим на имеющемся графе dfs из всех вершин первой доли, из которых не исходит ни одного ребра максимального паросочетания. Назовём помеченные вершины из первой доли A^+, помеченные вершины из второй доли B^+, а непомеченные соответственно A^- и B^-.

• **Утверждение:** из этих групп мы можем получить всю нужную информацию.

$$\mathbf{I_{max} = A^+ \cup B^-}$$

$$\mathbf{C_{min} = A^- \cup B^+}$$

Часть нужная с точки зрения “закодить задачу” закончилась.

17.2. Корректность алгоритма

Lm 17.2.1. $\max|M| \le \min|C|$

Доказательство. Рассмотрим любые паросочетание M, вершинное покрытие C.
У каждого ребра M один конец должен лежать в C, получили $\forall M, C\colon |M| \le |C|$. ■

Lm 17.2.2. $A^+ \cup B^-$ – независимое, $A^- \cup B^+$ – cover

Доказательство. Независимость $A^+ \cup B^-$: пусть есть ребро между $a \in A^+$ и $b \in B^-$, тогда `dfs` бы прошёл по нему, получается $b \in B^+$. $A^- \cup B^+$ – cover, как дополнение независимого. ■

Lm 17.2.3. $A^- \cup B^+ = |M|$

Доказательство.
Вершина из A^- – конец ребра паросочетания.
Вершина из B^+ – конец ребра паросочетания.
Между A^- и B^+ нет ребер из паросочетания: ребро паросочетания или $\langle A^+, B^+\rangle$, или $\langle A^-, B^-\rangle$.
Получили $A^- \cup B^+ \le |M|$. Поскольку $A^- \cup B^+$ – cover, применяем 17.2.1, получаем равенство. ■

Теорема 17.2.4. *Кёнига.* $\max|M| = \min|C|$

Доказательство. В лемме 17.2.3 мы предъявили вершинное покрытие равный максимальному паросочетанию. По 17.2.1 это вершинное покрытие минимально. ■

- **Следствие:** заодно доказали, что $A^- \cup B^+$ – минимальное вершинное покрытие.
- **Следствие:** $A^+ \cup B^-$ – максимальное независимое множество, как дополнение покрытия.

17.3. Всяческие ускорения алгоритма Куна

- **Обычный Кун**

Асимптотика $\mathcal{O}(VE)$

```
for (v: vertex) {
  u <- 0
  if (dfs(v))
    size++
}
```

- **Кун⁺**

Асимптотика $\mathcal{O}(V + |M| \cdot E)$

```
for (v : vertex)
  if (dfs(v))
    size++, u <- 0
```

Он обнуляет пометки вершин только, если мы нашли дополняющий чередующийся путь.

- **Быстрое обнуление массива за $\mathcal{O}(1)$:** нам лень бегать и обнулять весь массив, поэтому заведём число которое будет отвечать за "обнулённость", увеличивать его каждый раз когда должны были обнулить массив, сравнивать с ним и присваивать его.

- **Жадная инициализация**

```
for (e : edges)
  if (marked[e[0]] == 0 && marked[e[1]] == 0)
    add_to_matching(e)
```

Таким образом мы сразу же нашли паросочетание размера хотя бы $\frac{1}{2}|M_{max}|$
Далее запускаем алгоритм Кун⁺, который теперь работает в 2 раза быстрее.

- **И ещё одна оптимизация Куна**

Асимптотика тоже $\mathcal{O}(|M|E)$, но работает ещё быстрее, так как за 1-ю итерацию нашли сразу хотя бы $\frac{1}{2}|M_{max}|$ рёбер паросочетания. На каждой новой итерации находим хотя бы +1 ребро, возможно, больше.

```
while (run) {
  u <- 0
  run = 0
  forn(v, n)
    if (!paired[v] && dfs(v))
      run = 1
}
```

Лекция по алгоритмам #18
Обобщение Куна на произвольные графы

26 марта

18.1. Матрица Татта

Полиномиальный алгоритм($\mathcal{O}(V^3)$), выдающий с некоторой погрешностью результат – существует ли совершенное паросочетание в произвольном графе.

В ячейках соответствующим рёбрам стоят рандомные значения(для x_{ij}, где $0 < i < j \leq n$).

Матрица антисимметрична(т.е. в оставшихся x_{ij}, где $0 < j < i \leq n$), стоят значения противоположные x_{ji}.

Для отсутствующих рёбер поставим 0.

Теорема 18.1.1. Татта: В графе G $\exists$ совершенное паросочетание $\Leftrightarrow$ $[detA \neq 0]$

Теорему оставляем без доказательства и хотим сказать, что вероятность ошибки $\leqslant \frac{1}{p}$

18.2. Алгоритм поиска совершенного паросочетания:

```
1  bool may(G) {
2    for(int  i = 2; i <= n; i++) {
3      if(edge[1][i])
4        ans.add(1, i);
5      if(may(G - 1 - i))
6        return 1;
7      else
8        ans.pop();
9    }
10   return 0;
11 }
```

Работает за $\mathcal{O}((n-1)!!)$

18.3. Upgrade Куна почти всегда работающего на произвольном графе:

Рандомно выбираем вершины из которых ищем путь, перемешиваем рёбра перед тем как запускать dfs, если Кун пытается создать нечётный цикл – не идём по ребру.

$\exists$ тесты на которых вероятность успеха порядка $2^{-\frac{n}{2}}$, но их не очень много.

Можно запускать такое решение со счётчиком времени "Пока не TL".

18.4. Задача: Класифицировать рёбра на обязательно/возможно лежащие в max паросочетании для двудольного графа:

1)Для начала найдём max паросочетание $\mathcal{O}(VE)$ (умеем) $> \mathcal{O}(E\sqrt{V})$ (какой-то существующий) – M

2)Теперь рассмотрим какое-нибудь другое максимальное паросочетание – P

Теперь рассмотрим мн-во рёбер лежащих ровно в одном из них. Заметим, что такой набор рёбер образует граф, в котором у каждой вершины стпень $\leqslant 2$, кроме того в нём есть только

чётные циклы (из-за двудольности) и только пути чётной длины(т.к. в случае пути нечётной длины, конвертнув этот кусочек мы увеличим на 1 размер паросочетания второго графа).

Значит каждое ребро которое может лежать или не лежать, должно содержаться в цикле чётной длины с чередующимися (взяли/не взяли в паросочетание M) рёбрами. Аналогично, если ребро лежит на таком цикле, то мы можем конвертнуть цикл и получить опять же максимальное паросочетание.

Поиск циклов и путей чётной длины в двудольном графе:

- **Циклы.** $\mathcal{O}(E)$
  Если 2 ребра лежат в одной компоненте сильной связности, то существует чётный цикл содержащий их обоих. Соответственно ищем одним dfs-ом подходящие и помечаем рёбра.
- **Пути.** $\mathcal{O}(E)$
  Запускаемся от всех вершин у которых нет рёбер найденого паросочетания, всё до чего они дошли по чередующимся рёбрам является нужными путями, поэтому сразу помечаем вершины.

Лекция по алгоритмам #19
Marriage Problem, покраска рёбер графа

25 марта

19.1. Marriage Problem (stable matching)

Хоть это и не всем нравится, я буду придерживаться терминологии "девушек/юношей". Имеется n юношей и m девушек. У каждого юноши и у каждой девушки есть список нравящихся им представителей противоположного пола, причём список упорядочен по приоритету. Необходимо их стабильно поженить.

- $x.List$ – список нравящихся x
- $x.Pair$ – выбранный в пару x
- $x.charm(y)$ – привлекательность y с точки зрения x (позиция в списке)

Паросочетание M называется стабильным, если не существует ребра $(boy, girl) \notin M$: $boy.charm(girl) > boy.charm(boy.Pair) \wedge girl.charm(boy) > girl.charm(girl.Pair)$, то есть, нет эдаких Гурова и Анны Сергеевны, которым хочется быть друг с другом больше, чем со своими супругами (при этом мы считаем, что если юноша ни на ком не женат, то он женат на девушке с нулевой привлекательностью)

Задача – собственно найти стабильное паросочетание.

19.2. Технические моменты

Нам потребуется быстро понимать, кто из двух юношей больше интересует девушку. Для этого можно завести хеш-таблицу, которая по паре (x, y) возвращает $x.charm(y)$. Ещё можно просто отсортировать подсчётом все тройки (наши рёбра) $(boy, girl, c)$: $boy \in girl.List, c = girl.charm(boy)$, а затем пробежаться по всем спискам юношей и для каждой девушки x в списке юноши y запомнить $x.charm(y)$.

19.3. Алгоритм

Так называемый "алгоритм отложенного согласия".

1. Сначала юноши предлагают себя самым привлекательным с их точки зрения девушкам. Девушки среди всех юношей выбирают наиболее предпочтительных и говорят им "я подумаю", остальным говорят "нет".

2. Далее все отвергнутые юноши выкидывают из своих списков соответствующих девушек и пробуют удачу со следующими девушками (в их списках). При этом если некоторой девушке поступило более заманчивое предложение, она говорит "нет" своему текущему кандидату (и он переходит в мн-во отвергнутых) и "я подумаю" – новому.

3. Процесс продолжается пока не закончатся отвергнутые юноши с непустыми списками.

Это образно и на пальцах. В нашей реализации мы будем женить юношей по одному.

```
1  GetPair(boy):
2    while !boy.List.empty():
3      girl = boy.List.top(), boy.List.delete(girl) //достаём топовую девушку
4      if girl.Pair == -1: //если мы первые, кто делает ей предложение, то выходим
5        girl.Pair = boy, break;
6      else:
7        if girl.charm(girl.Pair) < girl.charm(boy): //если мы лучше текущего кандидата
8          swap(girl.Pair, boy) //продолжаем искать пару уже для него
9  FOR boy = 1..n
10   GetPair(boy)
```

19.4. Доказательство корректности

Докажем, что получившееся паросочетание (все пары $(girl, girl.Pair)$) стабильно. Пусть

$$\exists (boy, girl) \notin M : boy.charm(girl) > boy.charm(boy.Pair) \wedge girl.charm(boy) > girl.charm(girl.Pair)$$

В некоторый момент boy делал предложение $girl$, так как в итоге ему досталась девушка с меньшим приоритетом, а юноши предлагают себя девушкам в порядке убывания приоритета.

Однако в итоге $girl$ достался юноша с меньшим приоритетом, чем boy. Это невозможно, потому что по алгоритму:

- Юноша отвергается только если у текущего кандидата приоритет не меньше.
- Для каждой девушки $girl$ приоритет $girl.Pair$ может только увеличиваться в процессе алгоритма.

19.5. Замечания

- В графе может быть несколько стабильных паросочетаний
- Несмотря на то, что выбирают у нас девушки, алгоритм получает паросочетание, наилучшее для юношей. Соответственно, если обменять доли местами, получится паросочетание, наилучшее для девушек.

19.6. Покраска рёбер графа

Задача правильной покраска рёбер графа состоит в том, чтобы каждое ребро покрасить в некоторый цвет так, чтобы для каждой вершины все инцидентные рёбра были разных цветов, то есть, чтобы не было одноцветных рёбер с общими концами. Нам, естественно, интересно минимизировать число цветов. Эквивалентная переформулировка: разбить рёбра графа на минимальное число паросочетаний (каждый цвет порождает паросочетание и наоборот любое паросочетание можно покрасить в один цет).

Пусть D – максимальная степень вершины графа.

Теорема 19.6.1. *Визинга.* Существует правильная раскраска рёбер в D или в $D+1$ цвет.

Без доказательства.

Понятно, что меньшим числом цветов, чем D, не обойтись. Теорема Визинга так же даёт конструктивный алгоритм, красящий любой граф в $D+1$ цвет. Вопрос же о том, можно ли покрасить граф в D цветов в общем случае является NP-трудной задачей.

19.7. Двудольные графы

Тем не менее, двудольные графы графы несложно красятся в D цветов:

1. Сделаем из графа D-регулярный. Для этого просто добавим несколько вершин, чтобы сделать доли равными и соединим рёбрами вершины степени меньше D. Так как из левой доли выходит столько же рёбер, сколько входит в правую долю, если в левой доле есть вершина степени меньше D, то и в правой доли есть вершина степени меньше D.

2. В D регулярном графе обязательно найдётся совершенное паросочетание.

3. Найдём совершенное паросочетание и удалим из графа, покрасив все удалённые рёбра в первый цвет. У нас остался $(D-1)$-регулярный граф.

4. Повторяем, на каждом шаге крася рёбра в новый цвет, пока не закончатся рёбра.

5. Очевидно, мы сделаем D итераций, то есть задействуем ровно D цветов. Happy End.

Лекция по алгоритмам #20
Раскараска рёбер двудольных графов

25 марта

20.1. Вспомним прошлую лекцию

На прошлом занятии мы научились красить рёбра двудольного графа путём дополнения его до D-регулярного и разбиения его на D совершенных паросочетаний. Алгоритм работал за $\mathcal{O}(D^2 \cdot |V|^2)$. Сегодня мы научимся разбивать D-регулярный граф на D совершенных паросочетаний за $\mathcal{O}(D|V|^2 \log D)$.

Пусть D - чётно. Тогда мы можем выделить эйлеров цикл в каждой компоненте. Рассмотрим одну компоненту. Очевидно, что эйлеров цикл имеет чётную длину т.к. граф был двудольным (нечётный цикл нельзя раскрасить в два цвета). Построим два новых графа: в первый попадут рёбра эйлерова цикла на чётных позициях, во второй - на нечётных. Заметим, что степень каждой вершины уменьшилась ровно в два раза. Почему? Рассмотрим два последовательных ребра цикла. Пусть их общая вершина v. Раньше они давали вклад в её степень 2, а теперь одно из них имеет чётный номер и даёт вклад 1 в её степень в первом графе, а второе - нечётный номер и вклад 1 во втором графе. Через каждую вершину цикл пройдёт $\frac{D}{2}$ раз. Тогда если изначально у нас был D-регулярный граф, то в итоге мы получим два $\frac{D}{2}$-регулярных графа.

Представим, что D-степень двойки. Тогда за $\mathcal{O}(\log D)$ итераций мы получили бы D 1-регулярных графов, а 1-регулярные графы - это как раз искомые совершенные паросочетания.

Но мир настолько несовершенен, что далеко не все числа являются степенями двойки. Поэтому прежде чем перейти от D-регулярных графов к $\frac{D}{2}$ регулярным иногда придётся делать одно отщепление совершенного паросочетания, как в прошлом алгоритме. Итак, пусть $T(k)$ - полное время работы алгоритма если граф уже разбит на k-регулярные. Тогда $T(k) = D|V|^2 + T(\frac{k}{2})$, где $D|V|^2$ - верхняя оценка на время отщепления совершенных паросочетаний у подграфов нечётной регулярности (временем на поиск эйлерова цикла можно пренебречь). Почему $D|V|^2$? Отщепление совершенного паросочетания происходит за $\mathcal{O}(VE_i)$, тогда в сумме это займёт $\sum VE_i = V \sum E_i = VE$, а $E = \mathcal{O}(DV)$. Тогда $T(k) = D|V|^2 \log k$. А значит время работы алгоритма $\mathcal{O}(D|V|^2 \log D)$ т.е. $\mathcal{O}(|V|^3 \log V)$ в худшем случае.

20.2. Паросочетание, покрывающее вершины степени D

<u>Lm</u> 20.2.1. В двудольном графе существует паросочетание, покрывающее все вершины максимальной степени.

Действительно, вспомним алгоритм раскраски рёбер с предыдущей лекции. Пусть изначльно максимальная степень была равна D. Проделаем одну итерацию. Оставим теперь только изначальные рёбра и вершины. Очевидно, степень всех вершин неувеличилась. После удаления некоторого паросочетания (пересечение удалённого совершенного паросочетания и множества изначальных рёбер графа) максимальная степень вершин в графе не больше $D-1 => $ удалённое паросочетание покрывало все вершины степени D. А значит, такое паросочетание существует. Лемма доказана.

20.3. Покраска рёбер за $\mathcal{O}(E^2)$

Алгоритм прост - найдём паросочетание из леммы, удалим его и перейдём к меньшему графу. Т.к. максимальная степень вершин в графе от этого строго уменьшится, то не позже чем через D итераций мы получим пустой граф.

Пусть A - вершины первой доли, имеющие максимальную степень, B - вершины второй доли, имеющие максимальную степень. Сначала последовательно запустим dfs-ы из Куна из каждой вершины A и покроим паросочетанием все интересующие нас вершины из первой доли.

Почему это сработает? Заметим, что все покрытые вершины первой доли лежат в A. Пока мы ещё не добавили ни одного ребра в паросочетание, это утверждение верно. Дальше по индукции. Пусть мы нашли чередующийся путь. Начало найденного пути лежит в A по алгоритму. Все остальные вершины из A в пути были покрыты паросочетанием и на предыдущей итерации => и теперь утверждение верно. Докажем теперь, что если a лежит в A и ещё не покрыто паросочетанием, то существует нечётный чередующийся путь из a. Рассмотрим симметрическую разность M (текущего паросочетания) и P (паросочетания из леммы). Вершина a имеет в ней степень 1 т.к. покрыта P, но не покрыта M. Значит она - начало некоторого чередующегося пути. Чем он может оканчиваться? Если вершиной из второй доли, то мы нашли чередующийся нечётный путь из a. Победа. Пусть он заканчивается в вершине первой доли (а значит мы пришли в неё по ребру из M). Но тогда она лежит в A (т.к. покрыта M) => она должна быть покрыта и P => она не может быть концом пути т.к. её степень в симметрической разности чётна. Значит, такая ситуация невозможна. Уф.

Итак, мы покрыли паросочетанием все интересующие нас вершины первой доли. Теперь займёмся вершиными второй доли. Модифицируем dfs из Куна: теперь он заканчивается (и инвертирует рёбра на найденном пути) не только если нашёл нечётный чередующийся путь (т.е пришёл в свободную вершину), но и если пришёл в вершину той же доли, из которой стартовал, но которая не лежит в B. Докажем, что какой-то из этих путей (чётный, с концом в неинтересной вершине, или нечётный) всегда найдётся при запуске из непокрытой вершины b из B.

Рассмотрим симметрическую разность M (текущего паросочетания) и P (паросочетания из леммы). Вершина b имеет в ней степень 1 т.к. покрыта P, но не покрыта M. Значит она - начало некоторого чередующегося пути. Чем он может оканчиваться? Если вершиной из первой доли, то мы нашли чередующийся нечётный путь из b. Победа. Пусть он заканчивается в вершине второй доли. Если последняя вершина не интересная (не лежит в B), то опять же победа. Иначе она должна быть покрыта P, а т.к. мы пришли в неё по ребру из M, то её степень 2 => она не может быть концом пути. Уф.

Тогда вызвав модифицированный dfs из Куна из каждой непокрытой вершины B, мы посторим паросочетание покрывающее A и B => искомое.

Оценим время работы. Пусть на i-ой итерации мы удалили паросочетание размера M_i. Тогда все итерации займут не больше $\sum EM_i = E \cdot \sum M_i = E \cdot E = E^2$.

Лекция по алгоритмам #21
Вершинные покраски

25 марта

21.1. Теорема Брукса

Теорема 21.1.1. Вершины связного графа, в котором все вершины имеют не больше D соседей, можно раскрасить всего в D цветов, за исключением двух случаев — полных графов и циклов нечётной длины, для которых требуется D + 1 цвет.

Мы не будем доказывать эту теорему. Заметим только, что оценка сверху в D + 1 цвет очевидна. Действительно, давайте красить вершины в произвольном порядке. При покраске очередной вершины рассмотрим всех её покрашеных соседей и найдём цвет, которым ещё никто не покрашен. Т.к. соседей не больше D, а цветов D + 1, то свободный цвет всегда найдётся.

21.2. Алгоритм покраски вершин

На практике хорошо применим простой рекурсивный алгоритм: удалим вершину v минимальной степени, решим задачу для меньшего графа, вернём v в граф и покрасим в mex ($mex(X)$ - минимальное число из N_0, не встречающееся в X) цветов соседей. Для улучшения работы среди всех вершин минимальной степени рекомендуется выбирать случайную. Время работы зависит от структуры данных, которая используется для поиска вершины минимальной степени. Если использовать обыкновенную кучу, то время работы будет $\mathcal{O}(E \log V)$. Существует вариант реализации, работающий за $\mathcal{O}(E)$

21.3. Интересный факт про планарые графы

Lm 21.3.1. В любом планарном графе существует вершина степени не больше 5.

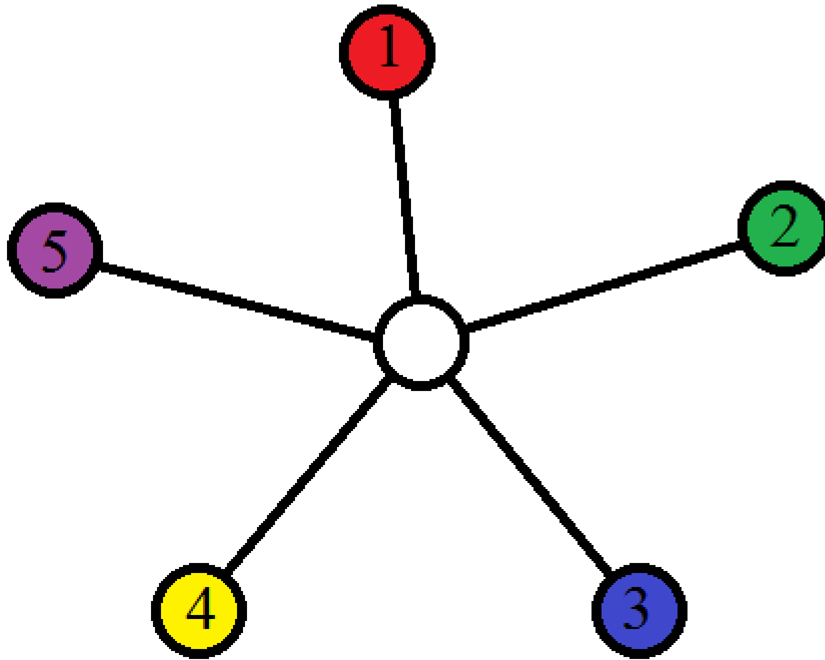
Примем лемму без доказательства.

21.4. Покраска планарного графа в 6 цветов

Благодаря последней лемме, описанный выше рекурсивный алгоритм покраски будет использовать не больше 6 цветов т.к. при покраске v у неё будет не больше 5 соседей.

21.5. Покраска планарного графа в 5 цветов

Модифицируем алгоритм из пункта 2, чтобы красить планарный граф в 5 цветов. Если степень вершины v при покраске оказалась меньше 5 или цвета каких-то соседей совпали, то их mex и так будет меньше 5 (цвета нумеруются с 0). Рассмотрим плохой случай.



Разноцветные соседи - это грустно. Давайте попробуем перекрасить кого-нибудь. Пусть этим "кто-нибудь- будет вершина 1 с рисунка. И перекрасим мы ей в цвет вершины 3. Но у вершины 1 могли быть соседи цвета вершины. Жаль. Придётся их перекрасить в цвет вершины 1. Таким образом, нам придётся перекрасить всю компоненту связности из вершин красного и синего цвета (по рисунку), в которой лежит первая вершина. Если при этом не перекрасилась вершина 3, то победа - можем покрасить v в красный. Иначе существет красно-синий путь из вершины 1 в вершину 3. Придётся смириться с этим. Но не будем терять надежду - попробуем перекрасить вершину 2 в цвет вершины 4. Но вдруг у нас не получится? Тогда будет существовать жёлто-зелёный путь из вершины 2 в вершину 4. Но это не возможно, т.к. граф планарный, а вершина 2 окружена красно-синим путём. Таким образом, не больше чем 2 dfs-ами мы можем решить проблему и покрасить v в один из 5 цветов.

