Алгоритмы и структуры данных-1
Коллоквиум

Винер Даниил @danya_vin

Версия от 10 июня 2025 г.

Содержание

1 Способы задания графов

1.1 Матрица смежности

Матрица смежности — матрица графа $G = (V, E)$ размера $V \times V$, такая что на пересечении $i-$ой строки и $j-$ого столбца стоит 1, если есть ребро между вершинами i и j, и 0 — если иначе

Если нужно узнать, есть ли ребро между i и j, достаточно обратиться к $A[i][j]$ — $O(1)$

Затраты по памяти — $O(V^2)$, так как храним матрицу размера $V \times V$

1.2 Список смежности

Список смежности — массив, в котором каждой вершине графа соответствует список, состоящий из соседей этой вершины

Пусть V — это количество вершин, E — количество рёбер. Изначально создается V динамически расширяемых пустых векторов

При считывании ребра $A_i - B_i$ в вектор с номером A_i добавляется ребро B_i (если граф неориентированый, то еще в вектор с номером B_i добавляется ребро A_i)

Сложность построения — $O(E)$

Сложность нахождения всех соседей каждой вершины — $O(V + E)$, так как мы проходим по всем вершинам за V и просматриваем каждое ребро за E

1.3 Список ребер

Определение. Список ребер — структура данных, представляющая собой набор из пар $A_i - B_i$, где A_i, B_i — вершины графа

Порядок в списке ребер не важен **только в случае неориентированных** графов

Сложность нахождения всех соседей каждой вершины

Если граф неориентирован, то оптимально будет сделать его ориентированным (помимо ребра $A_i - B_i$ добавить ребро $B_i - A_i$)

- Для поиска всех соседей вершины (в неориентированном случае) надо перебрать все ребра и сравнить текущую вершину со второй вершиной ребра. $O(E)$
- Сортировка ребер — $O(E \log E)$
- Поиск первого соседа в списке ребер для одной вершины — $O(\log E)$
- Поиск первого соседа в списке ребер для всех — $O(V \log E)$

Так как обычно в графах $E \geqslant V$, то для поиска всех соседей всех вершин потребуется $O(E \log E)$

2 Обход в глубину

2.1 Обход в глубину

Описание алгоритма

Обход начинается с любой вершины графа. Из этой вершины мы переходим в одного из непосещенных соседей. Если все соседи посещены, то мы возвращаемся вдоль всего пройденного пути, пока не наткнемся на вершину, у которой есть непосещенный сосед. Алгоритм завершает работу, когда мы возвращаемся в исходную вершину и все ее соседи посещены

Отметим, что в общем случае, когда неизвестно связный граф или нет, нужно запустить обход в цикле по всем вершинам

Применение алгоритма

1. Поиск случайного пути в лабиринте
2. Решение задач, связанных с построением маршрута: в сети, на карте, в сервисах покупки билетов и так далее
3. Проврека на наличие циклов, топологическая сортировка

Асимптотика

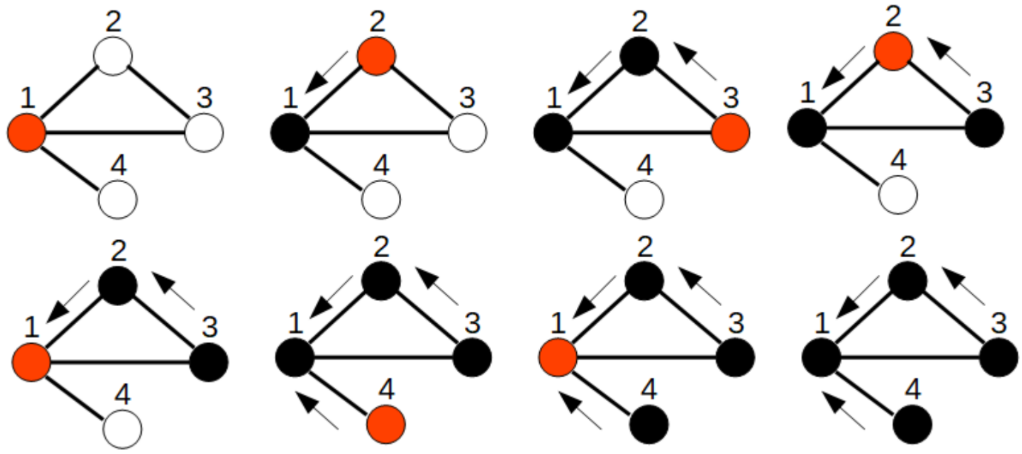
Сложность по памяти — $O(V)$, где V — количество вершин графа