Очевидно получается оценка на время работы в $\mathcal{O}(VE)$, но на практике алгоритм работает намного быстрее.

Лекция по алгоритмам #22
Венгерский алгоритм

25 марта

22.1. Задача и способы ее решения

Известна под названием “Задача о назначениях”. Дана матрица весов $w[i,j]$ – стоимость выполнения j-ой работы i-ым работником. Необходимо каждому рабочему назначить работу и минимизировать суммарную стоимость выполнения.

Способы решения:

1. MinCostFlow за $\mathcal{O}(V^3)$ или $\mathcal{O}(VE\log C)$.
2. Венгерский алгоритм за $\mathcal{O}(V^3)$. Преимуществом этого решения является маленькая константа.
3. Линейное программирование. Работает за $\mathcal{O}(V^3)$, хотя сам алгоритм экспоненциальный. Можно обобщить для произвольного графа.
   Заключается же он в назначении значений $0 \le x[i,j] \le 1$ при выполнении условий $\forall i \sum\limits_{j=1}^{n} x[i,j] = 1$ и $\forall j \sum\limits_{i=1}^{n} x[i,j] = 1$, минимизирую при этом $\sum\limits_{i=1}^{n}\sum\limits_{j=1}^{n} x[i,j]w[i,j]$. С этим помогает справится simplex method. Можно доказать, что значения $x[i,j]$, найденные этим алгоритмом, будут целыми.

22.2. Идея венегерского алгоритма

Заметим, что прибавив константу к столбцу либо строке, мы изменим ответ на эту же константу. Тогда сделаем все значения в матрице неотрицательными, т.е. отнимем от каждой строки ее минимум и получим $w[i,j] \ge 0$. Теперь, поддерживая это свойство и всегда добаваляя к паросочетанию ребра с весом 0, мы построим оптимальный ответ.

Осознав это, построим алгоритм:

1. Будем каждую вершину добавлять к паросочетанию, как-то изменяя текущее паросочетание. Поддерживаем четыре множества: A^+ – вершины первой доли, в которых мы уже были в dfs; A^- – в которых еще не были; аналогично определим для вершин второй доли B^+ и B^-.
2. Сразу заметим, что ребра текущего паросочетания выглядят как вершина из A^+ и B^+, либо из A^- и B^-. Тогда понятно, что хорошо бы рассмотреть в качестве кандидата на добавление нулевое ребро из A^+ и B^-, ведь, оказавшись бы та вершина незанятой, мы увеличили бы паросочетание, не нарушив при это вышеизложенных правил.
3. Теперь нам нужен такой 0. Для этого проделаем следующий “фокус”: найдем минимальный элемент в подматрице $A^+ \times B^-$, обозначим его m; вычтем m из всех строк A^+ и добавим m ко всем столбцам B^-. Вычитаем, чтобы получить 0, а добавляем для того, чтобы не получить элементы меньшие нуля в подматрице $A^+ \times B^+$.
4. В итоге получили 0. Если вершина из B^- свободна(а она могла быть занята текущим паросочетанием), то мы нашли увеличение. Иначе повторим шаг 3, изменив при этом поддерживаемые множества(одна вершина покинула $A-$ и появилась в $A+$, то же с вершиной из $B-$).

22.3. Псевдокод алгоритма

```
for (int i = 0; i < n; i++) {
  A_plus = v, B_minus = all_from_B;
  while (true) { \\ takes O(V) time
    m = min in A_plus x B_minus; \\ O(V^2)
    change matrix;  \\ O(V^2)
    find 0 in A_plus x B_minus; \\ O(V^2)
    if vertex free {
        update_route;
        break;
    }
  }
}
```

Видно, что алгоритм работает за $\mathcal{O}(V^4)$.

22.4. Оптимизация до $\mathcal{O}(V^3)$

Хочется сделать внутреннюю часть за $\mathcal{O}(V)$. Введем потенциалы для весов ребер: истинный вес $=$ `w[i,j] + row[i] + column[j]`. Уже умеем делать “фокус” за линейное время. Осталось лишь совершать две операции за линия. Здесь нам поможет аналогия с алгоритмом Прима. В нем поддерживается минимальный вес ребра, которое ведет в вершину – мы сделаем то же самое. Тогда отвечать на эти запросы мы сможем проходя весь массив минимумов за линию.

Лекция по алгоритмам #23
Потоки

5 апреля

23.1. Основные определения

Def 23.1.1. *Сеть. Пусть дан ориентированный граф $G = (V, E)$ без кратных ребер и петель с выделенными в нем вершинами $s, t \in V$. Вершины s и t называются истоком и стоком соответственно. Пусть также введена функция $c : V \times V \to \mathbb{R}_{\geq 0}$, которая называется пропускной способностью ребра, причем если $(u, v) \notin E$, то $c(u, v) = 0$. В таком случае четверка (G, s, t, c) называется* ***сетью****.*

Def 23.1.2. *Инцидентные ребра. Пусть дан ориентированный граф G. Для произвольной вершины $v \in V$ обозначим за $In\ v = \{e \in E : \exists u \in V, e = (u, v)\}$ – множество ребер, входящих в вершину v. Аналогично определим $Out\ v = \{e \in E : \exists u \in V, e = (v, u)\}$ – множество ребер, исходящих из вершины v.*

Def 23.1.3. *Поток в сети (по Асанову). Пусть дана сеть (G, s, t, c). Потоком в этой сети называется функция $f : E \to \mathbb{R}_{\geq 0}$, которая удовлетворяет следующим условиям:*

1. *$\forall e \in E : f(e) \leq c(e)$. То есть по ребру не может течь больше, чем это ребро способно пропустить.*
2. *$\forall v \in V \setminus \{s, t\} : \sum\limits_{e \in In\ v} f(e) = \sum\limits_{e \in Out\ v} f(e)$. То есть сколько в вершину, отличную от истока и стока, затекло, столько из нее и вытекло.*

Lm 23.1.4. $\sum\limits_{e \in Out\ s} f(e) - \sum\limits_{e \in In\ s} f(e) = \sum\limits_{e \in In\ t} f(e) - \sum\limits_{e \in Out\ t} f(e)$. То есть сколько потока вытекло из стока и не затекло обратно, столько и затекло в сток и не вытекло обратно.

Доказательство.

$$\sum_{v \in V} \sum_{e \in In\ v} f(e) = \sum_{e \in E} f(e),$$

$$\sum_{v \in V} \sum_{e \in Out\ v} f(e) = \sum_{e \in E} f(e) \Rightarrow$$

$$\Rightarrow \sum_{v \in V} \sum_{e \in In\ v} f(e) = \sum_{v \in V} \sum_{e \in Out\ v} f(e) \Leftrightarrow$$

$$\Leftrightarrow \sum_{v \in V} \left[\sum_{e \in In\ v} f(e) - \sum_{e \in Out\ v} f(e) \right] = 0 \Leftrightarrow$$

$$\Leftrightarrow \left[\sum_{e \in In\ s} f(e) - \sum_{e \in Out\ s} f(e) \right] + \left[\sum_{e \in In\ t} f(e) - \sum_{e \in Out\ t} f(e) \right] +$$

$$+ \sum_{v \in V \setminus \{s,t\}} \left[\sum_{e \in In\ v} f(e) - \sum_{e \in Out\ v} f(e) \right] = 0 \Leftrightarrow \text{(По (2) свойству потока)}$$

$$\Leftrightarrow \sum_{e\in In\ s} f(e) - \sum_{e\in Out\ s} f(e) + \sum_{e\in In\ t} f(e) - \sum_{e\in Out\ t} f(e) = 0 \Leftrightarrow$$
$$\Leftrightarrow \sum_{e\in Out\ s} f(e) - \sum_{e\in In\ s} f(e) = \sum_{e\in In\ t} f(e) - \sum_{e\in Out\ t} f(e)$$

■

Следствие 23.1.5. Если $deg_{in}\ s = deg_{out}\ t = 0$, то $\sum\limits_{e\in Out\ s} f(e) = \sum\limits_{e\in In\ t} f(e)$. То есть сколько потока из истока вытекло, столько и затекло в сток.

Доказательство. В таком случае просто $\sum\limits_{e\in In\ s} f(e) = \sum\limits_{e\in Out\ t} f(e) = 0$, так как таких ребер просто нет. ■

Def 23.1.6. *Величина потока. Величиной потока f в сети (G, s, t, c) называется $\sum\limits_{e\in Out\ s} f(e) - \sum\limits_{e\in In\ s} f(e)$, то есть количество потока, которое вытекает из истока и не затекает обратно. Обозначается $|f|$. Согласно лемме (23.1.4) $|f| = \sum\limits_{e\in In\ t} f(e) - \sum\limits_{e\in Out\ t} f(e)$.*

Замечание 23.1.7. Таким образом можно представлять поток как некое вещество, вытекающее из истока в некотором количестве, как-то хаотично протекающее через промежуточные узлы, не задерживаясь в них, и в конечном итоге стекающее в сток. Но на самом деле в потоке еще могут присутствовать и циркуляции – части этого вещества, которые циркулируют по некоторым циклам.

Def 23.1.8. *Циркуляция. Циркуляцией называется такой поток f, что $|f| = 0$, но $\exists e \in E : f(e) > 0$.*

Замечание 23.1.9. На практике с определением потока по Асанову оказывается трудно работать, поэтому вводится другое определение потока.

Def 23.1.10. *Поток в сети. Пусть дана сеть (G, s, t, c), в которой для каждого ребра **существует обратное**. Потоком в этой сети называется функция $f : V \times V \to \mathbb{R}$, которая удовлетворяет следующим условиям:*

1. $\forall u, v \in V : f(u, v) = -f(v, u)$. *Антисимметричность.*
2. $\forall u, v \in V : f(u, v) \le c(u, v)$. *То есть по ребру не может течь больше, чем это ребро способно пропустить.*
3. $\forall u \in V \setminus \{s, t\} : \sum\limits_{v\in V} f(u, v) = 0$. *Это свойство сходно со вторым свойством потоков по Асанову.*

Замечание 23.1.11. Если для некоторых вершин $u, v \in V : f(u, v) > 0$, то поток данной величины течет от вершины u к вершине v. Благодаря антисимметричности $f(v, u) < 0$ и это следует воспринимать так же, как отрицательную прибыль, то есть убыток. По ребру (v, u) как бы притекает отрицательное количество потока $f(v, u) = -f(u, v)$, иными словами утекает $-f(v, u) = f(u, v)$.
Если для некоторых вершин $u, v \in V : f(u, v) = 0$, то $f(v, u) = -f(u, v) = 0$.

Таким образом между двумя вершинами $u, v \in V$ поток либо течет в одну сторону, либо не течет вообще. Причем если $f(u,v) > 0$, то по второму свойству потока $c(u,v) \geq f(u,v) > 0 \Rightarrow (u,v) \in E$. То есть если поток течет от u к v, то он течет по существующему ребру.
Данные рассуждения подводят к следующей теореме.

Теорема 23.1.12. Пусть дана сеть (G,s,t,c) и поток f (по новому определению) в ней. Тогда $f' = \max\{0, f\big|_E\}$ является потоком по Асанову.

Доказательство. Образы f' неотрицательны по определению.
Первое свойство:

$$\forall e \in E : f'(e) = \max\{0, f(e)\} \leq c(e)$$

Второе свойство:

$$\forall u \in V \setminus \{s,t\} : \sum_{e \in Out\ u} f'(e) - \sum_{e \in In\ u} f'(e) =$$
$$= \sum_{e \in Out\ u} \max\{0, f(e)\} + \sum_{e \in In\ u} -\max\{0, f(e)\} =$$
$$= \sum_{e \in Out\ u} \max\{0, f(e)\} + \sum_{e \in In\ u} \min\{0, -f(e)\} = (\text{для оставшихся вершин значение } 0)$$
$$= \sum_{v \in V} \max\{0, f(u,v)\} + \sum_{v \in V} \min\{0, -f(v,u)\} =$$
$$= \sum_{v \in V} \max\{0, f(u,v)\} + \sum_{v \in V} \min\{0, f(u,v)\} =$$
$$= \sum_{v \in V} \Big[\max\{0, f(u,v)\} + \min\{0, f(u,v)\}\Big] =$$
$$= \sum_{v \in V} f(u,v) = 0 (\text{по третьему свойству потока})$$

■

Замечание 23.1.13. Как и в потоке по Асанову вводится величина потока $|f| = \sum\limits_{u \in V} f(s,u)$, которая совпадает с $\sum\limits_{u \in V} f(u,t)$, а также $|f| = |\max\{0, f\big|_E\}|$. Доказательство этих фактов просто комбинирует лемму (23.1.4) и теорему (23.1.12).

Замечание 23.1.14. В дальнейшем будут рассматриваться исключительно потоки по новому определению, так как с ними удобней работать.

Def 23.1.15. *Остаточная сеть. Пусть дана сеть (G,s,t,c) с потоком f в ней. Остаточной сетью называется новая сеть $G_f = (G,s,t,c_f)$, где $c_f = c - f$ и называется остаточной пропускной способностью.*
По второму свойству потока $\forall u, v \in V : f(u,v) \leq c(u,v) \Rightarrow c(u,v) - f(u,v) \geq 0$. Таким образом в остаточной сети пропускные способности неотрицательные.

Lm 23.1.16. Пусть дана сеть (G,s,t,c) с потоком f в ней, а также поток f' в G_f. Тогда $F = f + f'$ поток в исходной сети, причем $|F| = |f| + |f'|$.

Доказательство. Первое свойство:

$$\forall u, v \in V : F(u,v) = f(u,v) + f'(u,v) = -f(v,u) - f'(v,u) = -(f(v,u) + f'(v,u)) = -F(v,u)$$

Второе свойство:

$$\forall u, v \in V : F(u,v) = f(u,v) + f'(u,v) \le f(u,v) + c(u,v) - f(u,v) = c(u,v)$$

Третье свойство:

$$\forall u \in V \setminus \{s,t\} : \sum_{v \in V} F(u,v) = \sum_{v \in V} \Big[f(u,v) + f'(u,v) \Big] = \sum_{v \in V} f(u,v) + \sum_{v \in V} f'(u,v) = 0$$

Величина потока:

$$|F| = \sum_{u \in V} F(s,u) = \sum_{u \in V} \Big[f(s,u) + f'(s,u) \Big] = \sum_{u \in V} f(s,u) + \sum_{u \in V} f'(s,u) = |f| + |f'|$$

■

Замечание 23.1.17. На практике остаточная сеть хранится в виде пары чисел: пропускной способности ребра и количества потока, протекающего по этому ребру.

Def 23.1.18. *Разрез в сети. Пусть дана сеть (G, s, t, c). $\langle S, T \rangle$ разрезом в сети называется два множества $S, T \subset V$, такие что:*

1. $S \cap T = \varnothing$
2. $S \cup T = V$
3. $s \in S, t \in T$

То есть разрез – это разбиение вершин графа на две части таким образом, что исток и сток попадают в разные части.

Def 23.1.19. *Величина разреза. Величиной разреза называется $|\langle S, T \rangle| = \sum\limits_{u \in S, v \in T} c(u,v)$. То есть суммарная пропускная способность ребер, проходящих из S в T.*

Замечание 23.1.20. Величина разреза считается для исходной сети, а не для остаточной.

23.2. Задача о максимальном потоке, теорема Форда-Фалкерсона

Ясно, что величина потока ограничена сверху суммарной пропускной способностью ребер, исходящих из истока. Встает закономерный вопрос: существует ли максимальный поток среди всевозможных, и если да, то как его отыскать?

Def 23.2.1. *Дополняющий путь. Пусть дана сеть (G, s, t, c) с потоком f в ней. Дополняющим путем называется такой путь p из s в t, что $\forall e \in p : c_f(e) > 0$.*

Lm 23.2.2. Пусть дана сеть (G, s, t, c) с потоком f в ней. Если существует дополняющий путь, то f не максимален.

Доказательство. Пусть p простой дополняющий путь (если есть какой-то, то его можно сжать до простого). Рассмотрим поток f' в G_f, который пропускает по каждому ребру из p $\min\limits_{e \in p}\{c_f(e)\} > 0$ единиц потока. Тогда по лемме (23.1.16) $f + f'$ – поток в исходной сети, причем $|f + f'| = |f| + |f'|$, но $|f'| = \sum\limits_{u \in V} f'(s, u) = f'(s, p_1) = f'(p_0, p_1) = \min\limits_{e \in p}\{c_f(e)\} > 0$, где p_i – i-ая вершина в пути p. Таким образом $|f| + |f'| > |f|$, значит поток f не максимальный. ■

Lm 23.2.3. Пусть дана сеть (G, s, t, c) и разрез $\langle S, T\rangle$ в ней. Тогда $|f| = \sum\limits_{u \in S, v \in T} f(u, v)$. То есть поток, протекающий через разрез, равен по величине потоку в сети.

Доказательство.

$$\begin{aligned}
|f| = \sum_{u \in V} f(s, u) = (\text{по (3) свойству потока})\\
= \sum_{v \in V} f(s, v) + \sum_{u \in S\setminus\{s\}} \sum_{v \in V} f(u, v) =\\
= \sum_{u \in S} \sum_{v \in V} f(u, v) =\\
= \sum_{u \in S} \left[\sum_{v \in S} f(u, v) + \sum_{v \in T} f(u, v)\right] =\\
= \sum_{u \in S} \sum_{v \in S} f(u, v) + \sum_{u \in S} \sum_{v \in T} f(u, v) =\\
(\text{каждое ребро в } S \text{ посчиталось вместе с обратным, но по антисимметричности это } 0)\\
= \sum_{u \in S} \sum_{v \in T} f(u, v) =\\
= \sum_{u \in S, v \in T} f(u, v)
\end{aligned}$$

■

Lm 23.2.4. Пусть дана сеть (G, s, t, c). $\forall$ разреза $\langle S, T\rangle$ и потока $f : |f| \le |\langle S, T\rangle|$.

Доказательство.

$$\begin{aligned}
|f| = (\text{По лемме (23.2.3)})\\
= \sum_{u \in S, v \in T} f(u, v) \le (\text{по (2) свойству потока})\\
\le \sum_{u \in S, v \in T} c(u, v) =\\
= |\langle S, T\rangle|
\end{aligned}$$

■

Lm 23.2.5. Пусть дана сеть (G, s, t, c) с потоком f в ней. Если дополняющих путей не существует, то $\exists\langle S, T\rangle : |f| = |\langle S, T\rangle|$.

Доказательство. Построим этот разрез конструктивно. Возьмем в качестве множества S множество всех вершин, которые достижимы из s по ребрам, для которых $c_f > 0$. Тогда $T = V \setminus S$, и $t \in T$, так как иначе это бы означало, что существует дополняющий путь. $\forall u \in S, v \in T$: $f(u,v) = c(u,v)$, так как иначе было бы $f(u,v) < c(u,v) \Rightarrow c_f(u,v) = c(u,v) - f(u,v) > 0$ и v должна была бы попасть в S. Но по лемме (23.2.3): $|f| = \sum\limits_{u \in S, v \in T} f(u,v) = \sum\limits_{u \in S, v \in T} c(u,v) = |\langle S,T \rangle|$. ■

Следствие 23.2.6. Разрез $\langle S,T \rangle$ из леммы является минимальным разрезом.

Доказательство. От противного через лемму (23.2.4). ■

Теорема 23.2.7. (Форд-Фалкерсон). Пусть дана сеть (G,s,t,c) с потоком f в ней. Тогда следующие три утверждения эквивалентны:

1. Поток $|f|$ максимален.

2. Дополняющие пути отсутствуют.

3. $\exists \langle S,T \rangle : |f| = |\langle S,T \rangle|$.

Доказательство. $(1) \Rightarrow (2)$:
От противного по лемме (23.2.2).
$(2) \Rightarrow (3)$:
Лемма (23.2.5).
$(3) \Rightarrow (1)$:
От противного по лемме (23.2.4). ■

Замечание 23.2.8. Теорема Форда-Фалкерсона не отвечает на вопрос о существовании максимального потока в произвольной сети.

23.3. Алгоритм Форда-Фалкерсона

Def 23.3.1. *Целочисленная сеть. Сеть (G,s,t,c) называется целочисленной, если $Im\ c \subset \mathbb{Z}$.*

Def 23.3.2. *Целочисленный поток. Поток f называется целочисленным, если $Im\ f \subset \mathbb{Z}$.*

Теорема 23.3.3. (Алгоритм Форда-Фалкерсона). В целочисленной сети (G,s,t,c) существует максимальный поток, и причем целочисленный.

Доказательство. Конструктивное доказательство. Пусть в произвольный момент времени набран целочисленный поток f_k размера k. Если не существует дополняющих путей, то по теореме (23.2.7) поток f_k максимальный. Иначе остаточная сеть G_{f_k} целочисленная и в ней существует дополняющий путь, по которому можно пропустить единицу потока. Сделав это перейдем к потоку f_{k+1} размера $k+1$. Изначально поток нулевой, а увеличивать поток таким образом получится не более $\sum\limits_{u \in V} c(s,u)$ раз, что означает, что в какой-то момент не останется дополняющих путей и поток будет максимальным. ■

Замечание 23.3.4. Если искать путь с помощью обхода в глубину, то алгоритм работает за $\mathcal{O}(E \cdot |f|)$.

Замечание 23.3.5. С практической точки зрения алгоритм можно немного ускорить, если пропускать по пути не 1 потока, а минимум среди c_f на пути.

23.4. Существование максимального потока в произвольной сети. Алгоритм Эдмондса-Карпа

Алгоритм 23.4.1. (Эдмондс-Карп). Алгоритм ищет максимальный поток в произвольной сети (G, s, t, c). В произвольный момент времени набран некоторый поток f. Если не существует дополняющих путей, то по теореме (23.2.7) поток f максимальный. Иначе найдем дополняющий путь p минимальной длины (в ребрах) и пропустим по нему $\min\limits_{e \in p}\{c_f(e)\}$ единиц потока.

Все последующие рассуждения этого раздела неразрывно связаны с алгоритмом Эдмонса-Карпа и нацелены на доказательство его конечности.

Def 23.4.2. *Расстояние до вершины. Обозначим за $d_i(v)$ расстояние в ребрах от s до вершины $v \in V$ в остаточной сети G_{f_i} (то есть по ребрам для которых $c_{f_i} = 0$ ходить нельзя), где f_i поток, набранный алгоритмом Эдмондса-Карпа после добавления i дополняющих путей.*

Lm 23.4.3. Пусть $v \in V$, тогда $d_i(v) \nearrow$.

Доказательство. Зафиксируем момент времени i и докажем, что $\forall v \in V : d_{i+1}(v) \geq d_i(v)$.
Пусть $\exists v \in V : d_{i+1}(v) < d_i(v)$ (теоретически $d_i(v)$ может равняться $+\infty$, что означает, что вершина v была недостижима из s), среди таких возьмем v с минимальным d_{i+1}. $v \neq s$, так как $\forall i : d_i(s) = 0$. Тогда обозначим за u вершину на кратчайшем пути от s до v в $G_{f_{i+1}}$, предшествующую v. В таком случае $d_{i+1}(v) = d_{i+1}(u)+1 \Rightarrow d_{i+1}(u) < d_{i+1}(v) \Rightarrow$ (по предположению)$d_{i+1}(u) \geq d_i(u)$ (*). Возможно две ситуации:

1. $c_{f_i} > 0$, то есть ребро (u, v) присутствовало до добавления дополняющего пути. Тогда $d_{i+1}(v) = d_{i+1}(u) + 1 \geq$ (согласно (*))$d_i(u) + 1 \geq$ (неравенство треугольника, так как ребро было)$d_i(v)$, что противоречит предположению.

2. $c_{f_i} = 0$, то есть ребро (u, v) появилось после добавления дополняющего пути. Но тогда $c(u, v) - f_i(u, v) = 0, c(u, v) - f_{i+1}(u, v) > 0 \Rightarrow f_i(u, v) - f_{i+1}(u, v) > 0 \Rightarrow f_{i+1}(v, u) - f_i(v, u) > 0$, но поток f_{i+1} получился из f_i добавлением дополняющего пути, а значит разность и есть дополняющий путь. Тогда неравенство означает, что ребро (v, u) принадлежит дополняющему пути (что сразу исключает возможность того, что $d_i(v) = +\infty$), а так как он кратчайший, то $d_i(u) = d_i(v) + 1 \Rightarrow d_i(v) = d_i(u) - 1 \leq$ (согласно (*))$d_{i+1}(u) - 1 = d_{i+1}(v) - 2 < d_{i+1}(v)$, что противоречит предположению.

■

Def 23.4.4. *Насыщение ребра. Ребро (u, v) насыщается после добавления $(i+1)$-ого дополняющего пути, если $c_{f_i} > 0$, а $c_{f_{i+1}} = 0$.*

Lm 23.4.5. В течении работы алгоритма $\forall$ ребро (u, v) насытится не более $\frac{V}{2} + 1$ раз.

Доказательство. Рассмотрим некоторый момент a насыщения произвольного ребра (u, v) : $c_{f_a}(u, v) > 0, c_{f_{a+1}}(u, v) = 0$ и другой момент c насыщения этого же ребра $c_{f_c}(u, v) > 0, c_{f_{c+1}}(u, v) = 0$, так как $c_{f_{a+1}}(u, v) = 0$, а $c_{f_c}(u, v) > 0$, то $\exists b \in (a, c) : c_{f_b}(u, v) = 0, c_{f_{b+1}}(u, v) > 0$. По рассуждениям, аналогичными со вторым случаем в лемме (23.4.3), ребро (v, u) принадлежит $(b + 1)$-ому дополняющему пути, а значит $d_{b+1}(u) = d_{b+1}(v) + 1 \Rightarrow d_{a+1}(v) \leq d_{b+1}(v) = d_{b+1}(u) - 1 \leq d_c(u) - 1 = d_c(v) - 2$, оба неравенства верны по лемме (23.4.3). Таким образом $d_{a+1}(v) + 2 \leq d_c(v)$, то есть от одного насыщения до следующего расстояние до конца ребра увеличится хотя бы на 2. Так как это расстояние не может превышать V, то суммарное количество насыщений не более $\frac{V}{2} + 1$. ■

Теорема 23.4.6. Алгоритм Эдмондса-Карпа работает за конечное время.

Доказательство. Так как за добавление дополняющего пути хотя бы одно ребро насыщается (то, на котором достигается минимум среди остаточных пропускных способностей), то по лемме (23.4.5) через $E(\frac{V}{2} + 1)$ таких добавлений ни одно ребро нельзя будет насытить, что означает отсутствие дополняющих путей. ■

Замечание 23.4.7. Если искать кратчайший дополняющий путь с помощью обхода в ширину, то алгоритм будет работать за $\mathcal{O}(VE^2)$

Замечание 23.4.8. В целочисленной сети алгоритм найдет целочисленный поток.

Теорема 23.4.9. В произвольной сети (G, s, t, c) существует максимальный поток.

Доказательство. Алгоритм Эдмондса-Карпа является конструктивным доказательством этого факта. ■

23.5. Декомпозиция потока

Теорема 23.5.1. Любой поток f в сети (G, s, t, c) можно разбить на сумму потоков $\sum\limits_{i=1}^{n} f_i$, где f_i либо простой путь из s в t, либо цикл.

Доказательство. Конструктивное доказательство. Запустим обход из s по ребрам, вдоль которых течет поток. Если обход дошел до t, то найден простой путь из s в t, по которому можно отменить $\min\limits_{e \in p} \{f(e)\}$ потока. Если обход нашел цикл, то можно отменить поток по циклу. Когда обход заходит в очередную вершину, то если это не сток, то обход сможет пойти в следующую вершину благодаря (3) свойству потока. Так как каждый раз хотя бы через одно ребро перестает течь поток, то алгоритм завершится, когда из s не будет ребер, по которым течет поток. К этому моменту в сети могла остаться циркуляция, которую можно разбить аналогичным образом, запуская обход из вершин, отличных от s. ■

Замечание 23.5.2. Алгоритм работает за $\mathcal{O}(E^2)$.

Замечание 23.5.3. $|f| = \sum\limits_{i=1}^{n} |f_i|$.

Замечание 23.5.4. Декомпозиция целочисленного потока целочисленна.

Замечание 23.5.5. Время работы алгоритма можно улучшить до $\mathcal{O}(VE)$, если для каждой вершины поддерживать указатель на первое исходящее ребро по которому можно пройти. Тогда все обходы отработают за $\mathcal{O}(kV + E)$, где k количество обходов.

Лекция по алгоритмам #24
Сложные потоки

8 апреля

24.1. Вспомнить все

24.1.1. Разность потоков одинакового размера

Lm 24.1.1. $\forall f_1, f_2 : |f_1| = |f_2|$
$f_2 - f_1$ — циркуляция G_{f_1}
Доказательство: $f_2 - f_1 \leqslant c - f_1$

24.1.2. Эдмондс-Карп

1. BFS — кратчайший путь из истока в сток
2. Пускаем поток, равный минимальной остаточной пропускной способности на этом пути Time $= VE \cdot BFS = VE^2$

Lm 24.1.2. $d[s \to t] \nearrow$ в G_f
Посмотреть, где угодно, или вспомнить прошлую лекцию.

Lm 24.1.3. Всего будет не более $\frac{VE}{2}$ запусков.
Доказательство: $\forall$ — ребро насытится $\frac{V}{2}$ раз (так как d[e] строго увеличивается, а расстояние не превысит V; при этом, чтобы насытить, надо "насытить" в обратную сторону)

24.2. Масштабирование потока

$E \log U \cdot$ "$pathfinding$" $= E^2 \log U$, где U — максимальная пропускная способность

```
for (int k = 2 ^ t; k > 0; k >>= 1) { // U >= 2 ^ t > U / 2
    while (Exists such path) {
        Find path with c_e - f_e >= k;
        Push flow of size k;
    }
}
```

Lm 24.2.1. На каждом шаге не более $2E$ путей
Доказательство: рассмотрим переход от $2k$ к k. Мы уже не можем протолкнуть поток размера $2k$. Таким образом, задается разрез. Значит, его пропускная способность $\leqslant 2k \cdot E$. Значит, в нем мы протолкнем $\leqslant 2E$ раз поток размера k

24.3. Алгоритм Диница

$VE \cdot V$ (работаем, как Эдмондс-Карп, но ищем путь быстро):
Рассмотрим граф после работы BFS. Он делится на слои по расстоянию от истока. Построим новый граф. В нем останутся только те ребра, которые ведут из слоя в следующий. Теперь модифицированным dfs-ом ищем в новом графе пути длины d, где d — кратчайшее расстояние в остаточной сети. dfs при откате назад удаляет ребро, по которому возвращается.

```
for (;;) {
    d = bfs(s, t);
    while (Exists path of len d) {
        Find path;
        Push flow;
    }
}
```

Оценим время: пусть K_d — число путей длины d. Тогда для каждого d мы сделаем $E + d \cdot K_d$ операций. Сложим все. $\sum_d (E + d \cdot K_d) \leqslant VE + V \cdot \sum_d K_d \leqslant VE + V \cdot \frac{VE}{2} = \mathcal{O}(V^2 E)$

24.3.1. +Scaling

```
    for (k = 2 ^ t; k; k >>= 1) {
        Dinic(k);
    }
```

Теорема 24.3.1. $VE \log U$
Доказательство: посмотрим оценку $VE + V \cdot \sum_d K_d$. Где $\sum_d K_d \leqslant 2E$ из доказательства масштабирования

24.4. Хопкрофт-Карп

Ищем паросочетание Диницом

Теорема 24.4.1. $\mathcal{O}(E\sqrt{V})$
Доказательство:

1. Фаза $= \mathcal{O}(E)$
2. После первых $\sqrt{V}$ фаз $d(s,t) > \sqrt{V}$
3. Рассмотрим симметрическую разность полученного сочетания и максимального. Все разбилось на пути. Так как они не пересекаются, а длина их $> \sqrt{V}$, то всего их $\leqslant \sqrt{V}$. Тогда осталось достроить не больше корня. То есть всего $\leqslant 2\sqrt{V}$ фаз

Замечания:

1. Надо писать, как Куна (сразу обратно по взятому ребру прыгать).
2. Из структуры графа пересечение по вершине влечет за собой пересечение по ребру, поэтому достаточно помечать вершины и не ходить в них.

Лекция по алгоритмам #25
Первая Теорема Карзанова.

8 апреля

Def 25.0.2. *Единичная сеть*
Сеть, в которой все пропускные способности равны единице.

Теорема 25.0.3. *На единичных сетях алгоритм Диница работает за* $\mathcal{O}(VE)$

Доказательство. $Time = \sum\limits_{d}(E + d \cdot K_d) \le VE + V|f| \le 2VE$ ∎

Def 25.0.4. *В последующих теоремах:* $C := \sum\limits_{v \ne s,t} \min\{c_{in}[v], c_{out}[v]\}$

Теорема 25.0.5. *Первая теорема Карзанова.*
Для целочисленных пропускных способностей количество фаз $\le 2\sqrt{C}$.

Доказательство. Сделаем первые $\sqrt{C}$ фаз, получим некоторый поток f_1.
Рассмотрим $(f_{max} - f_1)$ – поток в G_{f_1}, где f_{max} – какой-то максимальный поток.
Рассмотрим декомпозицию на пути с единичным потоком и циркуляцию.
Каждый такой путь длины больше $\sqrt{C}$, и для каждой промежуточной вершины он "уменьшает" потенциал $\min\{c_{in}[v], c_{out}[v]\}$ на 1 (через вершину v не может пройти больше путей).
Таким образом каждый путь "уменьшает" величину $C = \sum\limits_{v \ne s,t} \min\{c_{in}[v], c_{out}[v]\}$ больше, чем на $\sqrt{C}$, поэтому путей меньше $\sqrt{C} \Rightarrow |f_{max} - f_1| < \sqrt{C} \Rightarrow$ алгоритм Диница в дальнейшем найдет меньше $\sqrt{C}$ путей $\Rightarrow$ будет меньше $\sqrt{C}$ фаз. ∎

- **Следствие #1:** На единичных сетях алгоитм Диница работает $\mathcal{O}(E\sqrt{C})$.

Доказательство. Каждый dfs в пределах фазы "забирает" некоторое количество ребер, а по обратным ребрам алгоритм в пределах той же фазы не пойдет $\Rightarrow$ время одной фазы $= \mathcal{O}(E)$ ∎

- **Следствие #2:** На единичных сетях алгоитм Диница работает $\mathcal{O}(E\sqrt{E})$.

Доказательство. $C \le 2E \Rightarrow Time = O(E\sqrt{E})$ ∎

- **Следствие #3:** Хопкрофт-Карп работает за $\mathcal{O}(E\sqrt{V})$

Доказательство. $C \le 2V \Rightarrow Time = O(E\sqrt{V})$ ∎

Лекция по алгоритмам #26
Потоки: LR.

12 апреля

26.1. Что хотим?

Def 26.1.1. *LR-поток* — $\forall e \in Edges\colon L_e \le flow_e \le R_e$

Варианты задач:

1. Найти какой-нибудь корректный поток.
2. Найти максимальный.
3. Естественным образом обобщается на mincost, об этом в других сериях.

26.2. Понемногу двигаемся к цели

1. Научимся искать поток в графе с несколькими истоками и стоками. Для этого добавим фиктивные исток и сток. Из фиктивного истока проведем ребра бесконечной пропускной способности во все реальные, из всех реальных стоков — бексонечные ребра в фиктивный сток. Найдем поток между фиктивными.
2. Решим такую задачу: в городе есть заводы и магазины. Они соединены дорогами, у каждой есть пропускная способность. Про заводы известно, сколько товара они производят, про магазины — сколько продают. Требуется составить такой план перевозок, что весь товар будет продан и ни в одном магазине не будет нехватки товара.
   Нарисуем граф, вершинам сопоставим значения — избытки (положительные) и недостатки (отрицательные) товара. Создадим фиктивный исток, из него проведем во все вершины с избытком ребра величины, равной избытку. Создадим фиктивный сток, в него проведем ребра из всех вершин с недостатком величины, равной модулю недостатка. Найдем максимальный поток в этом графе, он и будет ответом (в случае, если он насытил все ребра из истока и все ребра в сток).
3. Теперь пусть появятся величины L_e и R_e. Пусть $e.from = a, e.to = b$. Тогда добавим в вершине a избыток L_e, а в b — недостаток L_e. e зададим пропускную способность $(R_e - L_e)$. Что мы сейчас сделали? Свели задачу о LR-циркуляции (т.к. у нас нет истока и стока, это циркуляция, а не поток) к задаче из предыдущего пункта.
4. А теперь сведем задачу о LR-потоке к LR-циркуляции. Просто добавим бесконечное ребро из стока в исток.
5. Какой-нибудь нашли, но что делать дальше?

26.3. Максимизируем

Ищем LR-поток. Пусть f — найденный нами поток, f^{max} — какой-нибудь максимальный LR-поток. Рассмотрим $(f^{max} - f)$, это обычный поток в G_f (напоминаю, что так мы обозначаем остаточную сеть потока f).

Взглянем на какое-нибудь ребро, скажем, e. Про него верны следующие неравенства:

$$L_e \le f_e^{max} \le R_e$$

$$L_e \le f_e \le R_e$$

$$D_e = L_e - f_e \le f_e^{max} - f_e \le R_e - f_e = U_e$$

Остаточная сеть. В ней концы e соединены ребрами пропускной способности U_e в прямую и $-D_e$ в обратную сторону. Если $f \ne f^{max}$, то выходит, что в остаточной сети есть дополняющий путь (по пути все $U_e > 0$, а $D_e \le 0$ всегда). Значит, можно просто искать дополняющие пути для максимизации, что делали и с обычным. Получается, что LR-поток ищем за то же время, что и обычный — $\mathcal{O}(flow)$.

26.4. Не забываем старые знания.

В алгоритме Диница много времени занимает пускание потока по ней. Почему много? Потому что по одним и тем же ребрам, у которых от запуска к запуска пропускная способность остается положительной, мы можем ходить много-много раз. На помощь приходит link-cut tree! Будем поддерживать сеть сперва как лес (конечно же link-cut деревьев). Какие операции нужно будет реализовывать?

1. Искать минимум на пути и его позицию, чтобы находить ребра, которые насытятся на данном запуске.
2. Добавлять на пути (собственно запуск потока).
3. Удалять ребро, если насытилось.

Когда решаем найти очередной путь, из вершины ходим не по отдельным ребрам, а сразу переходим в корень ее компоненты в link-cut-е. Получается быстро, алгоритм начинает работать за $\mathcal{O}(VE \log V)$.

Немножко про это по ссылкам:
http://math.mit.edu/~rpeng/18434/blockingFlows.pdf
http://www.arl.wustl.edu/~jst/cse/542/text/sec19.pdf

Лекция по алгоритмам #27
Минимальный глобальный разрез и Диниц

12 апреля

27.1. Диниц с Link-cut tree

1. Как и в простом Динице, в начале каждой фазы строим слоистую сеть.
2. Как превратить нашу сеть в дерево? Запустим из стока DFS, который ходит только по ребрам, противоположные которым лежат в нашей слоистой сети. Ребра, противоположные тем, по которым он пройдет составляют дерево с корнем в истоке.
3. Создадим на этом дереве структуру Link-cut tree. Она должна будет хранить на путях минимум и его позицию(среди ребер), а также уметь вычитать из всех ребер пути одинаковую величину. Зачем: весом ребра будет разность пропускной способности ребра и потока по нему.
4. Возьмем минимум на пути из истока в сток (путь один, ибо дерево же!) и вычтем из всех ребер на этом пути. Хотя бы одно ребро обнулилось (причем точно обнулился тот минимум, позицию которого мы знаем из Link-cut tree). Сделаем cut по обнулившемуся ребру, а потом заменим его на другое (следующее в списке смежности соответствующей вершины, которое лежит в слоистой сети), сделав link его концов.

Таким образом, у нас есть $\mathcal{O}(V)$ фаз, на каждой из которых мы делаем DFS, а потом обнуляем не более E ребер, за $\mathcal{O}(\log V)$ каждое (ибо Link-cut со Splay-деревьями внутри). Таким образом, суммарная сложность – $\mathcal{O}(V \cdot E \cdot \log V)$.

Суть задачи – удалить в неориентированном взвешенном графе множество ребер минимально возможного суммарного веса так, чтобы нарушить связность графа.
Примечание: в данной задаче будет использоваться термин "слить/соединить вершины". Это значит – создать новую вершину, которая будет иметь все ребра обеих сливаемых (кратные сохраняются!) и убить две слитые.

27.2. Каргер-Штейн

Примечание: в этом алгоритме мы считаем, что веса всех ребер равны 1 (иначе вероятность испортится). Однако, можно ребро веса k представить в виде k кратных ребер. Сложность испортится, но работать будет.
Примечание: Данный алгоритм – вероятностный, а потому в оценке сложности будет присутствовать переменная C. Что это? Пусть у вас есть алгоритм, который имеет вероятность ошибки $\frac{1}{e}$. Пусть вы хотите, чтобы вероятность ошибки была $\frac{1}{C}$. Тогда вам всего лишь достаточно запустить ваш алгоритм $\log C$ раз и выбрать лучший из ответов.

27.2.1. $\mathcal{O}(n^4 \cdot \log C)$

Основная идея данного алгоритма: в минимальном разрезе немного ребер. А точнее, если минимальная степень вершины равна k, то в минимальном разрезе не более k ребер (ибо просто отрезать вершниу с минимальной степенью – тоже вариант). А всего в графе много ребер. А точнее, так как k – минимальная степень, $E >= \frac{n \cdot k}{2}$.

Значит, если выбрать случайное ребро, то с вероятностью хотя бы $\frac{k\cdot 2}{n\cdot k} = \frac{2}{n}$ есть минимальный разрез, в котором этого ребра нет. Значит, можно слить концы этого ребра, ибо в ответе они будут по одну сторону разреза.
Делаем так, пока в графе больше двух вершин. В графе из двух вершин разрез единственный, он же минимальный. Вероятность успешного нахождения ответа – $\prod\limits_{i=3}^{n}(1-\frac{2}{i}) = \prod\limits_{i=3}^{n}(\frac{i-2}{i}) = \frac{2}{(n-1)\cdot n}$.
Не очень-то здорово. Но то, что мы делали до сих пор работало всего лишь за $\mathcal{O}(n^2)$. Поэтому, можно просто повторить алгоритм несколько раз и взять лучший из ответов. Если точнее, то повторим алгоритм n^2 раз. Тогда вероятность того, что все отработали неверно – $(\frac{n^2-2}{n^2})^{n^2}$, что стремится к $\frac{1}{e^2}$ при n, стремящемся к бесконечности(причем, достаточно быстро). Применив то, о чем написано в примечании выше, получим нужные асимптотику и вероятность ошибки.
Примечание: можно сливать вершины не за линию, а быстрее, если использовать DSU.

27.2.2. $\mathcal{O}(n^2 \cdot \log^2 n \cdot \log C)$

Вернемся к моменту, когда мы оценивали вероятность нахождения правильного ответа одним проходом алгоритма. $\prod\limits_{i=3}^{n}(\frac{i-2}{i})$ – интересное произведение. Если точнее, то при достаточно больших i множители не сильно портят нашу вероятность, а при маленьких – очень сильно. Тогда давайте разобъем произведение на две части: $\prod\limits_{i=\frac{n}{\sqrt{2}}}^{n}(\frac{i-2}{i})$ и все остальное. Первая часть имеет вероятность ошибки $\frac{1}{2}$, а вторая... Со второй все похуже. Но можно запустить ее дважды (естественно, таким образом, чтобы разные ветки выбирали разные случайные ребра!). Вероятность корректного срабатывания (обозначу P(n)) теперь – $\frac{1}{2}\cdot(1-(1-P(\frac{n}{\sqrt{2}}))^2)$. Неприятная штука. Но посчитать ее все же можно, если перейти в дифференциальные уравнения(не очень понятно, почему это так, но это так):
$f'(n) = -\frac{(f(n))^2}{2}$
$-\frac{f'(n)}{n^2} = \frac{1}{2}$
Проинтегрировали обе части
$\frac{1}{f(n)} = \frac{k}{2} + Const$
$f(n) = \frac{1}{\frac{k}{2}+Const}$
Подставим k = $\log_{\sqrt{2}} n$
$f(n) = \frac{1}{\log n}$, где f(n) = P(n). Повторили $\log n$ раз и получили вероятность ошибки $\frac{1}{e}$ и сложность $\mathcal{O}(n^2\cdot\log^2 n)$, а получать отсюда требуемые сложность и вероятность ошибки мы уже умеем.

27.3. Штор-Вагнер

Этот алгоритм состоит из множества фаз. До первой фазы считаем, что ответ – бесконечность. На каждой фазе мы:

1. Берем произвольную вершину(например, первую) и добавляем ее в множество(в начале фазы множество пустое).
2. Начинаем добавлять в множество остальные вершины, причем каждый раз добавляется самая "связанная"с этим множеством вершина, еще не лежащая в нем.
3. Добавив таким образом все вершины в наше множество, посмотрим на последние две. Утверждается(у нас без доказательства, но на emaxx-е оно есть), что минимальный глобальный разрез, который разделяет их – это тот, который попросту отрезает последнюю вершину от

всех остальных. Значит, либо такой разрез – это ответ, либо эти две вершины в ответе будут лежать по одну сторону разреза. А тогда их можно объединить в одну вершину.

4. Значит, в конце фазы надо сделать две вещи: обновить ответ (минимум из текущего ответа и разреза, отрезавшего последнюю вершину от остальных) и слить две последние вершины в одну.
5. Конец фазы. Повторять, пока не останется 2 вершины. Для них разрез очевиден. Обновим ответ в последний раз и выведем его.

Сложность алгоритма – $\mathcal{O}(n^3)$, если искать самую связанную вершину за линию, или $\mathcal{O}(n \cdot (m + n \cdot \log n))$, если не полениться и написать кучу Фибоначчи.

Лекция по алгоритмам #28
Mincost flow

3 мая 2016г.

Есть граф. У каждого ребра есть пропускная способность, величина потока, через него протекающего и вес (c_e, f_e, w_e). Числа вещественные.

Ищем поток. Весом потока f является $w(f)\frac{1}{2}\sum\limits_{e\in f} f_e w_e$. Суммирование по прямым и по обратным рёбрам. Полусумма из-за того, что если есть ребро (f_e, w_e), то обратным ребром (которое мы сами сделаем) будет $(-f_e, -w_e)$.

28.1. 4(четыре) задачи

1. Mincost k-flow: (|f| = k) AND (w(f) → min)
2. Mincost flow: (w(f) → min)
3. Mincost circulation: Mincost 0-flow
4. Mincost maxflow: Mincost k-flow, |f| → max

Видим, что задачи очень похожи. Например, 1-ая(первая) задача к 3-ей(третьей) сводится добавлением ребра из стока в исток с пропускной способностью k и бесконечно большим отрицательным весом.

28.2. Алгоритм Клейна для mincost circulation

Пока есть цикл отрицательного веса, пустим по нему $\min\limits_{e\in negcycle}(c_e - f_e)$.