Временная сложность зависит от представления графа: матрица смежности, список ребер или список смежности

Рассмотрим каждый вариант:

1. Список смежности — $O(V + E)$
   - Вершины посещаются (проверяются на посещенность) только один раз, что составляет $O(V)$
   - Каждое ребро проверяется ровно один раз, что составляет $O(E)$
2. Матрица смежности — $O(V^2)$
   - Вся матрица имеет размер $V \times V$
3. Список рёбер — $O(E \log E + E)$ при предварительной сортировке и бинарном поиске

Пример



2.2 Связность неориентированного графа

Связный граф — граф, в котором существует путь от любой вершины до любой другой вершины

Проверка неориентированного графа на связность

Запуск DFS от любой из вершин графа

- Запускаем DFS из любой вершины
- После DFS проверяем, посещены ли все вершины. Если да — граф связный, нет — несвязный
- Сложность по памяти будет зависеть от способа хранения графа
- Сложность по времени — $O(V + E)$

2.3 Сильная связность ориентированного графа

Определение. В ориентированном графе сильная связность требует, чтобы для любых двух вершин u и v существовал ориентированный путь $u \to v$ и $v \to u$

Для проверки нужно запустить 2 DFS

1. Запускаем DFS на исходном графе
   - Выбираем любую вершину v
   - Запускаем из неё DFS, помечая все достижимые вершины
   - После завершения проверяем: если посетили не все V вершин, то граф не сильно связен
2. Второй DFS в транспонированном графе
   - Строим транпонированный граф, то есть все ребра $u \to v$ превращаются в $v \to u$
   - Запускаем DFS из той же вершины v
   - Снова убеждаемся, что посетили все вершины

Если во время обоих обходов мы посетили все вершины, то граф является сильно связным

2.4 Поиск компонент связности в графе

Компонента связности графа — подмножество вершин и соединяющих их ребер, такое что есть путь из каждой вершины в каждую

Алгоритм поиска компоненты связности Можно раскрасить граф в компоненты связности. Каждой вершине ставится в соответствие номер компоненты связности, к которой относится вершина

Реализация происходит через DFS
Может потребоваться несколько запусков поиска в глубину. Поэтому поиск проводим через цикл, перебирающий все вершины

- Заводим массив длинной V для хранения цвета каждой вершины
- При входе в DFS красим вершину в текущий цвет (`color[v] = component`)
- Если очередная вершина не покрашена, то счетчик количества компонент связности увеличивается и запускается DFS для этой вершины со значением цвета равным текущему значению счетчика

Затраты по памяти составят $O(V)$ под массив цветов и стек рекурсии (или очередь при итеративном обходе), плюс хранение графа

Сложность по времени: $O(V + E)$ при хранении графа списком смежности, потому что каждый DFS обходит каждую вершину и каждое ребро не более одного раза, а цикл по всем вершинам добавляет лишь проверку цвета: $O(V + E)$

2.5 Поиск цикла в графе

Чтобы найти цикл в графе нужно раскрасить его в три цвета:

- **Белый** — вершина непосещенная
- **Серый** — DFS вошел в вершину, но не обработал всех соседей
- **Черный** — Все соседи вершины посещены и помечены черным

Если в графе есть обратные ребра, т.е. ребра ведущие в серую вершину, то в графе есть цикл

Алгоритм

- В каждый момент времени серым цветом будут помечены вершины, лежащие на пути **DFS** от стартовой вершины до текущей
- Если из текущей вершины u есть ребро в серую вершину v, то в графе есть цикл, т.к. существует путь от v до u (v лежит на пути от стартовой вершины до u) и путь от u до v, проходящий по одному ребру

Временная сложность: $O(V+E)$, поскольку каждая вершина и каждое ребро обрабатываются не более одного раза

Память: $O(V)$ под массив меток и стек рекурсии.