Отметим, что при целых параметрах рёбер алгоритм конечный, просто потому что
$Time \le |w(f)| \cdot O(finding_negcycle)$

Lm 28.2.1. Среди всех потоков заданного размера поток f имеет минимальный вес $\iff$ в G_f нет negcycle.

Доказательство. ⇒ Если неверно, то пустим поток по negcycle, т.е. уменьшим w(f).

⇐ $G_f = f_1 - f =$ mincost - answer_we_have. Это циркуляция, т.е. набор циклов, причём неотрицательных. Поэтому $w(f_1) \ge w(f)$. Но f_1 - mincost. ■

28.3. Алгоритм через доп.пути

Пусть есть f_k - ответ для mincost k-flow.
$f_{k+1} = f_k + min_path(G_{f_k})$

Доказательство. Расмотрим разность f_{k+1} и f_k в G_f. Это путь и циркуляция, вес которой неотрицательный (лемма), значит, вес разности не меньше веса пути, который не меньше веса кратчайшего пути. ■

Если нет отрицательных циклов. то $f_0 = 0$, а если есть, то сделайте, чтоб не было.

Mincost flow можно искать так же, только остановиться надо, когда вес пути стал неотрицательным (можно бинпоиском такое поискать, если хочется).

Лекция по алгоритмам #29
MincostFlow

15 аперля

- **Задача**

Найти поток минимальной стоимости, любого размера.

29.1. Алгоритм первый, простейший.

Будем находить и добавлять к потоку путь минимального веса, пока этот вес отрицателен.

Доказательство. По сути мы перебираем k и выбираем лучший из Mincost k-flow. Поскольку веса кратчайших путей монотонно возрастают, $\exists k_0 \colon \forall k < k_0 \; minpath[k] < 0, \forall k > k_0 \; minpath[k] > 0$. Следовательно, Mincost k-flow до некоторого k_0 монотонно убывает, а потом монотонно возрастает. Алгоритм находит это k_0. ■

29.2. Алгоритм второй, бинпоиск по ответу.

Поскольку мы ищем минимальное значение аргумента, для которого функция положительна, можно делать это двоичным поиском.

Найдём бинпоиском $\min k \colon w(k) > w(k+1)$.

Время работы: $O(BS \times (MincostKFlow + FB))$, где BS – бинпоиск, а FB – алгоритм Форда-Белллмана.

29.3. Первый быстрый алгоритм.

Возьмём алгоритм Клейна и заменим поиск отрицательного цикла на поиск цикла минимального среднего веса. Будем повторять до тех пор, пока этот средний вес отрицателен.

Время работы: $O(nm\log(nC))$, где C – минимальный вес ребра. Без доказательства.

29.4. Второй быстрый алгоритм. Capacity Scaling.

Снова сведём задачу поиска минимального потока к задаче поиска минимальной циркуляции, добавив ребро t->s бесконечной пропускной способности и нулевого веса.

1. Создадим граф G, который будет отличаться от исходного тем, что все пропускные способности будут нулевыми.
2. Пройдём циклом по k от $\log C$ до 0, где C – максимальная пропускная способность ребра.
3. На каждом шаге пройдём по всем рёбрам e и для каждого проверим, стоит в k-м разряде двоичного представления $c(e)$ 0 или 1.
4. Если в k-м разряде 1, увеличим пропускную способность e в G на 2^k.
5. Возможно, увеличение пропускной способности ребра e в G привело к появлению отрицательного цикла. Для того, чтобы проверить, произошло это или нет, найдём алгоритмом Форда-Беллмана минимальный путь из b в a.
6. Если w(b->a) + w(e) < 0, то отрицательный цикл замкнулся. В таком случае пустим по нему поток размера 2^k.

Доказательство. Докажем оптимальность выбранной циркуляции для графа, полученного последним увеличением пропускной способности.

Разность оптимальных циркуляций в новом и старом графе состоит из циклов, проходящих по e, т.е. набор единичных путей b->a + много раз взятое e. Очевидно, что циклов (и путей) не более 2^k. 2^k путей найдётся, поскольку пропускные способности всех существующих в G рёбер кратны 2^k. ■

Время работы: (поиск путей) $*m * \log C$. Если искать пути алгоритмом Форда-Беллмана, получится $O * (nm^2 \log C)$. Можно искать пути алгоритмом Гольдберга за $O(m\sqrt{n} \log)$, и тогда время составит $O(m^2\sqrt{n} \log^2 C)$, но на практике выйгрыш во времени небольшой, а код заметно усложняется, и поэтому чаще используется Форд-Беллман.

Псевдокод:

```
for (k = 30; k >= 0; --k)
  for e: edges
    if (e.capacity & 2^k)
      add_in_g(e, 2^k)
      fb(b)
      if (b->a && w(b->a) + w(e) < 0)
        inc_flow(2^k, b->a + e)
```

29.5. Ускорение с помощью Дейкстры с потенциалами.

Базовый алгоритм и алгоритм с Capacity Scaling используют поиск кратчайшего пути между двумя вершинами. Эффективно решать задачу поиска кратчайшего пути в орграфе позволяет алгоритм Дейкстры. В случае поиска MincutFlow с алгоритмом Дейкстры возникает проблема: мы умеем применять алгоритм Дейкстры на графах с отрицательными рёбрами с помощью потенциалов, но мы не готовы к появлению в графе новых отрицательных рёбер, а появляться при добавлении потока они могут. Решим проблему прибавлением к потенциалу вершины расстояния до неё на каждой итерации.

$d[b] - d[a] \leq w(e)$ (иначе расстояние до b можно уменьшить через e) $\rightarrow d[a] + w(e) - d[b] \geq 0 \rightarrow$ веса с новыми потенциалами неотрицательны.

Псевдокод:

```
d0=FB
for i = 1..k
  (d1, path) = Djkstra(d0)
  d0 += d1
```

Лекция по алгоритмам #30
Строки. Поиск подстроки в строке.

19 апреля

30.1. Обозначения

s, t – строки, $|s|$ – длина строки,
`s[l:r)` или `s[l:r]` – подстрока,
`s[0:i)` – префикс, `s[i:|s|-1]` – суффикс.
Σ – алфавит, $|\Sigma|$ – размер алфавита.

30.2. C++.

В языке `C++` у строк типа `string` есть стандартный метод `find`. Работает за $\mathcal{O}(|s| \cdot |t|)$, возвращает целое число – номер позиции в исходной строке, начиная с которого начинается первое вхождение подстроки.

Функция из `<cstring>` `strstr(t, s)` ищет s в t и работает за линию под `Unix` и за квадрат под `Windows`.

В обоих случаях стоит заметить, квадрат имеет очень маленькую константу (AVX регистры).

30.3. КМП.

Постановка задачи. Есть строка t и образец s. Мы хотим приложить строку s ко всем позициям строки t и вернуть всё множество вхождений. Введем понятие префикс-функции строки.

Def 30.3.1. *Префикс-функция строки s – массив p. $p[i]$ – длина максимального собственного префикса `s[0:i]`, совпадающего с суффиксом `s[0:i]`. $p[0] = 0$.*

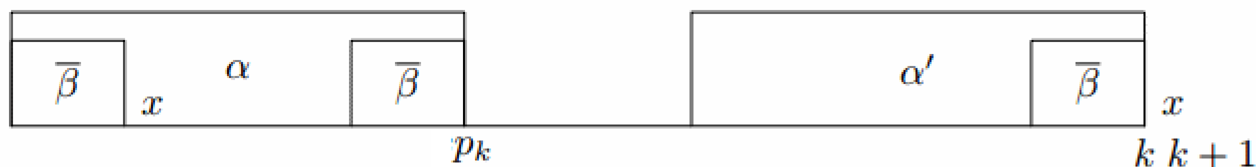
- **Алгоритм Кнута-Мориса-Пратта** .

`#` – любой символ, который не встречается ни в t, ни в s. Создадим новую строку: $s\#t$ и найдем от неё префикс-функцию. Ни одно значение префикс-функции на такой строке не может быть больше длины строки s, потому что у нас используется символ `#`. Так же, если префикс-функция равна длине s, то мы нашли искомое вхождение, так как префикс длины $|s|$ в $s\#t$ – это и есть строка s.

```
1 p[0] = k = 0
2 for (i = 0; i < n; i++)
3 while (k > 0 && s[k] != s[i])
4   k = p[k - 1]
5 if (s[k] == s[i])
6   k++
7 p[i] = k
```

Алгоритм работает за $\mathcal{O}(|s| + |text|)$ (от длины конкатенации). В приведенном коде видно, что внутри `while` переменная k убывает, а после этого цикла k увеличивается суммарно не более $n-1$ раз. При этом k всегда неотрицательно. Докажем корректность. Для этого введем несколько обозначений:

1. α – префикс длины p_k, а α' – суффикс.
2. x – $k+1$-й символ.
3. $\beta = \overline{\beta}x$ – префикс длины p_{k+1}, нахождение $\overline{\beta}$ эквивалентно вычислению p_{k+1}.



Теорема 30.3.2. $\forall k\colon p[k+1] \le p[k]+1$ имеем $p[k+1] = p[k]+1 \iff s[p_k+1] = s[k+1]$.

Доказательство. Если $p_{k+1} > p_k+1$, то $|\beta| > |\alpha|$, причём β является одновременно префиксом строки s и суффиксом $s[1:k]$, что противоречит выбору p_k.
Если $s[k+1] \ne s[p_k+1]$, то $p_{k+1} \le p_k$ и $\overline{\beta}$ – собственный префикс и суффикс α.

■

30.4. Z-функция.

Def 30.4.1. *Z-функция – массив. $Z[0] = 0$. $\forall i \ne 0\colon Z[i] = LCP(s, s[i:n])$, где LCP – наибольший общий префикс, n – длина строки s.*

Для поиска подстроки, снова введем строку $s\#t$ и посчитаем на ней значения Z-функции. Найдем все позиции i, для которых $Z[i] = |s|$, гарантируется, что это позиции всех вхождений строки s в строку t. Значение Z-функции не может быть больше, чем $|s|$, а значения, меньшие этого нас не устраивают.

```
1 Z[0] = 0
2 for (i = 0; i < n; i++)
3   int k = 0
4   while (s[i + k] == s[k])
5     k++
6   Z[i] = k
```

Приведенный алгоритм работает за $\Theta(n^2)$ для строки $aaa\ldots a$.

Lm 30.4.2. Пусть $l\colon l + Z[l] = \max$ среди всех $l < i$. $r = l + Z[l]$.
Тогда $s[0:Z[l]) = s[l:r)$ и $s[i-l:Z[l]) = s[i:r)$.

Следствие леммы: $Z[i] \ge \min(r-i, Z[i]-l)$.
Немного модифицируем код, чтобы получить асимптотику $\mathcal{O}(n)$.

```
1   Z[0] = 0, l = r = 0
2   for (i = 0; i < n; i++)
3     int k = r <= i ? 0 : min(r - i, Z[i] - 1)
4     while (s[i + k] == s[k])
5       k++
6     Z[i] = k
7     if (i + k > r) l = i, r = i + k
```

Lm 30.4.3. Приведенный выше алгоритм работает за $\mathcal{O}(n)$.

Доказательство. Если увеличивается k, то увеличивается и значение r, а оно не может увеличиваться больше n раз.
Пусть последний совпаший по лемме символ находится $< r$. У строки $i - l$ последний символ не совпал, значит, и рассматриваемый последний символ не совпадёт. Пусть $\geq r$, тогда $i + Z[i]$ будет правее r и k увеличится. ■

Лекция по алгоритмам #31
Алгоритм Бойера-Мура, Хеширование

19 апреля

31.1. Алгоритм Бойера-Мура

Алгоритм Бойера-Мура-Хорспула

Пусть дан алфавит Σ, текст t и шаблон s. Требуется найти хотя бы одно вхождение s в t или сказать, что их нет. АБМХ – сублинейный алгоритм, решающий эту задачу за время $\mathcal{O}(\frac{|t|}{|s|})$ в среднем. В худшем случае алгоритм работает за время $\mathcal{O}(|t| \cdot |s|)$.

Алгоритм:

Совместим начала текста и шаблона. Сравним последний символ шаблона с символом строки. Если они совпадают, сравним предпоследний. И так далее. Если все символы шаблона совпали с символами строки, вхождение найдено. Если в какой-то момент символы не совпали, тогда сдвинем шаблон вправо на несколько символов. Пусть i – позиция в тексте, которая сейчас совмещена с последним символом шаблона. Тогда сдвинем шаблон вправо на минимальное ненулевое количество символов так, чтобы буква шаблона, которая теперь соответствует позиции i, равнялась $t[i]$, либо, если $t[i]$ не встречается в s, на всю длину шаблона.

Пример:

t = "*abcabcabbababa*" s = "*baba*"

a	b	c	a	b	c	a	b	b	a	b	a	b	a
b	a	b	a										
		b	a	b	a								
						b	a	b	a				
								b	a	b	a		

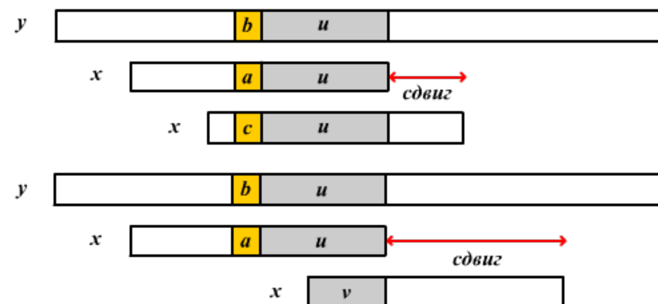
Пусть символы строки нумеруются с 0. Посчитаем для каждого символа c величину $last[c]$ – позиция последнего вхождения символа c в s или -1, если символ c не встречается в s или стоит только на последней позиции. $\mathcal{O}(|\Sigma| + |s|)$

Тогда если при очередном сравнении символы не совпали, а c – символ текста, на который сейчас накладывается последний символ шаблона, шаблон нужно сдвинуть на $|s| - last[c] - 1$.

Алгоритм Бойера-Мура

Описанное используемое для сдвига правило называется «правилом плохого символа». Если к нему добавить модификацию – ещё одно правило, «правило хорошего суффикса», получится алгоритм Бойера-Мура (вообще-то говоря, АБМХ – модифицация АБМ, а не наоборот).

Что же это за правило? Пусть на текущем этапе у s с t совпал некий суффикс u. Тогда можно сдвинуть s так, чтобы под этим u в t оказалось самое правое вхождение u из s, не считая самого суффикса. Если же u в s больше не встречается, можно попробовать провернуть то же самое с суффиксом поменьше. В результате этим самым правым вхождением может оказаться некоторый префикс v.



Сдвиги для суффиксов можно предподсчитать с помощью z/префикс функции.

Таким образом для текущей позиции высчитываются сдвиги по двум правилам и выбирается лучший сдвиг. Асимптотически АБМ работает столько же, сколько и АБМХ. На случайных тестах АБМХ работает быстрее. На реальных тестах с неравномерным распределением символов АБМ может дать как выигрыш, так и проигрыш.

31.2. Хеширование

Хэш-функция преобразует строку в число. Если две строки равны, то равны и их хеши. При этом далеко не всегда из равенства хешей следует равенство строк.

Полиномиальный хеш

Пусть есть строка $s = s_0 s_1 s_2 ... s_{n-1}$. Можно сделать из неё многочлен, используя символы в качестве коэффициентов:

$H_0 = 0,\ H_{i+1} = H_i \cdot p + s_i,\ H_n = s_0 \cdot p^{n-1} + s_1 \cdot p^{n-2} + ... + s_{n-1}$, где p – точка, в которой считается значение многочлена.

Такое преобразование можно использовать в качестве хеша. Причём если $p > |\Sigma|$, то отображение получается инъективным, то есть хеш каждой строки уникален. И всё бы хорошо, да вот только при таком подходе получаются слишком большие числа, с которыми абсолютно неудобно работать. Поэтому будем брать хеши по модулю:

$H_0 = 0,\ H_{i+1} = (H_i \cdot p + s_i)\ \%\ M$, где M – некоторый выбранный нами модуль.

При таком подходе в процессе работы могут возникнуть коллизии – ситуации, когда хеши двух различных строк равны. Далее сделаем несколько оценок на вероятность возникновения коллизий.

Теорема 31.2.1. Пусть $s_1,\ s_2, ...,\ s_k$ – различные строки. M – фиксированное натуральное число. P – случайное натуральное число. Тогда

$$Pr[Hash(s_1),\ Hash(s_2), ...,\ Hash(s_k) \text{ – различны}] \geq$$
$$\geq 1 - \sum_{i<j} Pr[Hash(s_i) = Hash(s_j)] = 1 - \frac{k(k-1)}{2} \cdot Pr[Hash(s_i) = Hash(s_j)]$$

Если строки фиксированы, то из следующих лемм будет следовать $Pr[Hash(s_i) = Hash(s_j)] \leq \frac{\max |s_i|}{M}$.

Если строки случайны, то из следующих лемм будет следовать $Pr[Hash(s_i) = Hash(s_j)] \leq \frac{1}{M}$.

Lm 31.2.2. $Hash(s_1) = Hash(s_2) \Leftrightarrow Hash(s_1 - s_2) = 0$

Доказательство. $Hash(s_1 - s_2) = \sum p^i \cdot (s_1 - s_2)[n - i - 1]$ ■

Таким образом, вероятность того, что у двух различных строк совпадут хеши, равна вероятности того, что хеш случайной строки будет равен нулю. С такой формулировкой куда проще оценивать.

Lm 31.2.3. Пусть p, M – фиксированные натуральные числа, $s = s_0 s_1 ... s_{n-1}$ – случайная строка, $s_i \in [0; M-1]$. Тогда

$$Pr[Hash(s) = 0] = \frac{1}{M}$$

Доказательство. Возьмём полином от строки – $S(x)$. Разделим его с остатком на $x - p$: $S(x) = (x-p)Q(x) + r$. Видно, что при $x = p$ получаем $S(p) = r$. $Q(x)$ – полином $(n-1)$-ой степени. При этом какие бы он не имел коэффициенты $q_0,\ q_1, ...,\ q_{n-2}$, всё равно $S(p) = r$. Всего наборов $(q_0, q_1, ..., q_{n-2}, r)$ – M^n. При этом $Hash(s) = 0$ на наборах, где $r = 0$, а таких M^{n-1}. ■

Таким образом, $Pr[Hash(s) = 0] = \frac{M^{n-1}}{M^n} = \frac{1}{M}$.

Lm 31.2.4. Пусть M и s – фиксированные модуль и строка. p – случайное число из промежутка $[0; M-1]$. M – простое. Тогда

$$Pr[Hash(s) = 0] \le \frac{|s|}{M}$$

Доказательство. Количество корней многочлена $S(x)$ не больше $|s|$. ■

Lm 31.2.5. Пусть M – фиксированный модуль, а s и p – случайные строка и число. Тогда

$$Pr[Hash(s) = 0] = \frac{1}{M}$$

Доказательство. `Lm 3.2.3` ⇒ `Lm 3.2.5`. В данной лемме у нас дано случайное p. Зафиксируем какое-нибудь одно p. Вероятность будет равна $\frac{1}{M}$. И так для любого $p \in [0; M-1]$. Тогда

$$Pr[Hash(s) = 0] = \frac{1}{M} \cdot \sum_{p=0}^{M-1} Pr[Hash(s) = 0, p \text{ и } M \text{ фикс}] = \frac{1}{M} \cdot M \cdot \frac{1}{M} = \frac{1}{M}$$

■

Вывод. Как приготовить качественные хеши?

M – фиксированное простое число. p – случайное число из промежутка $[|\Sigma| + 1; M - |\Sigma|]$.

Как мы уже доказали выше, $Pr[\text{возникновение коллизии}] \le \frac{n(n-1)}{2} \cdot \frac{1}{M}$, где n – количество различных строк. Поэтому, если взять достаточно большой модуль, даже при большом количестве строк вероятность возникновения коллизии будет очень мала. Например, если взять $M = 10^{18} + 3$, то при количестве строк до 10^7 вероятность возникновения коллизии будет не более 0.005%.

И ещё несколько фактов/трюков:

1. Не стесняйтесь использовать __int128, он существует и работает довольно быстро.

2. Модуль можно использовать как один, так и два, и три, и сколько угодно. В таком случае, если модули простые, вероятность коллизии будет обратно пропорциональна произведению модулей.

3. При равенстве хешей можно вручную проверить строки на равенство. Это не универсальный способ, но в ситуациях, где совпадение хешей маловероятно, это беспроигрышный вариант. Особенно если использовать `memcmp`, коротый работает ну очень быстро.

4. Можно использовать $M = 2^{64}$, это намного удобнее и работает быстрее. И в большинстве случаев этого хватит. Но не стоит использовать на серьёзных соревнованиях/проектах, поскольку очень легко генерируется строка Туэ-Морса, на которой такой модуль выдаст множество коллизий.

5. Для любой пары (M, p) существует алгоритм для построения тестов, на которых полиномиальный хеш с такими параметрами будет встречать коллизии. Из чего следует, что хеши не очень надёжная вещь и лучше использовать их только в случаях крайней необходимости.

Лекция по алгоритмам #32
Хеширование

22 апреля

32.1. Хеши, теория

<u>Lm</u> 32.1.1. Количество корней случайного многочлена степени n над полем вычетов по модулю p. Ответ: 1.

Доказательство. У нас была лемма: $\forall a$ с вероятностью $\frac{1}{p}$ a будет корнем многочлена. Всего чисел в поле: p. Поэтому имеем N многочленов, $\frac{N}{p}$ пар, где a – корень.

Т.е. фиксируем пару <a, p>. Считаем количество корней: $\frac{\frac{N}{p} \cdot p}{N} = 1$. Взяли количество пар, где a – корень, умножили на количество чисел в поле и разделили на общее количество многочленов. ■

На лекции было замечено, что мы пользуемся простотой модуля. Пример: пусть мы имеем многочлен x^{64} по модулю 2^{64}. Любое четное число является корнем этого многочлена. Таким образом, оно имеет аж 2^{63} корней, нам бы хотелось быть иметь 1 корень. Поэтому мы с большой долей вероятности тыкаемся в корень этого многочлена, т.е. какой-то символ в строке не особо вносит какой-то вклад. Т.е. меняем его и получаем другую строку, а хеш у нее такой же.

<u>Lm</u> 32.1.2. Зафиксируем пару <p, M>. Тогда рассмотрим вероятность :
$Pr = 1 - \frac{n(n-1)}{2} \cdot Pr[Hash(s_1) == Hash(s_2)]$.
Тогда $Pr \leqslant \frac{|s|}{M}$ при фиксированных строчках и при случайных строчках имеем: $Pr = \frac{1}{M}$
Т.е. это по сути вероятность совпадения хешей у двух строк одинаковой длины.

Доказательство. Рассмотрим разность строк s_1 и s_2 по модулю p : $(s_1 - s_2)(p) == 0$. p выбирается случайно. С какой вероятностью p является корнем многочлена? У нас многочлены выбираются случайно, значит по предыдущей лемме у них один корень в среднем. Тогда из предыдущих лемм у нас уже есть на это оценки. На лекции лемма, через которую следует первое утверждение обозначалась первой леммой. В конспекте у Никиты это Lm 31.2.4. Вторая лемма – лемма 3 в лекции и в конспекте у Никиты она тоже идет следующей. ■

Замечание: несмотря на утверждение леммы, на практике мы имеем $\frac{1}{M}$.

32.2. Что есть в C++

Напоминание в C++ есть unordered_set<string> – хеш таблица строк. Вычисляет хеш(ключ) и по нему уже запихивает в таблицу.

```
std::hash<string> h;
auto p = h(s);
```

Полиномиальный хеш, работает за длину строки. Т.к. работает в 32-ом режиме, есть коллизии.

32.3. Строка Туэ-Морса

Построим строку, на которой падает модуль 2^k. Т.е. какое бы случайное p мы бы не брали, все будет плохо. Хешом у нас будет: $\sum s_i \cdot p^i\ mod\ M$. Где $M = 2^k, s_i \in \{0, 1\}$

Два определения строки Туэ-Морса :

- s[i] = __builtin_popcount(i) mod 2. Т.е. количество единичных битов в двоичной записи числа i. __builtin_popcount(i) – встроенная функция языка C++.
- Строим рекурсивно. $s_n = s_{n-1} + not\ s_{n-1}$. not – инвертирование битов в строке s.

База: $s_0 = 0$.
Далее имеем следующее: $s_1 = 01, s_2 = 0110, s_3 = 01101001$ и тд.

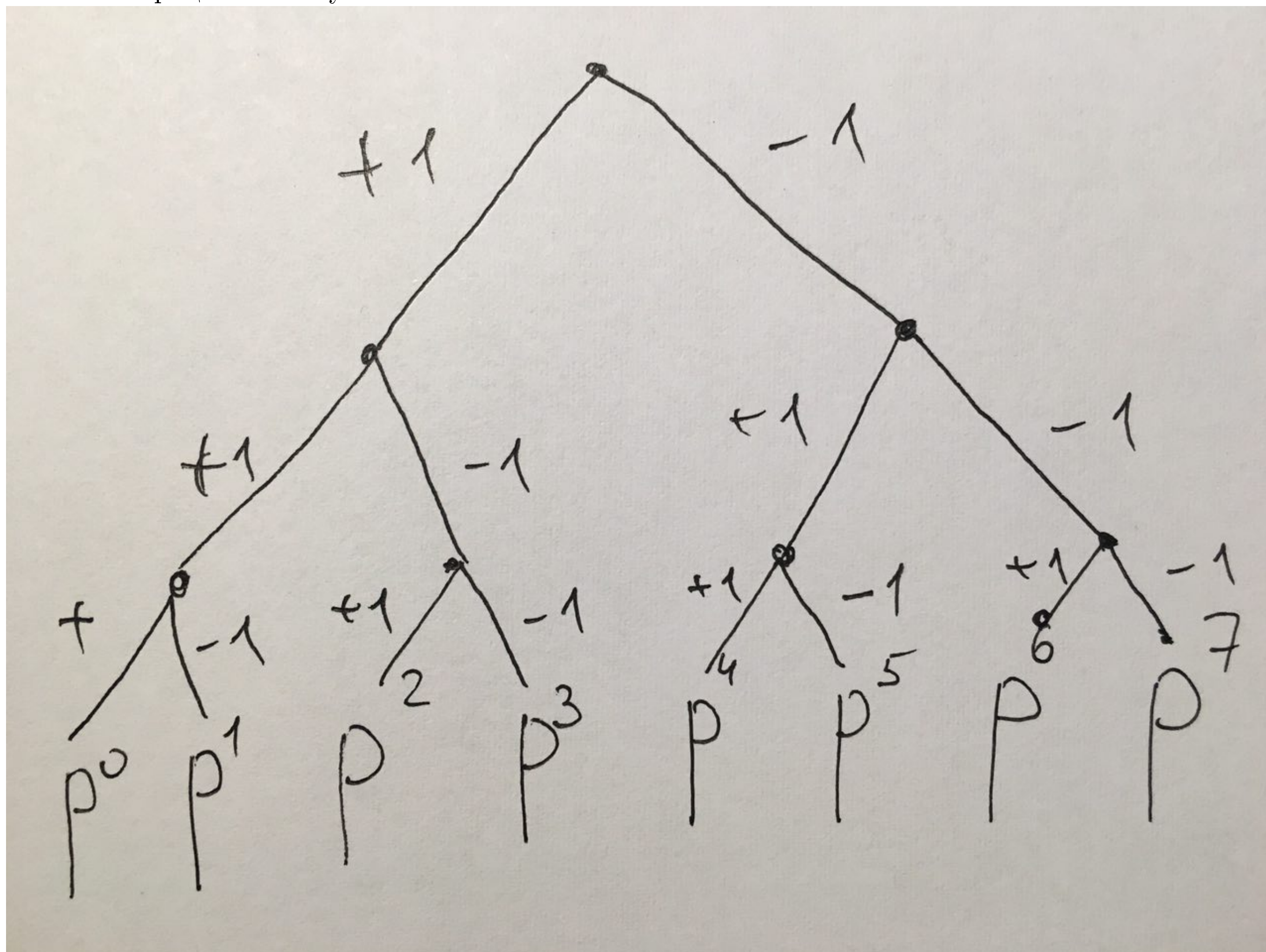
Хотим найти для какой длины строки случится коллизия. Рассмотрим
$Hash(s_i) == Hash(\overline{s}_i)$ для достаточно большого i. Когда так произойдет?
Смотрим на разность $Hash(s_1) - Hash(\overline{s}_i) = 0$
Это есть следующее: $\sum\limits_{j=0}^{2^i-1} p^j \cdot t_j$, где $t_j \in \{+1, -1\}$ – соответствующий коэффициент.
Это равно $-(1-p)(1-p^2)\dots(1-p^{2^i-1})$. Здесь i множителей. Понимать этот переход скорее всего нужно в обратную сторону. А именно, раскроем все скобки. Каждое слагаемое есть уникальная комбинация то что стоит в скобках. Т.е. мы из каждой скобки взяли единицу или степень двойки p. Таким образом итоговое слагаемое есть p в какой-то степени. Эта степень есть сумма степеней двоек. А это двоичная запись какого-то числа, каждое число уникальное, получаем нужную сумму p-шек.
Далее нужно заметить, что $Hash(s_i) == Hash(\overline{s}_i) \cdot (1-p^{2^i-1}) = Hash(s_{i+1}) == Hash(\overline{s}_{i+1})$. Т.к. $s_i - \overline{s}_i = s_{i-1}\overline{s}_{i-1}s_{i-1}$ это расписано по определению.
Т.е. если мы введем функцию f_i – разность соотвествующих хешей, то она будет устроена так: $f_i = f_{i-1} \cdot (1 - p^{2^i-1})$ поскольку $|s_{i-1}| = 2^{i-1}$
Пусть p – четно. Тогда как было разобрано выше, это плохой хеш, поскольку начиная с некоторого места оно занулится. Т.е. у нас модуль – степень двойки, а p-шка – в какой-то большой степени, по этому модулю будет равно 0. Поэтому рассмотрим более содержательный случай с нечетным p.
Заметим, что $(1 - p^4) = (1 - p^2)(1 + p^2) = (1 - p)(1 + p)(1 + p^2)$
Каждая из скобок четная, т.е. имеет 1, 2, 3... i четных сомножителей.
Получаем $1+2+\dots+i = \frac{i(i+1)}{2}$. Мы хотим, чтобы $k \leqslant \frac{i(i+1)}{2}$ это следует из неравенства: $\sqrt{2k} \leqslant i$. Поэтому при $k = 64$. $\sqrt{128} \leqslant i$. Значит, $i = 12$. $2^i = 4096$. Т.е. это длина строки на которой модуль упадет.

32.4. Алгоритм Капуна

Построим антитест для фиксированной пары <p, M>. Пусть у нас алфавит будет состоять из единиц и нулей. $|s_1| = |s_2| = n$. Хотим $Hash(s_1) == Hash(s_2)$.
Перейдем к разности. $0 = Hash(s_1) - Hash(s_2) = \sum p^j \cdot t_j, t_j \in \{-1, 0, +1\}$.
Т.е. имеем некоторую сумму p^i, идея в том, чтобы найти коэффициенты такие, что эта сумма равна нулю по модулю M.
Положим $A_0 : \{p^0, p^1, \dots, p^{n-1}\}$. Если A_0 содержит ноль, тогда тест у нас есть.

А именно: пусть $p^i = 0 \ mod \ M$, для некоторого i.
Тогда тест такой : строка $s_1 = 0 \dots 0 \dots 0$. И строка $s_2 = 0 \dots 1 \dots 0$. Эти строки различаются в i-ой позиции, в остальных совпадают.
Если нуля в множестве A_i нет, тогда построим множество A_{i+1}. Множество A_i мы воспринимаем как массив, поэтому отсортируем по возрастанию.
Далее $A_{i+1} = \{A_i[1] - A_i[0], A_i[3] - A_i[2] \dots\}$. $|A_{i+1}| = \frac{|A_i|}{2}$. Размер нового множества уменьшился в два раза.
Если у нас получилось 0 в множестве A_i тогда мы победили. Почему? Представим себе полное бинарное дерево. Каждый лист которого есть уникальная степень p.
Тогда напишем на ребре +1, если соединяем вершину с левым сыном и -1 если соединяем с правым. Корень дерева имеет высоту $i + 1$. Что означают эти странные цифры на ребрах? Собственно, они служат для определения коэффициента соответствующего p^j. Конкретнее: рассмотрим путь от корня до листа, перемножим числа на пути и получим с каким коэффициентом мы берем каждое p^j. Почему это так? Мы же имеем $A_i[a] - A_i[a-1] = 0$. Что такое $A_i[a]$? Это то что мы взяли со знаком +1, а $A_i[a-1]$ со знаком минус. Это соотвествует двум ребрам в дереве, для этих двух вершин уже все посчитано, мы просто поменяли знак там, где нужно.

Иллюстрация к этому.



Собственно, мы получили, что хотели : некоторую сумму p-шек равную нулю по модулю M. Что с этим делать? Возьмем все p-шки со знаком $+$ и оставим их в левой части, p-шки со знаком - перенесем в правую часть. Тогда алгоритм такой : смотрим на левую часть, напишем в строке s_1 единицу на i-ой позиции, если слагаемое p^i присутствует в левой части, иначе напишем 0. Правило для правой части такое же, т.е. правая часть аналогичным образом задает строку s_2. Одна из этих строк есть инвертированная версия другой.

Пример. Пусть мы получили сумму такую: $p^0 - p^1 - p^2 + p^3 = 0 \ mod \ M \iff$
$p^0 + p^3 = p^1 + p^2$
$s_1 = 1001$
$s_2 = 0110$

Конец алгоритма. Строгого доказательства этого не было. Поэтому тут некоторые его наброски.
Каждый раз при построении множества A_i его размеры уменьшаются в два раза. Что можно сказать о диапазоне чисел? Для A_0 диапазон был таким : $0, \ldots, M-1$.
Диапазон $A_1 \sim \frac{M}{n+1}$, где n - количество чисел в A_0. Почему так? А фиг его знает. Сережа сказал, что разность соседних случайных чисел до M оценивается именно так.
Диапазон $A_2 \sim \frac{M}{(n+1)(\frac{n}{2}+1)}$. И так далее. Для A_i диапазон будет делиться на $\frac{n}{2^{i-1}} + 1$
На пальцах объяснение такое: диапазон схлопнется быстрее, чем размер множества A_i станет нулем, т.е. для каждой пары <p, M> мы гарантированно можем вычислить антихеш тест.

Оценим длину строки, которую мы хотим взять.

$M \leqslant \frac{n}{1} \cdot \frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{4} \cdot \frac{n}{8} \ldots$ То что написано справа – это есть изменение диапазона. Возьмем для простоты $M = 2^m$.
Тогда: $n = 2^k = 2^{\sqrt{2m}}$. Про это следует заметить, что длина строки совпадает с длиной строки Туэ-Морса.
В общем случае делается так: $M \leqslant \frac{n}{1} \cdot \frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{4} \cdot \frac{n}{8} \cdots \sim \frac{n^{\log n}}{2^{\frac{(\log n)^2}{2}}} = \frac{2^{(\log n)^2}}{2^{\frac{(\log n)^2}{2}}} = 2^{\frac{(\log n)^2}{2}}$
$\iff M = 2^{\frac{(\log n)^2}{2}} \iff \log M = \frac{(\log n)^2}{2} \iff \log n = \sqrt{2 \log M} \iff n = 2^{\sqrt{2 \log M}}$.
Подставляя сюда $M = 2^k$ получаем результат, который был выше.

Лекция по алгоритмам #33
Применение хешей

27 апреля

33.1. Алгоритм Рабина-Карпа

Постановка задачи: нам даны две строки s, t, надо найти все вхождения строки s в строку t.

Посчитаем хеш от строки s, обозначим его за $hash_s$. Заметим, что для этого вычисления нам не нужна дополнительная память, нам хватит одной переменной $hash_s$, в которой будет хранится хеш-функция текущего префикса строки s (фактически, мы будем считать хеш как обычно, но так как для пересчета хеша нам нужно лишь значения предыдущего префикса, мы и будем хранить только хеш-функцию предущего префикса). Чтобы подсчитать хеш следующего префикса, надо сделать $hash_s = hash_s \cdot p + s[i]$. В конце в $hash_s$ будет лежать хеш самого длинного префикса, то есть строки s.

Теперь для каждой подстроки t длины $|s|$ надо проверить, не совпадает ли она с s. Надо понять, как по массиву хешей префиксов вычислять хеш от подстроки. Пусть $hast_t[i] = hash(t[0..i))$. Тогда утверждается, что $hash_t([l, r)) = hash_t[r] - hash_t[l] \cdot p^{r-l}$. Доказательство: $hash_t([l, r)) = p^{r-l-1}s[l] + p^{r-l-2}s[l+1] + ... + s[r-1] = (p^{r-1}s[0] + p^{r-2}s[1] + ... + s[r-1]) - (p^{r-1}s[0] + ... + p^{r-l}s[l-1] = hash_t[r] - p^{r-l}(p^{l-1}s[0] + p^{l-2}s[1] + ... + s[l-1]) = hash_t[r] - p^{r-l}hast_t[l]$. Поэтому решение получается простое: мы насчитываем хеши префиксов как обычно (массив $hash_t$), а потом за $O(|t|)$ перебираем подстроку t, имеющую длину s, берем ее хеш за $O(1)$ и сравниваем с $hash_s$. Чтобы считать хеш за $O(1)$ надо еще в начале насчиать массив степеней p за линию. Итого, сейчас мы получили время $O(|s| + |t|)$.

Замечание.

В коде лучше сделать функцию $substr(l, r)$, возвращающую хеш подстроки. Ее тело очень простое:

```
1 int substr(int l, int r) {
2   return ((hash[r] - hash[l] * p[r-l]) % MOD + MOD) % MOD;
3 }
```

Важно помнить, что отрциательное число, взятое по модулю, не обязательно станет неотрицательным. Поэтому после $(hash[r] - hash[l] \cdot p[r - l])\%MOD$ мы могли получить отрицательное число. Я добавил к результату MOD, чтобы он стал точно положительным, и снова взял по модулю. Так работает дольше из-за двух взятий по модулю, от второго можно избавиться, заменим взятие на проверку $(if(x \geqslant MOD)then\ x- = MOD)$. Также если вы постоянно сравниваете подстроки с помощью вызовов функции $substr$, есть смысл завести отдельную функцию для сравенения.

```
1 bool equal(int l1, int r1, int l2, int r2) {
2   return (r1 - l1 == r2 - l2 && ((hash[r1]-hash[l1]*p[r1-l1])-(hash[r2]-hash[l2]*
      p[r2-l2])) % MOD == 0);
3 }
```

Такой код делает одно взятие по модулю вместо двух.

Замечание.

Алгоритм Рабина-Карпа можно писать с $O(1)$ дополнительной памяти (то есть, кроме строк s, t ничего не хранить). Мы уже умеем считать $hash_s$. Так же посчиатем $hash_t([0..|s|))$. Теперь заметим, что мы можем за $O(1)$ пересчитать $hash_t([i+1..i+1+|s|))$, зная $hash_t([i..i+|s|))$. Выведем формулу: $hash_t([i+1..i+1+|s|)) = p^{|s|-1}t[i+1]+p^{|s|-2}t[i+2]+..+t[i+|s|] = p(p^{|s|-1}t[i]+p^{|s|-2}t[i+1]+..t[i+|s|-1])+t[i+|s|]-p^{|s|}t[i] = p\cdot hash_t([i..i+|s|))+t[i+|s|]-p^{|s|}t[i]$. Заметим, что $p^{|s|}$ мы можем один раз насчитать в начале программы без дополнительной памяти. Итого, алгорим такой: считаем хеш от первой подстроки, а дальше двигаем "окошко"длины $|s|$, пересчитывая хеш окошка за $O(1)$, выкидываю первый символ "окошка"и добавляя новый.

33.2. Сравение строк, LCP

Пусть нам даны строки s, t с уже насчитанными хешами на префиксах. Мы хотим за $O(\log(min(|s|,|t|)))$ понять, какая строчка лексикографически меньше. Для решения заметим, что у строк s,t есть максимальный общий префикс длины k (он обозначается $LCP(s,t)$ (Largest Common Prefix)), а в следующем символе они уже отличаются, поэтому если мы найдем k, то дальше мы за $O(1)$ поймем, какая из строк меньше $s<t \leftrightarrow s[k]<t[k]$. Если у строк есть общий префикс длины k, то префиксы длины $\leqslant k$ у строк s,t тоже равны, а если длины k, а если нет длины k, то и большей нет. Это означает, что k можно найти бинпоиском. Мы проверяем, что $s[0..\frac{l+r}{2}] = t[0..\frac{l+r}{2}]$ с помощью хешей (мы уже научились за $O(1)$ брать хеш подстроки). Если строки равны, то двигае левый указатель, иначе двигаем правый. Теперь мы умеем находить наибольшой общий префикс двух строк, а также сравнивать их на больше-меньше.

Замечание. Заметим, что кроме строк s, t, мы можем сравнить и любые их две подстроки за $O(\log(min(|s|,|t|)))$. Для этого надо делать так же бинпоиск по длине общего префикса и проверять, что $s[l..l+k]=t[r..r+k]$ с помощью хешей подстрок.

33.3. Суффиксный массив

Нам дана строчка s. Мы хотим отсортировать все ее суффиксы в лексикографическом порядке. Каждый суффикс однозначано задается позицией в s своего первого символа, поэтому на выходе мы хотим получить перестановку чисел от 1 до n - отсортированные суффиксы.

Решение 1. Отсортируем все суффиксы с помощью $std::sort$, где компаратор будет просто сравнивать суффиксы за $O(n)$ (то есть просто проходим по строкам до первого несовпадения). Решение работает за $O(n^2\log n)$, так как мы делаем $O(n\log n)$ сравнений строк, каждое работает за $O(n)$.

Решение 2. Отсортируем все суффиксы с помощью $std::sort$, где компаратор будет сравнивать суффиксы за $O(\log n)$ (мы учились это делать хешами чуть выше). Такое решение работает уже за $O(n\log^2 n)$. Заметим, что мы используем линейное количество дополнительной памяти (один массив хешей префиксов строки s).

33.4. Применение суффиксного массива

Нам дана строка t, и приходят запросы s_i и надо ответить, является ли s_i подстрокой t.

Построим суффиксный массив строки t. Легко видеть, что строка s_i, если входит в t, является началом какого-то суффикса. Далее заметим, что в суффиксом массиве сначала лежат все суффиксы, меньшие или равные s, а потом большие s (так как все суффиксы отсортированны). Поэтому можно найти эту границу бинарным поиском (пусть k - последний суффикс, который меньше или равен s), а потом проверить, что $t[k..k+|s|)=s$ за один проход (достаточно проверить только k, так как если этот префикс длины $|s|$ этого суффикса не равен

s, то и у любой более высокой в суффиксном массиве строки он тоже не равен (так как эти строки только меньше, чем k)). Как найти границу? Пусть в бинпоиске мы стоим в суффиксе m. Нам надо проверить, больше он s_i или нет. Это можно сделать за линейное время обычным сравением (и получить $O(|s_i| + \log|t| \cdot |s_i|)$), а можно за $O(\log n)$ с помощью хешей (для этого надо их насчитать один раз для t и для каждого s_i) и получить $O(|s_i| + \log|t| \cdot \log|s_i|)$. В обеих асимптотиках учитывается $|s_i|$ - считывание строки, вычисление хешей префиксов, финальное сравнение, $\log|t|$ на бинпоиск, и еще что-то на сравнение строк ($|s_i|$ в первом случае и $|\log s_i|$ во втором).

33.5. Наибольшая общая подстрока двух строк

Даны две строки s, t, необходимо найти их наибольшую общую подстроку. Заметим, что если у них есть общая подстрока длины k, то есть и любой меньшей длины, а если нет длины k? то и большей нет. Поэтому будем делать бинпоиск по длине k. Как проверить, что у строк есть равные строки длины k? Посчитаем хеши все подстрок s длины k (делаем это как в алгоритме Рабина-Карпа, например) и сложим их в хеш-таблицу. Теперь идем по строке t, берем очередную подстроку длины k и проверяем, нет ли ее в хеш-таблице. Если есть, значит, мы нашли общую подстроку длины k. Все, что внутри бинпоиска, работает за линейное время (подсчитать хеши подстрок одной и той же длины можно за линию как в Рабине-Карпе, а хеш-таблица состоит из $O(|s|)$ элементов и мы делаем к ней $O(|t|)$ запросов, поэтому работает за $O(|s| + |t|)$), следовательно, суммарно получаем $O(\log(min(|s|, |t|)(|s| + |t|))$.

Лекция по алгоритмам #34
Суффиксный массив за $\mathcal{O}(n \log n)$ и LCP

23 октября

34.1. Суффиксный массив за $\mathcal{O}(n \log n)$

Что сортируем?

Поймем, что не имеет значения, сортируем мы суффиксы или циклические сдвиги
Сдвиги $\to$ суффиксы: $s \to s\#$ (# - символ, который меньше всего алфавита)
Суффиксы $\to$ сдыиги: $s \to ss$
Отсортируем теперь циклические сдвиги.

Алгоритм за $\mathcal{O}(n^2 + \Sigma)$

Цифровая сортировка.
Пусть уже отсортировали все подстроки длины k в циклической строке.
Переход $k \to k + 1$ выполняем за $\mathcal{O}(n)$.

Улучшение до $\mathcal{O}(n \log n)$

Будем делать переход $k \to 2 \cdot k$.
Пусть $p[i]$ - указатели на начало подстрок длины k в отсортированном порядке.
$color[j]$ - числа, обладающие свойством $color[p[i]] \le color[p[i + 1]]$, причем равенство достигает только в случае равенства строк.
Отсортировать подстроки длины $2 \cdot k$ значит отсортировать $\langle color[j], color[j + k] \rangle$
Отсортируем их за $\mathcal{O}(n)$ (числа от 0 до $n - 1$, мы так хотим)
Получили новый отсортированный порядок. Осталось только пересчитать $color$

```
1     c = 0;
2     for (j = 0 .. n - 1)
3         //"j && .." здесь означает, что j != 0
4         if j && pair[p[j]] != pair[p[j - 1]]
5             c++;
6         color[p[j]] = c;
```

Сортируем пары за $\mathcal{O}(n)$

```
1     for (i = 0 .. n - 1)
2         cnt[color[j]]++;
3     for (j = 1 .. n - 1)
4         pos[j + 1] = pos[j] + cnt[j];
5     for (j = 0 .. n - 1)
6         i = ((p[j] - k) + n) \% n; // переход на k символов левее
7         p_new[pos[color[i]]++] = i;
8         // все и так отсортировано по второй половине
9         // достаточно стабильно отсортировать по первой
```

34.2. Суффиксный массив и LCP

Алгоритм Касаи

Начинаем считать с самого большого суффикса.
Мы знаем, что $LCP[p^{-1}[j]] \geq LCP[p^{-1}[j-1]] - 1$ (p^{-1} по суффиксу возвр. его позицию в суффмассе)
Будем честно считать $LCP[i]$, но начиная с $LCP[i-1]-1$ позиции.

```
1      for (i = 0 .. n - 1)
2          p2[p[i]] = i;
3      k = 0;
4      for (j = 0 .. n - 1)
5          i = p2[j];
6          k--;
7          if (k < 0 || i == n-1)
8              k = 0;
9          if (i != n - 1)
10             while (s[p[i] + k] == s[p[i + 1] + k])
11                 k++;
12         lcp[i] = k;
```

Lm 34.2.1. $Time = \mathcal{O}(n)$

Доказательство. k суммарно уменьшится не более чем на $2 \cdot n$
$k \leq n$ (КО)

Лекция по алгоритмам #35
Суффиксный массив: продолжение

22 апреля

35.1. Алгоритм Каркайнена-Сандерса ($\mathcal{O}(n)$)

Хотим построить суффиксный массив за $\mathcal{O}(n)$, где $n = |s|$, $0 \le s_i \le \frac{3}{2}n$. Сортируем именно суффиксы, а не циклические сдвиги.