Восстановление цикла

Допустим, для u нашелся серый сосед v, тогда запоминаем номер v и выходим из рекурсивной функции, запоминая номера вершин, пока не дойдем до вершины, в которой нашелся цикл

2.6 Проверка графа на двудольность

Примечание. Граф — двудольный тогда и только тогда, когда все циклы в графе имеют четную длину

Алгоритм

- Выбираем произвольную вершину и красим ее в цвет *color*
- Всех соседей этой вершины красим в цвет $3 - color$
- Соседей соседей — в цвет *color* и т.д.
- Если в какой-то момент сосед вершины уже покрашен в тот же цвет, что и вершина, то алгоритм завершает работу, так как граф не двудольный, потому что есть циклы нечетной длины

Если граф не является связным, то нужно запустить **DFS** из каждой вершины каждой компоненты связности (как в п. 2.4)

2.7 Диаметр и центр дерева

Диаметр дерева — максимальная длина (в рёбрах) кратчайшего пути в дереве между любыми двумя вершинами

Центр дерева — вершины (одна или две) максимально удаленные от других вершин дерева

Примечание. Центр можно понимать так: это вершина дерева, такая что при подвешивании дерева за нее глубина дерева минимальна

Алгоритм поиска диаметра дерева

Требуется использовать два **DFS**

- Берем любую вершину дерева (пусть это a) и ищем самую удаленную от нее вершину b с помощью **DFS**
- Из вершины b запускаем **DFS** и ищем самую удаленную от нее вершину (пусть это c)
- Путь из b в c — диаметр дерева

3 Задача построения дерева кратчайших расстояний

3.1 Обход в ширину

Наивный алгоритм

Создаем массив, заполняем его бесконечностями. Для начальной вершины установим значение 0

Выполняем $v-1$ шаг. Нужно перебрать все вершины и выбрать те, которые находятся на расстоянии равном номеру шага, а также пометить все соседние вершины числом на 1 большим, чем номер текущего шага

- На нулевом шаге выбираем начальную вершину и помечаем ее соседей 1
- Затем выбираем вершины, находящиеся на расстоянии 1 и их непомечанных соседей помечаем 2 и т.д.

Сложность по времени — $O(V^2+E)$, так как мы сделаем V шагов, переберем V вершин, а также просмотрим все ребра

Сложность по памяти — $O(V)$ заводится под массив расстояний

Реализация через очередь

1. Инициализировать для всех u: $\text{dist}[u]=\infty$, $\text{visited}[u]=\text{false}$.
2. $\text{dist}[start]=0$, $\text{visited}[start]=\text{true}$, положить $start$ в очередь q.
3. Пока q не пуста:
   - $u=q.\text{pop}()$.
   - Для каждого соседа v из $\text{adj}[u]$:
     - Если $\neg\text{visited}[v]$:
       * $\text{visited}[v]=\text{true}$,
       * $\text{dist}[v]=\text{dist}[u]+1$,
       * $q.\text{push}(v)$.
       * Если v — искомая вершина, выйти из всех циклов.