35.1.1.

Допишем к строке 3 нулевых символа. Теперь сделаем новый алфавит: $(s_i, s_{i+1}, s_{i+2}) = w_i$. Занумеруем их от 0 до $n-1$ с помощью цифровой сортировки наших троек.

35.1.2.

$t_0 : w_0w_3w_6\dots$
$t_1 : w_1w_4w_7\dots$
$t_2 : w_2w_5w_8\dots$

35.1.3.

Запустим рекурсивно самих себя от строки t_00t_1. Теперь мы умеем сравнивать между собой суффиксы 0-типа и 1-типа.

35.1.4.

Суффикс 2-типа = символ + номер суффикса 0-типа. Отсортируем цифровой сортировкой.

35.1.5.

Осталось сделать merge двух суффиксных массивов. Мы умеем сравнивать между собой суффиксы 0-типа и 1-типа. Суффиксы 0-типа и 2-типа – это первый символ и суффиксы 1-типа и 0-типа, так что мы тоже сумеем их сравнивать. Суффиксы 1-типа и 2-типа – это первый символ и суффиксы 2-типа и 0-типа, их мы только что научились сравнивать. Победа.

35.2. Поиск строки в тексте

$\mathcal{O}(|s| + \log(|text|))$.

35.2.1.

Сделаем бинпоиск по суффиксному массиву, на котором уже насчитали lcp. Будем поддерживать lcp искомой строки и границ бинпоиска. Посмотрим на строку посередине отрезка. Ясно, что $lcp(s,m) \ge max\{min\{lcp(s,l), lcp(l,m)\}, min\{lcp(s,r), lcp(r,m)\}\}$. Так что lcp с m мы начинаем находить с этого максимума, а не с начала строки. $lcp(l,m) = min$ на отрезке, $lcp(r,m)$ – аналогично, два других lcp мы уже знаем. Пусть мы построили дерево отрезков на значениях lcp. Заметим, что каждый раз интересующий нас диапазон уменьшается ровно вдвое. Т.е, мы как бы всё это время спускаемся вниз по дереву отрезков. Т.е., если хранить, в какой вершине мы сейчас стоим, можем получать минимум на отрезке за $\mathcal{O}(1)$.

35.2.2.

Далее, почему все поиски lcp суммарно работают за длину строки. $lcp(s,r)+lcp(s,l)$ не уменьшаются, т.к. не уменьшается ни минимум, ни максимум из этих двух величин ($lcp(s,m)$ больше либо равен минимума, т.е. если он станет новым минимумом, то минимум не уменьшится, если новым максимумом, то аналогично). А как только мы увеличиваем $lcp(s,m)$ руками, т.е., продолжаем втупую идти с некоторого места строки, то сейчас $lcp(s,m) = max\{lcp(s,l), lcp(s,r)\}$.

Лекция по алгоритмам #36
Алгоритм Ахо-Корасик

24 апреля

36.1. Бор

Вспомним, что такое бор. Бор - корневое дерево, в котором рёбра подписаны буквами. Бор нужен для хранения слов (строк) как путей от корня до листа. По умолчанию каждое ребро подписано одной буквой, но, если у какой-то вершины ровно один потомок, можно объединить идущие из неё "наверх" и "вниз" рёбра в одно, на котором будут записаны сначала буквы с первого, а потом - со второго.

Хранить можно, например, массивом Next[s][$\sum$], где s - длина слова, а $\sum$ - размер алфавита.

Способ хранения	Время спуска по строке	Память на ребро
Array	$\mathcal{O}(\|s\|)$	$\mathcal{O}(\|\sum\|)$
List	$\mathcal{O}(\|s\| \cdot \|\sum\|)$	$\mathcal{O}(1)$
TreeMap	$\mathcal{O}(\|s\| \cdot \|\log\sum\|)$	$\mathcal{O}(1)$
HashMap	$\mathcal{O}(\|s\|)$ с большой const	$\mathcal{O}(1)$
SplayMap	$\mathcal{O}(\|s\| + \|\log\sum\|)$	$\mathcal{O}(1)$

36.2. Суффиксная ссылка

Def 36.2.1. *Суффиксная ссылка вершины* ***v*** *- префикс-функция пути до* ***v****, то есть ссылка на вершину* ***u****, в которой оканчивается максимальный путь от корня, совпадающий с каким-то собственным суффиксом пути от корня до* ***v****. Если длина такого максимального пути равна нулю, то суффиксная ссылка указывает на корень.*

36.3. Алгоритм Ахо-Корасик

Алгоритм по набору строк строит бор и считает на нём суффиксные ссылки.

Он делает это. Он делает это. Он делает это! Он это делает.

Бор строить уже научились, осталось посчитать суффиксные ссылки. Рассмотрим 2 способа.

36.3.1. Способ первый - как префикс-функция

При подсчёте суффиксной ссылки вершины **v** будем переходить по уже посчитанной суффиксной ссылке её отца **parent[v]** и сравнивать следующий на пути в **v** символ с тем, что на ребре (**parent[v]**, **v**). Заметим, что если бы в боре была одна строка, то мы бы именно посчитали префикс-функцию от неё.

```
1 Calc(v) {
2   u = suf[parent[v]];
3   while (Next[u][pch[v]] == 0)// no edge
4     u = suf[u];
5   u = Next[u][pch[v]];// Next[u][pch[v]] != 0
6   suf[v] = u;
7 }
```

pch[v] - символ на ребре (**parent[v]**, **v**).

Для того, чтобы не было крайнего случая, заведём фиктивную вершину **null**. Суффиксные ссылки, которые должны были вести в корень, бедут вести в **null**, а из **null** будут все рёбра, причём вести они будут в корень.

Когда мы искали префикс-функцию, мы говорили, что если сейчас рассматривается **i**-я позиция, то есть множество суффиксов отрезка до (**i-1**)-й позиции и мы умеем его перебирать. Перебирали, находили самый длинный, к которому можно преписать символ **i**-й позиции и приписывали. Сделаем то же самое на боре.

```
1 BFS() {
2   v = queue.pop();// returns popped element
3   for (int c = 0; c < alphabet_size; c++)
4     if (Next[v][c])
5       deque.push(Next[v][c])
6   Calc(v);
7 }
```

Онлайн сложно, но можно. Пополняемые структуры данных в помощь.

Теорема 36.3.1. Время работы

Построение работает за сумму длин строк $O(\sum\limits_{i=1}^{n} |s_i|)$.

Доказательство. Для каждого пути от корня до листа бора алгоритм вычисления всех суффиксных ссылок на этом пути работает ровно так же, как вычисление префикс-функции на соответствующей этому пути строке. При вычислении суффиксной ссылки от вершины **v** мы переходим в **parent[v]** и берём её суффиксную ссылку, оказываемся в том же положении, в котором закончили предыдущий шаг. ∎

Важно понимать, что алгоритм работает именно за сумму длин строк, а не за размер бора. Например, может быть "веник" - несколько строк, различающихся только последней буквой.

Применение - поиск нескольких строк (слов) в тексте. Простой вариант - узнать, есть ли хотя бы одна из данных строк в тексте. Для этого не нужно добавлять текст в бор. Достаточно пройтись по нему со следующим циклом.

```
1 while (Next[v][text[i]] == 0)
2   v = suf[v];// going up
3 v = Next[v][text[i]];
```

Lm 36.3.2. Время работы
Проверка, входит ли какое-то из данных слов в текст займёт $\mathcal{O}$(построение бора + $|text|$).

Доказательство. Глубина вершины внутри **while** убывает. ■

Можно было дописать в начало текста ранее отсутствовавший в алфавите символ и, добавляя текст в бор, проверять, не создали ли мы суффиксную ссылку на старый лист. Но это могло бы потребовать очень много памяти.

Мы находим самый длинный суффикс. Но, возможно, был более короткий суффикс, являющийся словарным словом. Если некоторые словарные слова оканчиваются не в листе, то для каждой вершины нужно хранить, терминальная она или нет. Изначально как терминальные помечены те вершины, в которых заканчиваются слова. А в конец алгоритма нужно добавить **is_terminal |= is_terminal[suf[v]]**.

36.3.2. Способ второй - полный автомат

При построении бора и при проходе по тексту мы используем один и тот же код. Дабы не копипастить, вынесем его в отдельную функцию. Эта функция будет считать, куда мы попадём, если захотим спуститься из вершины **v**. Осталось перебрать все вершины. Теперь **while** из прохода по тексту не нужен, ведь **Next[v][text[i]]** - то, где мы хотим оказаться.

```
1 Calc2(v) {
2   suf[v] = Next[suf[parent[v]]][pch[v]];
3   for (int c = 0; c < alphabet_size; c++)
4     if (Next[v][c] == 0)
5       Next[v][c] = Next[suf[v]][c];
6 }
```

Lm 36.3.3. Время работы и память
Время работы $=$ память $= \mathcal{O}(|\text{бор}| \cdot |\sum|)$.

Доказательство. В каждой вершине выполняется $\mathcal{O}(1)$ действий, а потом - ещё по $\mathcal{O}(1)$ действий на каждый символ алфавита. ■

Лекция по алгоритмам #37
Суффиксное дерево. Алгоритм Укконена.

26 апреля

37.1. Суффиксный бор, дерево

Def 37.1.1. *Суффиксный бор - бор всех суффиксов строки.*

Def 37.1.2. *Сжатый бор - бор, в котором у всех вершин, кроме листов есть хотя бы два сына. При этом на рёбрах теперь написаны не буквы, а строки.*

Def 37.1.3. *Суффиксное дерево - сжатый суффиксный бор.*

Lm 37.1.4. Если сделать из строки s строку $s\#$, где $\#$ - символ, который не встречается в строке, то все её суффиксы будут заканчиватся в листах.

37.2. Элементарный алгоритм построения

Рассмотрим следующий алгоритм построения суффиксного дерева за $\mathcal{O}(n^2)$.

```
1 for(i,n)
2   Add(s[i:n])
```

Где *Add* - функция, которая за линию спускается по дереву добавляя суффикс.

Lm 37.2.1. $SuffTree(s)$ содержит не более $2|s|$ вершин.

Доказательство. Новые вершины создаются, только когда *Add* в процессе спуска не может перейти по очередному символу. В этот момент создаётся не более двух вершин. Первая в середине ребра, если в момент неудачи функция стояла посередине ребра и вторая - вершина-конец суффикса. ∎

Техническая деталь: позицию в дереве хранить удобнее всего как пару (e, x) - номер ребра, и позицию на нём.

37.3. Алгоритм Укконена

Внимательно посмотритм на жизненный цикл суффикса в суффиксном дереве:

1. Рождение суффикса в корне дерева.
2. Спуск конечной вершины суффикса по дереву.
3. Разветвление суффикса в момент, когда дальнейший спуск невозможен.
4. Свободное произрастание в личине листа дерева.

37.3.1. Реализация алгоритма Укконена за $\mathcal{O}(n^2)$

На каждом шаге будем поддерживать суффиксное дерево строки $s[0 : i]$. Будем помнить позиции концов всех суффиксов в дереве. При добавлении символа будем пытаться продлить каждый суффикс на этот символ. Если надо будем разветвляться.

37.4. Улучшение до $\mathcal{O}(n)$

Lm 37.4.1. Если в ребро ведущее в лист написать либо $[l, inf)$, либо $[l, \&n]$, то они будут расти сами без нашего вмешательства.

Lm 37.4.2. Суффикс $s[i : n]$ находится в состоянии (2) $\Rightarrow \forall j > i$ $s[j : n]$ тоже в состоянии (2).

Доказательство. Если бы какой-то более короткий суффикс ответвился по символу, то данный длинный суффикс бы тоже ответвился. ■

Это подталкивает нас к мысли следить только за самым длинным неразветвившимся суффиксом. Получается следующий алгоритм:

```
suffarr(){
  for(i,n)
    addchar(s[i])
addchar(c){
x:
  if succesful descend with char c
    pass
  else
    create new vertex
    switch to next suffix
    goto x
}
```

Осталось объяснить что такое *switch to next suffix*. Это значит уменьшить на единичку длину текущего суффикса, за которым мы следим и перейти к нему в дереве.

Как же мы перейдём в дереве? Поднимемся до вершины из которой исходит ребро, на котором мы стоим. Перейдём по суффиксной ссылке. Эта суффиксная ссылка обязательно ведёт в вершину, т.к. если подстрока кончается в вершине суфф. дерева, то её суффиксы тоже кончаются в вершинах. Теперь спустимся вниз на строку, на которую поднялись. Спускаться будем не по каждому символу отдельно, а быстро по рёбрам.

А ещё в какой-то момент нужно пересчитать суффиксные ссылки свежесозданных вершин. Например, можно при спуске сложить все вершины на стек, а потом проставить ссылки разматывая этот стек.

Теперь нужно оценить временную сложность всех наших спусков.

Lm 37.4.3. Все спуске в сумме отработают за $\mathcal{O}(n)$.

Доказательство. Рассмотрим потенциал $\phi(v)$ = количеству вершин на пути от корня до v. $phi(v) \le n$. С другой стороны, за каждый спуск к следующему суффиксу потенциал может уменьшиться не более, чем на 1. Т.е. $phi(v) - 1 \le phi(suf(v))$. Это так, потому что суффиксная ссылка из каждой вершины ведёт в свою вершину, кроме, возможно, самой неглубокой вершины из которой ссылка может вести в корень. ■

Лекция по алгоритмам #38
Хеширование

29 апреля

38.1. Введение

Def 38.1.1. *Хеш-функция.*
Сжимающее отображение $h : U \to M$, $|U| > |M|$, $|M| = m$.

Def 38.1.2. *Универсальная система хеш-функций.*
H - множество хеш-функций:
Для случайной $h \in H$, $\forall x, y$ $x \neq y$: $\underset{h \in H}{pr}[h(x) = h(y)] \leq \frac{1}{m}$.

Def 38.1.3. *Универсальная система хеш-функций 2.*
$\sum\limits_{h \in H} [h(x) = h(y)] \leq \frac{|H|}{m}$.

Теорема 38.1.4. $H_{p,m} = \{(a, b) : x \to ((ax + b) \mod p) \mod m\}$, $a \in [1, p), b \in [0, p)$ - универсальное для $|U| \leq p$.

Доказательство. Зафиксируем входные $x, y : x \neq y$.

$$\begin{cases} ax + b \equiv r \pmod p \\ ay + b \equiv s \pmod p \end{cases}$$

$\sum\limits_{(a,b)} = \sum\limits_{r \neq s} [r \equiv s \pmod m] \leq \frac{p(p-1)}{m} = \frac{|H|}{m}$

■

38.2. Хеш-таблица

Мы будем рассматривать реализацию на списках, которая использует случайную хеш-функцию h из универсальной системы H.

n - количество элементов в таблице, m - количество списков.

Оценим количество коллизий и матожидание длины списка.

1. Количество пар $x, y : x \neq y, h(x) = h(y)$
   $E \leq \frac{n(n-1)}{2} \cdot \frac{1}{m} \leq \frac{n-1}{2}$
2. Для фиксированного x, матожидание длины списка $h(x)$
   $E \leq (n - 1)\frac{1}{m} + 1 \Rightarrow$

Add: $\mathcal{O}(1)$
Find, Del: $E = \mathcal{O}(1)$

38.3. Хеширование Кукушки

Add: $E = \mathcal{O}(1)$

Find, Del: $\mathcal{O}(1)$ в худшем случае

Заведём массив размера m и две хеш-функции: $h_1, h_2 \in H$.

Find(x): достаточно найти в любой из $h_1(x), h_2(x)$.

Del(x): аналогично.

Add(x): добавляем в любую из свободных $h_1(x), h_2(x)$. Если заняты обе - начинаем перемещать элементы в альтернативные ячейки. Берём элемент из $h_1(x)$ или $h_2(x)$ и перемещаем его в ячейку, соответствующую его второй хеш-функции, а в текущую помещаем новый элемент. Повторяем пока все элементы не найдут место. Если зациклились, то берём новые случайные h_1, h_2 и перестраиваем всю таблицу.

Если представить себе граф с рёбрами между $h_1(x)$ и $h_2(x)$, то можно понять, что циклов мало.

38.4. Perfect Hashing

Даны n элементов, необходимо построить хеш-функцию, сопоставляющую им числа от 0 до m-1 без коллизий. $n \le m$.

1. Двухуровневая схема
   Возьмем $h \in H_{p,n} : x_i \to 0..n-1$.
   Пусть в i-ой ячейке k_i элементов. Заведём $h_i : x_i \to 0..k_i^2 - 1$.

   <u>Lm</u> 38.4.1. Pr[у h_i нет коллизий] $\ge \frac{1}{2}$

   Доказательство. $E = \frac{k_i(k_i-1)/2}{k_i^2}$ ∎

   $$E[\sum k_i^2] = E\left[\sum_{x,y}[h(x) = h(y)]\right] = \sum_{x,y}\Pr[h(x) = h(y)] = n + n(n-1)\frac{1}{n} = 2n - 1$$

   <u>Lm</u> 38.4.2. $\Pr[\sum k_i^2 \le 4n] \ge \frac{1}{2}$

2. Ацикличный граф
   x_i - ребро между $h_1(x_i)$ и $h_2(x_i)$.
   m вершин и n рёбер.

   <u>Lm</u> 38.4.3. $m \ge 3n \Rightarrow$ Pr[нет циклов] $\ge \frac{1}{2}$

   $hash(x_i) = (f[h_1(x_i)] + f[h_2(x_i)]) \mod m$, где f - значение в вершине графа, которые проставляются обходом в глубину.

Лекция по алгоритмам #39
Игры

05 мая

39.1. Симметричные игры на графе

Def 39.1.1. *орграф + выделенная вершина*
ход – сдвиг по ребру,
lose: degout 0

Def 39.1.2. *Upgrade:*

- *мн-во вершин win*
- *мн-во вершин lose*
- *мн-во никаких*

39.1.1. Решение для ациклического орграфа

Lm 39.1.3. win $\Leftrightarrow \exists$ ход в lose
lose $\Leftrightarrow \nexists$ ход в win

Динамика.
База:
dp[v] = 1 $\Leftrightarrow$ win
dp[v] = 0 $\Leftrightarrow$ lose
Переход:
dp[v] = OR $\{dp[u_i]\}$
Порядок вычисления: или dfs или topsort

Замечание: можно и для upgrade

39.1.2. Решение с циклами (ретроанализ)

```
queue <- WIN, LOSE

while !queue.empty()
  v <- queue
  if lose[v]:
    for x <- v,
      win(x)
  if win[v]:
    for x <- v :
      count[x]--
      if !count[x] and !wiv[x]:
        lose(x)
```

Lm 39.1.4. N ⇔ DRAW, где N – не помеченные

N $\nexists$ → LOSE (иначе вершина попала бы в WIN)

N $\exists$ → N (есть ребро и не в LOSE, и не в WIN (иначе мы вершина была бы в LOSE); можно перейти в N (ничейную))

39.1.3. Прямая сумма игр

Def 39.1.5. *прямая сумма игр:* $\langle G1, v1 \rangle \times \langle G2, v2 \rangle$

2 графа ходим либо в первом либо во втором.

Def 39.1.6. *функция Гранди:* $f[v] = mex\{f[x_1] \cdots f[x_k]\}$

// mex множества A считается за $\mathcal{O}|A|$

Lm 39.1.7. f[v] = 0 ⇔ LOSE

Индукция,

0$\nexists$ → 0,

!0$\exists$ → 0.

// Корректность показывается так же, как и в решении для ациклического графа.

Теорема 39.1.8. $f(v1) \oplus f(v2) = f(\langle v1, v2 \rangle)$, без док-ва

// функция Гранди для суммы игр – xor функций Гранди этих игр.

39.1.4. Ним

n кучек

в i-й a_i камней

за 1 ход нужно взять ненулевое число камней из выбранной кучки.

Лекция по алгоритмам #40
Быстрое проеобразование Фурье

6 мая

40.1. Немного о многочленах

Давайте в рамках этого конспекта под *степенью* многочлена понимать количество коэффициентов (т.е. старшая степень + 1). Известно, что в поле комплексных чисел между многочленами *степени* не больше n и наборами их значений в n точках существует биекция.

40.2. Идея

Пусть у нас есть два многочлена P и Q степеней n и m. Мы хотим их перемножить. Степень их произведения будет $m+n-1$. Если мы посчитаем значения многочлена $T = PQ$ в $m+n-1$ точке, то по ним мы сможем однозначно восстановить T. Чтобы узнать значение T в точке x достаточно перемножить значения P и Q в точке x. Итак, алгоритм: считаем значения P и Q в достаточно большом количестве точек, перемножаем их, по полученным точкам восстанавливаем многочлен T. Хотя зачем нам всё это? Считать значения многочлена степни n в n точках, да ещё и потом обратно восстанавливать... Это ж долго! А за квадрат можно и в столбик было перемножить.

40.3. Мотивация

Оказывается, если брать не какие-то точки, а специальные, то посчитать значения и восстановить по ним многочлен можно быстро, за $\mathcal{O}(n \log n)$. Преобразование из массива коэффициентов многочлена в значения этого многочлена в *этих точках* называется дискретным преобразованием Фурье.

40.4. Быстрое преобразование Фурье

// Напомним, что $e^{ix} = \cos x + i \sin x$
// $(e^{ix})^n = e^{ixn}$

Пусть P – многочлен степени n. Пусть n – степень двойки (если нет, дополним до ближайшей сверху). Посчитаем значения P в точках $e^{-\frac{2\Pi i}{n} j}$ для всех j от 0 до $n-1$. Обозначим за P_0 – многочлен из коэффициентов P с чётными номерами, P_1 – с нечётными. Например,

$$P = 10 + 1x + 2x^2 + 3x^3 + 4x^4 + 5x^5 + 6x^6 + 7x^7$$

$$P_0 = 10 + 2x + 4x^2 + 6x^3$$
$$P_1 = 1 + 3x + 5x^2 + 7x^3$$

.

Методом пристального взгляда поймём, что $P(x) = P_0(x^2) + xP_1(x^2)$.

Точки, в которых мы считаем значения, являются корнями 2^k-ой степени из 1. При возведении в квадрат они, внезапно, превращаются в корни 2^{k-1}-степени из 1 (в следствие периодичности sin/cos, квадраты j-ого и $(j + 2^{k-1})$-ого по номерам корней совпадут). Степени P_0 и P_1 – 2^{k-1}. Мы хотим посчитать их значения в корнях 2^{k-1}-степени из 1. А как раз это мы

умеем находить быстрым преобразованием Фурье (рекурсивный вызов). Зная значения P_0 и P_1 в нужных точках, можно за линию найти значения P. Глубина рекурсии – $\mathcal{O}(\log n)$. На каждом в сумме уровне $\mathcal{O}(n)$ операций. Тогда время работы $\mathcal{O}(n \log n)$.

40.5. Медленная реализация БПФ

Ниже приведён псевдокод быстрого преобразования фурье и умножения многочленов через него. Эту реализацию можно сильно соптимизировать. Об этом нам расскажут в следующих конспектах.

```
1  FFT(n, array)
2      if (n == 1)
3          return array[0]
4      FOR (i = 0 ... n - 1)
5          b[i & 1].push_back(array[i])
6      f0 = FFT(n / 2, b[0])
7      f1 = FFT(n / 2, b[1])
8      FOR (j = 0 ... n - 1)
9          f[j] = f0[j % (n/2)] + exp(2 * Pi * i * j / n) * f1[j % (n/2)]
10     return f
11
12
13 MUL(n, a, m, b)
14     // ^ is not xor here
15     k : 2^k >= n + m
16     // append zeros to a and b to do they long enough
17     f_a = FFT(2^k, a)
18     f_b = FFT(2^k, b)
19     FOR (i = 0 ... 2^k-1)
20         f_c[i] = f_a[i] * f_b[i]
21     c = FFT(2^k, f_c)
22     reverse(c + 1, c + 2^k)
23     FOR (i = 0 ... 2^k-1)
24         c[i] = round(c[i] / 2^k)
25     return c
```

Лекция по алгоритмам #41
Действительно Быстрое проеобразование Фурье

6 травня

Приведённый выше код БПФ за $\mathcal{O}(n\log n)$ решает поставленную задачу, но есть несколько недостатков:

- По-настоящему быстрым его назвать нельзя;
- Непонятно, почему обратное преобразование вообще корректно работает.

- ***Комментарий***

Но где доказательства?

41.1. Фурье обращается просто

Начнём с обращения. Пусть $w = e^{\frac{2\pi i}{n}}$, a_i — коэффициенты многочлена, а f_i — его значение в точке w_i. Заметим, что:

$$f_0 = a_0 + a_1w^0 + a_2w^0 + a_3w^0 + \dots$$
$$f_1 = a_0 + a_1w^1 + a_2w^2 + a_3w^3 + \dots$$
$$f_2 = a_0 + a_1w^2 + a_2w^4 + a_3w^6 + \dots$$
$$f_3 = a_0 + a_1w^3 + a_2w^6 + a_3w^9 + \dots$$

и так далее...

<u>Lm</u> 41.1.1. $\sum f_i = a_0 \cdot n$

Доказательство. Действительно, можно понять, что в каждом столбце, кроме первого сумма равна нулю. Сумма в k-м столбце:

$$a_k(1 + w^k + w^{2k} + \dots + w^{k(n-1)}) = a_k\frac{1 - w^{kn}}{1 - w} = 0$$

<u>Lm</u> 41.1.2. $\sum\limits_{i=0}^{n-1} f_i w^{-ik} = a_k \cdot n$

Доказательство. В j-м столце слагаемые приняли вид $a_i w^{i(j-k)}$. В k-м столбце сумма, очевидно, равна $a_k \cdot n$, а в остальных, аналогично доказательству предыдущей леммы, — 0.

Как мы видим, обратное преобразование действительно очень похоже на прямое, осталось только понять, что $w^{-k} = w^{n-k}$, поэтому после прямого преобразования мы получим $(a_0, a_{n-1}, \dots, a_1)$, следовательно, нужно развернуть элементы со второго по последний.

- ***Комментарий***

Вы понимаете, о чём говорят все эти люди? Я нет.

41.2. Фурье погрешен

Пусть мы коэффициенты перемножаемых многочленов не больше M, тогда коэффициенты произведения не больше nM^2. Ясно, что, чтобы алгоритм корректно посчитал результат, используя тип `double` для хранения вещественной и мнимой частей комплексных чисел, необходимо, чтобы эта величина не превосходила 10^{15} (для `long double` — 10^{18}). Но можно сказать кое-что поточнее.

Эмпирический факт Если $nM^2 \leqslant 10^{16}$, то всё на ура посчитается в `long double` с ошибкой, по модулю меньшей 0.5 (то есть округление до целого будет работать корректно).

- ***Комментарий***

Сильное заявление. Проверять я его, конечно, не буду.

41.3. Коротко о длинных числах

Заметим несколько вещей:

1. Длинные целые числа — те же многочлены, только уже вычисленные в точке $BASE$ (основание системы счисления), следовательно их можно перемножать абсолютно так же. Следует, однако, не забыть сделать переносы в старшие разряды.
2. Вещественные числа — те же целые числа, только домноженные на $BASE^{-k}$, следовательно проблем с их перемножением тоже быть не должно.
3. Наконец, можно упомянуть, что в STL уже реализованы комплексные числа, называются они `std::complex`. Тем не менее, собственноручно написанная реализация комплексных чисел будет почему-то работать быстрее.

- ***Комментарий***

Я не хочу этот банан.

41.4. Фурье оптимизируется и избавляется от ненужной рекурсии

41.4.1. $2\mathbb{R} = 1\mathbb{C}$

До сих пор мы БПФ, работающему с комплексными числами давали на вход вещественные числа и игнорировали комплексную часть. Пора исправить это недоразумение.

Пусть мы хотим сделать ДПФ для вещественных последовательностей a_j и b_j. Построим последовательность $c_j = a_j + ib_j$. Посчитаем по нему ДПФ и, получив fc_j, хотим посчитать fa_j и fb_j.

Lm 41.4.1.

$$fa_j = \frac{1}{2}(fc_j + \overline{fc_{n-j}})$$

$$fb_j = \frac{1}{2i}(fc_j - \overline{fc_{n-j}})$$

Доказательство.

$$fc_j = c(w^j) = a(w^j) + ib(w^j) = fa_j + ifb_j$$

$$\overline{fc_{n-j}} = \overline{a(w^{n-j})} + \overline{ib(w^{n-j})} = a(w^j) - ib(w^j) = fa_j - ifb_j$$

Таким образом, мы из двух запусков БПФ сделали один, ну это ли не чудо?

• ***Комментарий***

Звучит неплохо.

41.4.2. cos и sin — долгая штука

Каждый раз вычислять w^k слишком долго, ведь $e^{ix} = \cos x + i \sin x$, а вычисление тригонометрических функций занимает время. К счастью, так как мы хотим всегда вычислять корни из 1 степени 2^k, мы можем предпосчитать в массиве `root` корни степени n, а корни других степеней уже будут в этом массиве.

• ***Комментарий***

Честно сказать, не хочется комментировать эту ситуацию.

41.4.3. Нерекурсивная реализация

На первом шаге БПФ по данному массиву создаёт два: из коэффициентов с чётными номерами и из коэффициентов с нечётными. А давайте не будем создавать новые массивы, а просто поместим первый из них в левую половинку исходного массива, а второй — в правую. На следующих шагах будем действовать аналогично, более того, потребуем, чтобы результат вызова БПФ лежал на месте входного массива.

Мы получили inplace алгоритм, но не будем останавливаться на достигнутом. Заметим, что при спуске в рекурсии происходит перестановка элементов массива, а при подъёме — комбинация двух результатов следующего уровня. Сейчас мы избавимся от спуска, сразу переставив элементы на их окончательные места, и сделаем один подъём, комбинируя результаты параллельно.

Нетрудно понять, что i-й элемент переход на место $rev[i]$, где $rev[i]$ — число полученное разворотом битовой записи числа i. Вот как считать $rev[i]$:

```
1 for i = 1..n-1
2     rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << k) // k = log n
```

Подъём будет производиться в несколько шагов, на t-м шаге части длины 2^t будут попарно объединяться в части длины 2^{t+1}. Впрочем, хватит слов, прочитайте код, и всё сразу станет ясно.

```
 1 FFT(a, n)
 2     for i = 0..2**k-1
 3         f[rev[i]] = a[i]
 4     for t = 0..k-1
 5         // f[0..2**t)[2**t..2**(t+1)) -- recursive calls
 6         // f[0..2**(t+1))             -- that's what we want to get
 7         for (i = 0; i < 2**k; i += 2**(t+1))
 8             for j = 0..2**t-1
 9                 tmp = f[i+j+2**t] * root[j << (k - t - 1)]
10                 f[i+j+2**t]=f[i+j]-tmp
11                 f[i+j]=f[i+j]+tmp
```

• ***Комментарий***

У меня есть запасной вариант, и я им воспользуюсь.

Лекция по алгоритмам #42
Длинная арифметика

13 мая

Lm 42.0.2. многочлены $\Leftrightarrow$ целые неотрицательные $\Leftrightarrow$ целые со знаком $\Leftrightarrow$ вещественные

Длинные числа можно хранить как массив фиксированной (`a[N]`) или динамической (`vector<int> a`) длины.

42.1. Простые операции

За $\mathcal{O}(n)$:

- длинное [$+ | - | <$] длинное
- длинное [$\cdot | \div$] короткое ($0 \le x < BASE$) (одна цифра в системе счисления $BASE$)

Псевдокод умножения длинного на короткое (цифру):

```
1  operator*(a, x) -> result:
2      // сначала без переносов
3      for i in [0; N):
4          a[i] *= x
5
6      // теперь делаем переносы
7      for i in [0; N - 1):
8          if a[i] >= BASE:
9              a[i + 1] += a[i] / BASE
10             a[i] %= BASE
11
12     return a
```

Псевдокод деления длинного на короткое:

```
1  div(a, x) -> (result, rest):
2      var carry = 0
3      for i in [0; N).reversed():
4          a[i] += carry * BASE
5          carry = a[i] % x
6          a[i] /= x
7
8      return (a, carry)
```

42.2. Умножение длинных чисел

1. за $\mathcal{O}(nm)$, эффективно при небольшой длине ($length \le C$, где $C \approx 8$)

2. за $\mathcal{O}(n^{\log_2 3}) = \mathcal{O}(n^{1.5849\ldots})$, используя трюк Карацубы

3. за $\mathcal{O}(n \log n)$, используя быстрое преобразование Фурье

Псевдокод умножения многочленов за $\mathcal{O}(nm)$:

```
operator*(a, b) -> result:
    var an = a.length
    var bn = b.length

    var c = new BigInt(length = an + bn)

    for i in [0; an):
        for j in [0, bn):
            c[i + j] += a[i] * b[j]

    // не нужно, если используются массивы фиксированной длины
    while c.length > 1 && c.back == 0:
        c.pop_back
```

Псевдокод умножения длинного на длинное за $\mathcal{O}(nm)$, ускоренный для 64-битной архитектуры.

```
operator*(a, b) -> result:
    const BASE = 1e9
    const K = 7
    const M = K * BASE^2

    var an = a.length
    var bn = b.length

    var c = new BigInt(length = an + bn)

    for i in [0; an):
        for j in [0, bn):
            c[i + j] += a[i] * b[j]
            if c[i + j] >= M:
                c[i + j] -= M
                c[i + j + 2] += K

    for i in [0; an + bn - 1):
        c[i + 1] += c[i] / BASE
        c[i] %= BASE

    // не нужно, если используются массивы фиксированной длины
    while c.length > 1 && c.back == 0:
        c.pop_back
```

42.3. Деление длинных чисел за $\mathcal{O}(nm)$

Псевдокод деления многочленов:

```
div(a, b) -> (result, rest):
```

```
2
3     var an = a.length
4     var bn = b.length
5
6     for i in [bn - 1; an - 1].reversed():
7         x = a[i] / b[bn - i]
8         for j in [0; bn):
9             a[i - j] -= x * b[bn - j - 1]
10        c[i - bn - 1] = x
11
12    return (c, a)
```

42.4. Двоичная арифметика

Возведение в степень $= \log k$ умножений, где k - степень, в которую возводим (не длина)
Умножение $= \log k$ сложений, где k - число, на которое умножаем.
$\log k = \Theta(k.length)$
Псевдокод деления за $\mathcal{O}(n^2)$:

```
1  div(a, b) -> (result, rest):
2      if a < b:
3          return (0, a)
4
5      (c, r) = div(a, 2b)
6      c *= 2
7
8      if r >= b:
9          r -= b
10         c += 1
11
12     return (c, r)
```

Пример:
$(103/16) \to (6, 7)$
$(103/32) \to (3, 7)$
$(103/64) \to (1, 39)$
$(103/128) \to (0, 103)$

Псевдокод gcd за $\mathcal{O}(nm)$:

```
1  gcd(a, b) -> result:
2      if a == 0:
3          return b
4      if b == 0
5          return a
6
7      if a % 2 == 0 && b % 2 == 0: // проверяем за O(1)
8          return gcd(a / 2, b / 2) * 2
9
10     while a % 2 == 0:
11         a /= 2
12     while b % 2 == 0:
13         b /= 2
14
```

```
15      if a < b:
16          swap(a, b)
17      return gcd(a - b, b)
```

Теорема 42.4.1. Все 3 функции за $\mathcal{O}(\frac{(\log_2 n)^2}{\omega})$, где n - число (не длина).

Доказательство. Копирование происходит за $\mathcal{O}(\frac{\log_2 n}{\omega})$ ■

Лекция по алгоритмам #43
Деление длинных чисел

6 мая

Обозначения: n — длина делимого в двоичной системе, m — длина делителя в двоичной системе, w — длина машинного слова (пропорциональна основанию нашей системы счисления)

43.1. Двоичный поиск по частному

Двоичным поиском перебираем частное. Логарифм частного — его длина, т. е. примерно $n-m$. Для проверки используем умножение в столбик, получаем асимптотику $\mathcal{O}(m(n-m)^2/w^2)$. На w делим потому что длины чисел в нашей системе счисления в w раз меньше, чем в двоичной.

43.2. Двоичный поиск по цифре

Делим в столбик. Цифру подбираем двоичным поиском, который работает за $\mathcal{O}(w)$. Асимптотика — $\mathcal{O}(n(n-m)/w)$.

43.3. Деление за честный квадрат

Научимся получать цифру без бинпоиска.

Пусть надо разделить $X = \overline{x_0 x_1 \dots x_{n-1}}$ на $Y = \overline{y_0 y_1 \dots y_m}$. Разделим два вещественных числа $x = \overline{x_0, x_1 x_2 \dots}$ на $y = \overline{y_0, y_1 y_2 \dots}$ с нужной точностью и домножим на B в нужной степени (B — основание системы счисления).

Важное замечание. В алгоритме x_0 не всегда будет цифрой, но будет сохраняться инвариант: $x_0 \leqslant y_0 B$.

Пусть $x/y = z = \overline{z_0, z_1 z_2 \dots}$. Сперва получим z_0. Известно, что $z_0 = \lfloor \frac{x}{y} \rfloor$ но как посчитать $\lfloor \frac{x}{y} \rfloor$? Оценим z_0 снизу чем-то, что мы можем быстро посчитать: $z_0 \geqslant \lfloor \frac{x_0 + x_1 B^{-1}}{y_0 + y_1 B^{-1} + B^{-1}} \rfloor = \lfloor \frac{x_0 B + x_1}{y_0 B + y_1 + 1} \rfloor =: L$ (числитель – уменьшенный x, знаменатель — увеличенный y). Оказывается, L отличается от z_0 несильно.

Теорема 43.3.1. $z_0 - L \leqslant 4$.

Доказательство. Аналогично нижней границе существует и верхняя: $R := \lfloor \frac{x_0 B + x_1 + 1}{y_0 B + y_1} \rfloor \geqslant z_0$.

$$z_0 - L \leqslant R - L = \lfloor \frac{x_0 B + x_1 + 1}{y_0 B + y_1} \rfloor - \lfloor \frac{x_0 B + x_1}{y_0 B + y_1 + 1} \rfloor \leqslant \frac{x_0 B + x_1 + 1}{y_0 B + y_1} + 1 - \frac{x_0 B + x_1}{y_0 B + y_1 + 1} =$$

$$= \frac{x_0 B + x_1 + y_0 B + y_1}{(y_0 B + y_1)(y_0 B + y_1 + 1)} + 1 \leqslant \frac{y_0 B^2 + (B-1) + y_0 B + (B-1)}{y_0^2 B^2} + 1 \leqslant 4$$

■

Таким образом, мы можем за $\mathcal{O}(1)$ вычислить L и перебрать следующие 4 цифры. Псевдокод:

```
1 x -= y * L;
2 while (x >= y)
3   x -= y, ++L;
```

После вычитания yz_0 из x сдвигаем запятую в a вправо на одну позицию, поддерживая инвариант $x_0 \leqslant y_0 B$. После этого так же получаем z_1, z_2 и так далее.

Для получения одной цифры мы использовали константное число линейных операций (вычитания и умножения на цифру). Асимптотика — $\mathcal{O}(n(n-m)/w^2)$.

43.4. Деление за $\mathcal{O}(n \log^2 n)$

Для понимания рекомендуется сначала ознакомиться с предыдущим алгоритмом.

Воспользуемся тем же фокусом, что и в предыдущем алгоритме, только получим не одну цифру, а сразу $n/2$. Для этого поделим $x' := \overline{x_0, x_1 \dots x_k}$ на $y' := \overline{y_0, y_1 \dots y_k}$ ($k := \frac{n}{2} + 1$), а точнее, получим $L := \frac{x'}{y' + B^{-k}}$ — нижнюю границу для z.

$z' := \overline{z_0, z_1 \dots z_{k-1}}$

$z - z' \leqslant B^{-(k-1)}$, поэтому $L - B^{-(k-1)}$ является нижней границей для z'.

Теорема 43.4.1. $z' - (L - B^{-(k-1)}) \leqslant 3B^{-(k-1)}$

Доказательство.

$$R := \frac{x' + B^{-k}}{y'} \geqslant z \geqslant z'$$

$$R - (L - B^{-(k-1)}) \leqslant \frac{x' + B^{-k}}{y'} - \frac{x'}{y' + B^{-k}} + B^{-(k-1)} = \frac{(x' + y' + B^{-k})B^{-k}}{y'(y' + B^{-k})} + B^{-(k-1)} \leqslant$$

$$\leqslant (x' + y' + B^{-k})B^{-k} + B^{-(k-1)} \leqslant 2B \cdot B^{-k} + B^{-(k-1)} = 3B^{-(k-1)}$$

■

Таким образом, с помощью рекурсивного вызова от чисел вдвое меньшей длины мы получили почти точно z'. Чтобы получить z' точно, нужно поступить аналогично предыдущему алгоритму и повычитать $yB^{-(k-1)}$ из x ещё некоторое (константное) количество раз.

Итак, мы получили z' — первую часть цифр, осталось получить вторую часть. Пусть $\frac{x}{y} = z' + z'' \cdot B^{-k}$, где $z'' < B$. Тогда x Тогда $z'' = \frac{(x - z'y)B^k}{y}$. Нас интересуют первые k цифр числа z'', поэтому поступим так же, как и раньше, разделив первые $k+1$ цифр $(x - z'y)B^k$ на первые $k+1$ цифр y рекурсивным вызовом и уточнив затем результат.

Оценим время работы алгоритма.

$T(n) = \mathcal{O}(n \log n) + 2T(n/2 + 1)$, где $n \log n$ — время работы умножения z' на y, используем FFT.

$T(n) = \mathcal{O}(n \log^2 n)$.

Существует также алгоритм деления за $\mathcal{O}(n \log n)$.

Лекция по алгоритмам #44
Классы сложности

24 мая

44.1. Типы задач

Пусть X — множество всех конечных строк. Тогда задачи делятся на два типа:

1. `Поиск (Search Problem)`
   $X \mapsto Y$
2. `Распознавание (Decision Problem)`
   $X \mapsto \{0, 1\}$

Займемся решением задачи распознавания, т.к. очевидно, что через нее можно выразить задачу поиска.

44.2. Классы сложности задач

1. `P`
   $\exists$ полиномиальное решение задачи, т.е. алгоритм, проверяющий, лежит ли строка s в множестве всех строк L, для которых ответ на задачу 1

2. `NP`
   Есть два равноправных определения. Первое — историческое, второе — практическое.

   1. Деревом возможностей работы алгоритма назовем дерево конечной глубины, каждый узел (кроме листьев) которого соответствует случайному выбору поддерева дальнейших вычислений, а ребра и листья являются детерминированными полиномиальными алгоритмами. Естественно, не утверждается, что у любого алгоритма существует такое дерево, но очевидно, что любому такому дереву соответствует алгоритм.
      Задача лежит в классе `NP` если существует дерево возможностей работы алгоритма (и, соответственно, алгоритм A) такое, что $s \in L \Leftrightarrow \exists$ ветка вычислений такая, что $A(s) = 1$.
   2. Задача лежит в классе `NP` если существует полиномиальный алгоритм $R(x, y) : x \in L \Leftrightarrow \exists y : |y| = poly(|x|) \wedge R(x, y) = 1$

   Из второго определения первое следует очевидным образом (просто угадываем y), а обратное следствие так же очевидно (y — последовательность выборов, $|y|$ точно меньше глубины дерева)

44.3. Сравнение задач

$A \leq B$ (Задача A сводится к задаче B) если $\exists f \in \texttt{poly} : \forall x A(x) = B(f(x))$
Очевидно, что это отношение эквивалентности.

Типы сведений:

1. `poly` (По Карпу; Рассмотрено выше)
2. `many -- one` (По Карпу; Решив задачу B можем решить сразу много задач A)
3. `many -- many` (По Куку)
   Алгоритм A имеет оракула L, если может за время $\mathcal{O}(1)$ решить задачу L. (Обозначается A^L)
   $L_1 \le L_2$ если $\exists A^{L_2} \in \texttt{poly} : A^{L_2}(s) = L_1(s)$

Lm 44.3.1. $L_1 \le L_2, L_2 \in \texttt{NP/P} \Rightarrow L_1 \in \texttt{NP/P}$

Доказательство. Очевидно: сведем L_1 к L_2 и решим. ■

44.4. Классы сложности задач (продолжение)

1. `NP-hard`
   $A \in \texttt{NP-hard} \Leftrightarrow \forall L \in \texttt{NP} : L \le A$
2. `NP-complete`
   $\texttt{NP-complete} = \texttt{NP} \cap \texttt{NP-hard}$

Lm 44.4.1. $L_1 \le L_2, L_1 \in \texttt{NP-complete} \Rightarrow L_2 \in \texttt{NP-hard}$

Доказательство. $\forall A \in \texttt{NP} : A \le L_1 \le L_2 \Rightarrow A \le L_2$ ■

Лекция по алгоритмам #45
Классы сложности

24 мая

45.1. Примеры задач из разных классов

Немного истории. Концепция NP-полноты развивалась в 1960х-1970х годах независимо в СССР и США (Эдмондс, Левин, Яблонский). В 1971 году Стивен Кук сформулировал гипотезу о P vs. NP.

45.1.1. SAT

Задача: Проверить выполнимость данной формулы.

Левин и Кук независимо доказали NP-полноту задачи SAT, которая стала универсальной для доказательства NP-полноты многих других задач.
У выполнимой формулы существует выполняющий набор. Он будет являтся сертификатом. Очевидно, существует полимиальный алгоритм, который подставляет значения набора и проверяет, выполнилась ли формула. Значит, $SAT \in NP$
Доказательство $SAT \in NP - hard$ не было дано (да и это не нужно). В общих словах, оно заключается на написании формулы на ячейках памяти и состоянии машины Тьюринга, при выолнении которых машина даст ответ 1.

45.1.2. CNFSAT

Задача: Проверить выполнимость формулы, заданной в КНФ.

$CNFSAT \in NP$ – аналогично SAT.
Рассмотрим дерево расщипления по данной формуле. По этому дереву с помощью добавления новых переменных можно построить формулу в КНФ (подробно это было на курсе ДМ). Размер этой формулы будет полимиален относительной исходной. Т.к. $SAT \in NP - hard$, а мы свели SAT к $CNFSAT$, то $CNFSAT \in NP - hard$.

45.1.3. 3-CNFSAT

Комментарий: Я не уверен, что это было на лекции.

Задача: Проверить выполнимость формулы, заданной в 3-КНФ.

Рассмотрим формулу в КНФ. Путь в некотором ее дизъюнкте k переменных.