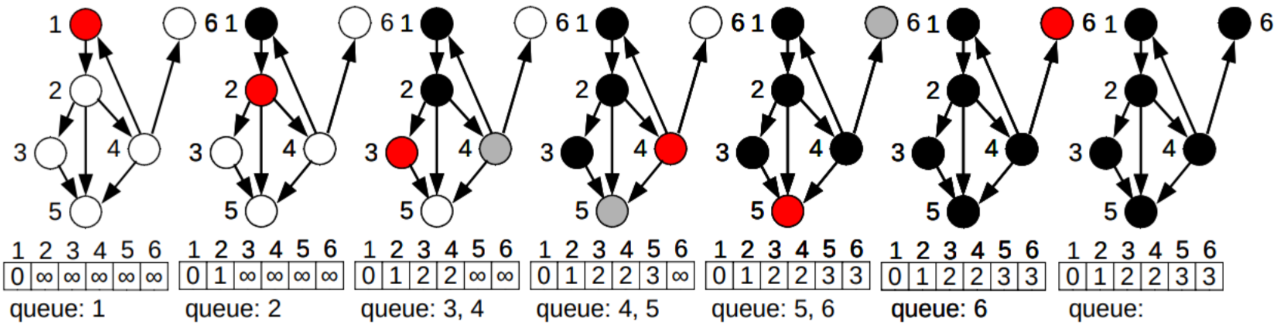
Сложности:

- Время: $O(V+E)$.
- Память: $O(V)$ под очередь, массивы `dist` и `visited`, плюс хранение графа.

Применение алгоритма

1. Для поиска кратчайшего пути в неявно заданных и невзвешенных графах
2. Для обнаружения кратчайших путей и минимальных покрывающих деревьев

Белый — еще не посещенные вершины, *черный* — уже посещенный, *красный* — обрабатывающиеся в данный момент (в начале очереди), *серый* — вершины, находящиеся в очереди



3.2 Алгоритм Дейкстры

Attention: веса ребер должны быть неотрицательными, иначе алгоритм не будет работать корректно

Случай для плотного графа

Создадим такие массивы

- `dist` — размером $v+1$. В нем для каждой вершины хранится текущая длина кратчайшего пути от начальной вершины s. (`d[s] = 0`, а все остальное — inf)

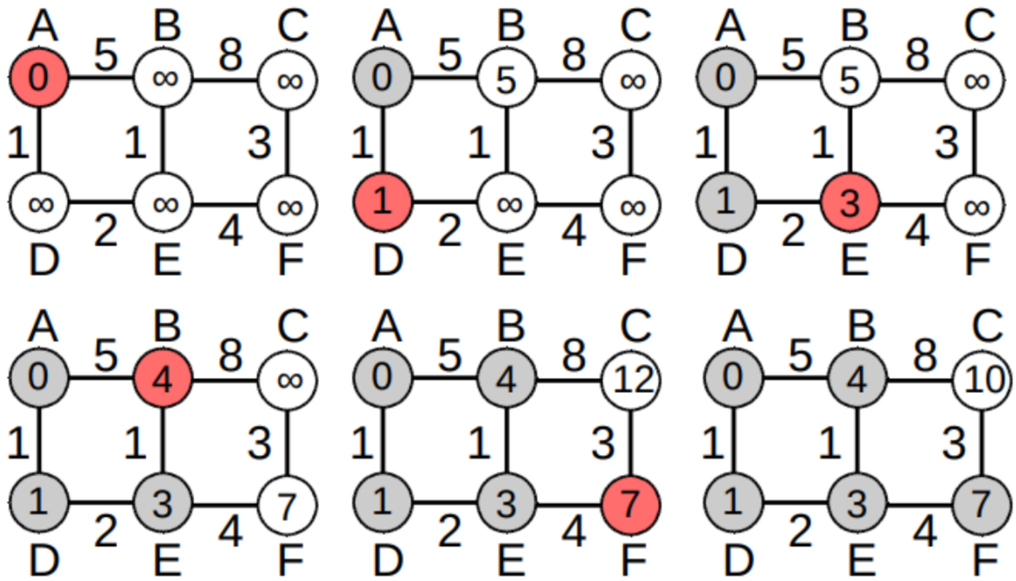
- `visited` — размером $v+1$. Хранит иформацию о том, обработана вершина или нет

Алгоритм состоит из $v-1$ шага. На каждом шаге делаем следующее
Для i=1...V:

1. Выбрать необработанную вершину v с минимальным `dist[v]`.