1. Если $k < 3$, то продублируем некоторую переменную из дизъюнкта, чтобы получить 3 переменных в дизънкте.

2. Если $k > 3$. Назовем их $x_1, x_2, \dots, x_k$. Добавим новые переменные $z_1, z_2, \dots, z_{k-2}$. Рассмотрим формулу:

$$(x_1 \lor x_2 \lor z_1) \land (x_3 \lor \neg z_1 \lor z_2) \land \dots \land (x_i \lor \neg z_{i-2} \lor z_{i-1}) \land \dots \land (x_{k-1} \lor x_k \lor \neg z_{k-2})$$

Пусть в выолняющем наборе исходной формулы некоторое $x_i = 1$. Тогда, в новой формуле мы можем расставить выполняющие значения в z_i (сначала будем ставить от первой скобки до скобки с x_i, потом от последней; эти значения будут чередоваться, но это не важно, т.к. скобка с x_i все равно выполнится).
Обратно, если все $x_i = 0$, то новая формула не может выполниться. Чтобы выполнилась 1-я скобка, $z_i = 1$. Для выполнения 2-й скобки $z_2 = 1$. Аналогично получаем $z_i = 1$. Но тогда не выполнится последняя скобка.
Получаем, что новая формула равносильна исходной, а значит мы свели $CNFSAT$ к $3 - CNFSAT \Rightarrow 3 - CNFSAT \in NP - hard$

Отсюда можно доказать $k - CNFSAT \in NP - complete$ для любого $k > 2$.

45.1.4. Independent set

Задача: В данном неор. графе проверить наличие независимого множества размера k.

Для множества вершин за $O(poly)$ можно проверить его независимость $\Rightarrow IND \in NP$.

Сведем к IND задачу $3 - CNFSAT$. Пусть в формуле k дизъюнктов. Создадим для каждого дизъюнкта 3 вершины, соединим их ребрами. Также соединим ребрами все пары вершины, которые соответствуют переменной и ее отрицанию.
Если для данной формулы существует выполняющий набор, но существует независимое множество размера k. В каждом дизъюнкте возмем по 1 вершине, переменная которой равна 1. Т.к. у нас не будет взята вершины из пар переменная-ее отрицание, то мы получили независимое множество.
В обратную сторону, если мы нашли независимое множество размера k, то в нем есть по 1 вершине для каждого дизъюнкта. Переменная и ее отрицания не могут быть одновременно в этом множестве, а значит мы получили корректный выполняющий набор.

Вершинное покрытие сводится к независимому множеству, а значит тоже лежит в $NP - comlete$. Аналогично в $NP - comlete$ лежит задача о клике.

45.1.5. Halting problem

Задача: По данному алгоритму и входным данным проверить, завершится ли когда-нибудь его выполнение.

Эта задача лежит в классе $NP - hard$, но не лежит в NP.
Задача из NP сводится к $Haltingproblem$ записыванием алгоритма-перебора строк-сертификатов и запуска проверки, что этот сертификат подходит. Если данная программа завершится, то будет найдет сертификат.

45.1.6. Другие задачи

Задача изоморфизма графов является NP, но неизвестно, принадлежит ли она $NP - hard$. Также к NP относятся задача факторизации. Рюкзак является $NP - comlete$ задачей.

Лекция по алгоритмам #46
Теория чисел

23 мая

Def 46.0.1. $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p$ *поле остатков по модулю* p.

Def 46.0.2. $(\mathbb{Z}/m\mathbb{Z})^*$ *Группа по умножению* $\{a\colon (a,m) = 1 \cap 1 \leq a < m\}$.

Def 46.0.3. *Линейное диофантово уравнение* (a, b, c *даны, нужно найти* x, y).

$$ax + by + c = 0; x, y \in \mathbb{Z}$$

Деление $\frac{a}{b} = a \cdot b^{-1}$

46.1. Расширенный алгоритм Евклида

Чтобы решить само уравнение, нужно домножить результат алгоритма на $c/\gcd(a,b)$. Если без остатка не делится, то решения нет.

```
 1 int euclid(int a, int b, int &x, int &y) { ax + by = gcd(a, b)
 2   if (a == 0) {
 3     x = 0;
 4     y = 1;
 5     return b;
 6   }
 7   int x1, y1;
 8   int d = euclid(b%a, a, x1, y1);
 9   x = y1 - (b / a) * x1;
10   y = x1;
11   return d;
12 }
```

Lm 46.1.1. (Без доказательства) Все промежуточные и конечные x и y по модулю не больше $\max(|a|, |b|)$.

46.2. Обратные в кольце по модулю

a и m даны, хотим найти x.

$$a \cdot x = 1 \mod m$$
$$ax + my = 1 = \gcd(a, m)$$

Другой способ:

$$a^{p-1} = 1 \mod p \Rightarrow x = a^{p-2}$$
$$a^{\phi(m)} = 1 \mod m \Rightarrow x = a^{\phi(m)-1}$$

Замечание Функцию Эйлера считать долго!

$$\phi(n) = n \prod \frac{p_i - 1}{p_i};\ n = \prod p_i^{\alpha_i}$$

46.3. Возведение в степень за $\mathcal{O}(\log n)$

```
1 int pow(int x, int y) {
2   res = pow(x, n / 2);
3   res *= res;
4   return (n & 1) ? res * x : res;
5 }
```

46.4. Обратные для чисел от 1 до $k-1$ по модулю за $\mathcal{O}(k)$

Модуль m, считаем динамикой.

Теорема 46.4.1.

$$i^{-1} = -\lfloor\frac{m}{i}\rfloor \cdot (m \mod i)^{-1}$$

Доказательство.

$$m \bmod i = m - \lfloor\frac{m}{i}\rfloor \cdot i$$
$$m \bmod i = -\lfloor\frac{m}{i}\rfloor \cdot i$$
$$1 = -\lfloor\frac{m}{i}\rfloor \cdot i \cdot (m \mod i)^{-1}$$
$$i^{-1} = -\lfloor\frac{m}{i}\rfloor \cdot (m \mod i)^{-1}$$

■

46.5. RSA

Два типа шифрования:

Симметричная криптография Ключ позволяет зашифровать и расшифровать сообщение. Пример: xor.

Криптография с открытым ключем Боб хочет послать сообщение Алисе и шифрует его открытым ключем Алисы (e), который виден всем. Для расшифровки Алисе понадобится ее закрытый ключ (d), который знает только она. Функции для шифровки и расшифровки открыты.

RSA вкратце:

$$n = pq;\ p, q \in \mathbb{P}$$
$$\phi(n) = (p-1)(q-1)$$
$$e \cdot d = 1 \mod \phi(n)$$
$$e, d < \phi(n)$$

Выберем случайные p, q и e, а n, $\phi(n)$ и d вычислим по формулам. Тогда открытым ключем будет $\langle e, n\rangle$, а закрытым — $\langle d, n\rangle$.

Зашифруем и расшифруем:

$$m \Rightarrow m^e(n) \Rightarrow (m^e)^d(n) = m^{\phi(n)\cdot k+1}(n) = m(n)$$

Алгоритм надежен настолько, насколько сложна задача факторизации чисел. Числа умеют факторизовать так:

1. $\mathcal{O}(n)$ Тривиально.
2. $n^{1/4} \cdot gcd$ Эвиристика Полларда, будет в курсе.
3. $e^{\sqrt[3]{\ln n}}$

Обычно в RSA используют ключ длины 2048. Можно шифровать за $\Theta(k^3/w^2)$ — k операций “деление по модулю”. Можно деление реализовать за $\mathcal{O}(k \log k)$, получается $\mathcal{O}(k^3)$.

46.6. Первообразный корень

Def 46.6.1. F_p: $1, g, g^2, g^3, \ldots, g^{p-2}$
Такой корень, который пораждает всю мультипликативную группу.

Из алгебры знаем, что он существует.

Алгоритм для поиска:

```
1 return random[1, p-1]
```

Lm 46.6.2. $Pr \geq \frac{1}{\ln \ln p}$

Доказательство. Алгоритм “ткнет” в g^i, нам лишь нужно, чтобы $\gcd(i, p-1) = 1$.

$$\frac{\phi(p-1)}{p-1} \geq \frac{1}{\ln \ln p}$$

Покажим, что ϕ не бывает слишком маленьким. Оценим худший вариант: произведение простых чисел. $1/2 \cdot 2/3 \cdot 4/5 \cdot 6/7 \cdot \ldots$ Оценивается так: заменим произведение на сумму логарифмов и возьмем два первых члена ряда Тейлора. ■

Проверка корректности (следующие условия должны выполняться):

1. $g^{p-1} = 1$
2. $g^{(p-1)/q} \neq 1 \ \forall q$: q делит $(p-1)$

Другой подход: пусть у нас есть алгоритм, использующий первообразный корень, который за $A(n)$ умеет решать какую-то полезную задачу, а за $B(n)$ умеет проверять ответ на верность. Запустим его много раз и получим правильный ответ с хорошей вероятностью.

Если p маленькое, то можно подсчитать корень заранее и забить константу в алгоритм.

Лекция по алгоритмам #47
Теория чисел

10 июня

47.1. Китайская теорема об остатках

Из алгебры известно, что если $a_1, \ldots, a_k$ - попарно взаимно простые числа, $\forall i\ \alpha_i \geq 1$, то

$$\mathbb{Z}/(a_1 \cdot \ldots a_k) \simeq \mathbb{Z}/a_1 \oplus \ldots \oplus \mathbb{Z}/a_k)$$

Нам интересно взять не абы какие взаимно простые числа, а различные простые. Выражение можно переписать попроще так:

$$\forall x \in \mathbb{Z}: \quad x \bmod (p_1 \cdot \ldots \cdot p_k) \simeq (x \bmod p_1, \ldots, x \bmod p_k)$$

Или, если словами, остаток от деления любого числа x на произведение простых чисел однозначно определяется остатками от деления x на каждое из этих чисел.

Этот знаменательный факт даёт нам возможность производить вычисления, не влезая по уши в длинную арифметику. Например, если мы знаем, что результат вычислений не превосходит, скажем, 10^{1000}, достаточно взять 112 простых, больших 10^9 и считать остатки по каждому из простых по отдельности.

Научимся восстанавливать число по его остаткам на различные простые. Ясно, что достаточно научиться восстанавливать для двух модулей, то есть по остаткам от деления на m_1 и m_2 получать остаток от деления на $m_1 m_2$. Для большего числа модулей - берём первые два, схлопываем в один и так далее. Итак:

$$x \equiv a_1 \pmod{m_1}$$

$$x \equiv a_2 \pmod{m_2}$$

Мы хотим подобрать такие e_1 и e_2, что

$$e_1 \equiv 1 \pmod{m_1} \qquad e_1 \equiv 0 \pmod{m_2}$$

$$e_2 \equiv 0 \pmod{m_1} \qquad e_2 \equiv 1 \pmod{m_2}$$

Тогда $x \equiv a_1 e_1 + a_2 e_2 \pmod{m_1 m_2}$. Нам подойдут такие:

$$e_1 = m_2 \cdot (m_2^{-1} \pmod{m_1}) \qquad e_2 = m_1 \cdot (m_1^{-1} \pmod{m_2})$$

Так же советую почитать про алгоритм Гарнера.

Сравним времена работы длинной арифметики и арифметики на остатках. Пусть T - кол-во операций, n - длина промежуточных вычислений, m - длина результата.

1. Длинка: $T \cdot n \log n$
2. Остатки (нам понадобится порядка m различных модулей): $T \cdot m + m^2 \log n$

47.2. Решето Эратосфена

Решето за $\mathcal{O}(n \log \log n)$:

```
for(int i = 2; i < n; ++i)
        if (!is[i])
        for(int j = i * i; j < n; j += i) is[j] = true; //не простое
```

Решето за $\mathcal{O}(n)$. Пусть $d[x]$ - минимальный делитель x, больший 1. Тогда;
$\forall x: \ x = d[x] \cdot y$. Если x - не простое, то $y > 1$, и $d[y] \geq d[x]$.

```
for(int y = 2; y < n; ++y){
    if (d[y] == 0)
       d[y] = y, primes.push_back(y);
    for(int i = 0; p[i] <= d[y] && y*p[i] <= n; ++i)
        d[p[i]*y] = p[i];
}
```

Почему линия? Во внутреннем for выражение $p[i] * y$ каждый раз новое (любое число x однозначно представляется в виде $d[x] * y$), а всего их n.

47.3. Вычисление мультипликативных функций

Функция наз-ся мультипликативной, если:

$$gcd(a, b) = 1 \to f(ab) = f(a)f(b)$$

. Пусть нам надо вычислить $f(1), \ldots, f(n)$. Естественная динамика: пусть $x = \sqcap p_i^{\alpha_i}$, тогда:

$$f(x) = f(\frac{x}{p_1^{\alpha_1}}) \cdot f(p_1^{\alpha_1})$$

Особого внимания заслуживает функция эйлера:

$$\varphi(x) = x \cdot \sqcap \frac{p_i - 1}{p_i}$$

Пусть $y = x/d[x]$. Тогда $\varphi(x)$ удобно считать так:

$$\varphi[x] = \varphi[y] \cdot (d[y] == d[x]?d[x] : d[x] - 1)$$

47.4. Простые числа

Генерация. Поскольку простые числа распределены достаточно плотно, чтобы получить большое простое, берём случайное большое число и прибавляем к нему 1, пока оно не станет простым.

Проверка на простоту

1. В лоб - $\sqrt{x}$. Просто ищем делитель тупым перебором.

2. Если много запросов, можно предпосчитать простые числа за $\sqrt{x}$ и на запросы отвечать уже за $\frac{\sqrt{x}}{\log x}$, итерируясь только по простым.

3. Тест Ферма. Берём случайное a и проверяем, что $a^{x-1} \equiv 1 \pmod{x}$. Числа, удовлетворяющие условию, образуют подгруппу в группе обратимых элементов $x\mathbb{Z}$ (взаимно простые). Если это собственная подгруппа, её размер не более $\frac{\varphi(x)}{2}$ по теореме Лагранжа. Поэтому вероятность ошибиться не более $\frac{1}{2}$. Если же подгруппа совпадает со всей группой (но число не простое), то нам не повезло. Это значит, мы наткнулись на одно из чисел Кармайкла, для которых тест Ферма не работает (мы всё ещё можем наткнуться на число, не взаимно простое с x, но таких обычно мало).

   Можно доказать, что у любого числа Кармайкла хотя бы три делителя. Это даёт нам алгоритм проверки на простоту за $\sqrt[3]{x}$. Сначала проверяем, что нам не подсунули число Кармайкла (у него есть делитель $\leq \sqrt[3]{x}$), потом запускаем тест Ферма.

4. Тест Миллера-Рабина. Этот тест уже всегда работает (мы не доказывали). a нужно брать случайным, но по какой-то гипотезе можно взять одно из первых $\log^2 x$ или что-то вроде того.

$$x - 1 = 2^s \cdot t$$

   Алгоритм:

```
a=a^t
FOR i = 1...s
   if a^2=1 && |a|!=1
       NOT PRIME
   a=a^2
return a==1 ? PRIME : NOT PRIME
```

47.5. Факторизация

Задача: по числу n либо сказать, что оно простое, либо представить его в виде произведения двух множителей (каждый больше 1).

Уже умеем за $\sqrt{n}$ и $\sqrt{n}/\log n$ с предподсчётом. Оказывается, можно и быстрее.

Эвристика Полларда Пусть $f : [0 \ldots n-1] \to [0 \ldots n-1]$ - некоторая случайная функция (подходит $f(x) = (x^2+3) \bmod n$. Без док-ва).

Возьмём случайное a и рассмотрим цепочку: $\{x_i\} : \ a, f(a), f(f(a)), \ldots$. Она должна зациклиться примерно через $\sqrt{n}$ шагов. Вероятность того, что зациклится сильно раньше или позже очень низкая.

Пусть $n = pq \ p \leq q, p > 1$ и $y_i = x_i \bmod p$. Последовательность y_i должна зациклиться примерно через $\sqrt{p} \leq \sqrt[4]{n}$ шагов. Значит в некоторый момент случится так, что:

$$x_i \equiv x_{i+T} \pmod{p} \qquad x_i \not\equiv x_{i+T} \pmod{n}$$

Значит $gcd(x_{i+T} - x_i, n) \geq p$, и $gcd(x_{i+T} - x_i, n) \neq n$, т.е. gcd - некоторый нетривиальный делитель n. Это как раз то, что нам надо. Итак, код:

```
a = RANDOM
b = f(a)
while (true){
```

```
a = f(a)
b = f(f(b))
x = gcd(b - a, n)
if x != 1 return x
```

Асимптотика - $\mathcal{O}(\sqrt[4]{n} \cdot gcd)$. На практике даже приемлемо работает.

Лекция по алгоритмам #48
Окончание лекции по теории чисел

1 июня

48.1. 3 задачи про разные степени

1)(Напоминалка)Первообразный корень $g^{p-1} = 1$ vs $g^{\frac{p-1}{x}} = 1 (x > 1)$, второе человечество умеет решать за $fact(x) + \log^2 n$, факторизацию умеем за $\sqrt[4]{x} \cdot gcd(x)$

2)$x^a \equiv b(mod\ p)$ - Дискретное логарифмирование

3)(решается через 1 и 2) $x^a \equiv b(modp)$ - извлечение корня, где p - простое

48.2. Решения:

2) Baby-Step-Giant-Step

Ассимптотика: $\mathcal{O}(\sqrt{p})$

Примем $k = \lceil \sqrt{p} \rceil$

Вспомним старую добрую Теорему Ферма $a^{p-1} \equiv 1(modp) \Rightarrow 0 \leq x < p$ <- т.к. остатки циклятся

Пусть мн-во $A = \{a^0, a^-1, a^-2, ..., a^{-(k-1)}\}$ Считаем за $\mathcal{O}(\log p + k)$, сначала ищем $a^{-(k-1)}$, а потом за линию оставшиеся отрицательные степени.

$B = \{a^0, a^k, a^2k, ..., a^{k(k-1)}\}$ <- большие степени

Из большой и маленькой степени можно скомпоновать любую от 0 до p(т.к. поделив любую степень на k, получим остаток и частное, не превосходящие k).

Этим и воспользуемся. Положив все большие в Хеш-таблицу(Map), где ключом будет значение, а значением, спепень, в которую возвели.

```
1 unordered_map<int,int> B;
2 for (int z : A) {
3   if (B.count(z * b))
4     ans = B[z * b] + deg[z];
5 }
```

Тем самым мы решили задачу за $\mathcal{O}(\sqrt{n})$

3) Сведение к первым 2-м.

Ассимптотика: $\mathcal{O}(\sqrt{p})$

$x^a \equiv b(modp)$, вомним, что существует натуральный логарифм.

Заменим переменные $x = g^i$, $b = g^j$ (эту задачу мы уже научились решать в пункте 2))

$g^{i \cdot a} \equiv g^j$ (mod p), теперь задача поменялась и нужно найти i. Теоритически умеем делать это $\mathcal{O}(\log p)$

$i \cdot a \equiv j \cdot (p-1) \Rightarrow i = j \cdot a^-1$

Замечание: Ни факторизацию, ни дискретное логарифмирование человечество за $\mathcal{O}(\log_2^k n)$ решать не умеет.

Ассимптотика решения Факторизации:

1)$\frac{\sqrt{n}}{\log n}$

2)$\sqrt[4]{n}^{\frac{1}{4}} \cdot gcd$ - реализацию первых двух алгоритмов знаем

3)$exp(\sqrt{\log n})$ - алг. Крайчика

4)$exp(\sqrt[3]{\log n})$ - Элептические кривые, алг. Ленстры

Напоминалка: $exp(k \cdot \log \cdot \log) = \log^k$ $exp(k \cdot \log) = n^k$

Лекция по алгоритмам #49
Вероятностные алгоритмы

27 мая

49.1. Фильтр Блюма

Детерминированная структура данных, похожая на хэш-таблицу. Умеет добавлять элементы и проверять вхождение. Работает за настоящее $\mathcal{O}(1)$. Единственный минус заключается в том, что иногда он говорит “да” на запрос о несуществещем элементе.

Алгоритм: Есть m бит, n элементов, k *универсальных* хэш функции h.

```
void add(int x) {
  // h is a vector of hash functions, a may be a bitset
  for (int i = 0; i < h.size(); i++)
    a[h[i](x)] = 1;
}

bool get(int x) {
  retval = 1;
  for (int i = 0; i < h.size(); i++)
    retval &= a[h[i](x)];
  return retval;
}
```

Заполненность не более $\max\left(0, \frac{nk}{m}\right)$.

Как выбрать оптимальное k? Давайте посчитаем вероятность ошибки:

$$P[\text{error}] = \left(1 - (1 - \frac{1}{m})^{nk}\right)^k = \left(1 - e^{-\frac{nk}{m}}\right)^k = 2^{-k}\ (\text{if } k = \ln 2 \cdot \frac{m}{n}) = 0.6185^{m/n}$$

49.2. Вероятностные алгоритмы

RP Randomized polinomial. Если ответ нет, тогда говорим нет, если ответ да, то говорим да с вероятностью не менее $\frac{1}{2}$.

coRP Дополнение RP. Если ответ да, тогда говорим да, если ответ нет, то говорим нет с вероятностью не менее $\frac{1}{2}$.

ZPP Zero-error Probablistic Polynomial. Матожидание времени полиномиально, всегда дает верный ответ.

BPP Bounded-error Probablistic Polynomial. Полиномиален по времени, вероятность правильного ответа $\geq \frac{2}{3}$.

Теорема 49.2.1. $ZPP = RP \cap coRP$

Доказательство. Докажем, что $ZPP \in RP \cap coRP$. Запустим алгоритм $C \in ZPP$, дадим ему работать как минимум в два раза дольше матожидания. Тогда мы получим правильный ответ с вероятностью не менее $\frac{1}{2}$ (По неравенству Маркова: $Pr(X \geq a) \leq \frac{E[x]}{a}; a = E[x] \cdot 2$).

Докажем теперь в обратную сторону, это делается конструктивно:

```
1  while true:
2    if RP_algo(x) == 1:  return 1;
3    if coRP_algo(x) == 0: return 0;
```

$$x \in L; F(Time) = \frac{Time(RP_{algo})}{Pr(RP_{algo})}$$

$$E = 1 + p + p^2 + p^3 + \cdots = \frac{1}{1 - Pr[Error]}$$

■

49.3. Понижение ошибки и прочее

Lm 49.3.1. О понижении ошибки в BPP

Утвержение: можно за $\mathcal{O}(k)$ запусков уменьшить ошибку до обратной экспоненты. Нужно всего лишь запустить алгоритм k раз и выбрать более популярный ответ.

Доказательство. Закон больших чисел для распределения Бернулли.

$$\Pr = \sum_{a=k/2}^{k} C_k^a \cdot \left(\frac{1}{3}\right)^a \cdot \left(\frac{2}{3}\right)^{k-a} \leq \frac{2^k}{2} \cdot \left(\frac{2}{9}\right)^{k/2} = 1\frac{1}{2} \cdot \left(\frac{2}{9}\right)^{k/2} \tag{1}$$

■

Lm 49.3.2.

$$x \in L \Rightarrow \Pr[1] > \frac{1}{2} + \varepsilon$$
$$x \notin L \Rightarrow \Pr[0] < \frac{1}{2} + \varepsilon$$

Тоже по закону больших чисел для распределения Бернулли.

Некоторые вероятностные методы:

Метод Лас-Вегаса ткнем в случайный объект, скажем что этот ответ. Работает неожиданно часто.

Метод Монте-Карло хотим посчитать площадь объекта. Натыкаем много точек в квадрате, который точно включает объект. Тогда процент точек, попавших в объект, будет равен проценту, который его площадь занимает от площади квадрата.