2. Пометить v как обработанную и выполнить релаксацию всех ребер $(v \to u)$:

$$\texttt{dist[u]} = \min(\texttt{dist[u]}, \texttt{dist[v]} + w(v,u)).$$



Красный — вершины, для которых будет произведена релаксация, *серый* — обработанные вершины

Сложность со списком смежности — $O(V^2+E)$, так как мы на каждом из $v-1$ шагов ищем минимум из V чисел для вершин и просматриваем каждое ребро один раз

Сложность с матрицей смежности — $O(V^2)$

Случай для разряженного графа

Reminder. Разряженным называется граф, у которого количество ребер значительно меньше, чем V^2

Алгоритм Дейкстры для разреженных графов работает следующим образом:

1. Инициализация:

   - Задаётся начальная вершина, от которой будут рассчитываться кратчайшие пути
   - Все расстояния до остальных вершин инициализируются как бесконечность, кроме начальной вершины, для которой расстояние равно нулю
   - Создаётся структура данных, например, `ordered set` или куча, для хранения и быстрого доступа к вершинам и их текущим кратчайшим расстояниям

2. Пока есть непосещённые вершины:

   - Выбирается вершина с минимальным текущим расстоянием (с использованием кучи или `ordered set`)

- Эта вершина помечается как посещённая
- Для каждой соседней вершины, смежной с текущей:
  - Рассчитывается новое потенциальное расстояние как сумма текущего расстояния до рассматриваемой вершины и веса ребра между текущей вершиной и соседней
  - Если новое расстояние меньше известного расстояния до соседней вершины:
    * Обновляется расстояние до соседней вершины
    * В `ordered set` или куче обновляется информация о расстоянии до этой вершины

3. Обновление структуры данных:

   - При обновлении расстояний до соседних вершин, в структуре данных (куче или `ordered set`) происходит операция удаления старого значения и добавления нового значения, что обеспечивает эффективность алгоритма
   - В случае использования кучи без изменения элементов (например, `priority queue`) просто добавляются новые значения, а устаревшие игнорируются

Процесс повторяется, пока не будут посещены все вершины или не будут определены кратчайшие пути до всех достижимых вершин

Сложность — $O(E \log V + V \log V)$. Узнаем минимум за $O(1)$ (удаление минимума тратит $O(\log V)$), изменяем элементы за $O(\log V)$. Для большинства графов сложность будет просто $O(E \log V)$, так как каждое из E ребер может привести к уменьшению пути и изменению значения в set

В итоге массив расстояний содержит кратчайшие пути от начальной вершины до всех остальных вершин

3.3 Алгоритм Форда-Беллмана

Создадим такой массив

- `dist` — размером $v + 1$. Массив кратчайших расстояний. (`d[s] = 0`, а все остальное — inf)

Алгоритм состоит из $v - 1$ шага. Пусть есть ребро (u, v) и его вес w. На каждом шаге перебираем все ребра и проводим релаксацию по этому ребру, то есть

```
if (dist[v] > dist[u] + w && dist[u] != inf){
    dist[v] = dist[u] + w
}
```

Применение алгоритма

Алгоритм хорош в поиске кратчайших путей от одной вершины до всех остальных на разряженных графах, если в графе есть отрицательные ребра

Временная сложность: $O(V \times E)$

Слоность по памяти: $O(V + E)$

- $O(V)$ под массив `dist`;
- $O(E)$ под список рёбер.

3.4 Алгоритм Флойда–Уоршалла

Работаем с матрицей размера $V \times V$, где

$$\text{dist}[i][j] = \begin{cases} w(i,j), & \text{если } (i \to j) \text{ есть ребро,} \\ 0, & i = j, \\ \infty, & \text{иначе.} \end{cases}$$

Основной тройной цикл:

```
for k in 1..V:
    for i in 1..V:
        for j in 1..V:
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
```

Обнаружение отрицательного цикла:
Если после выполнения алгоритма $\text{dist}[i][i] < 0$ для некоторого i, в графе есть отрицательный цикл.

Применение

- Поиск кратчайших путей «каждый→каждый»

Сложности:

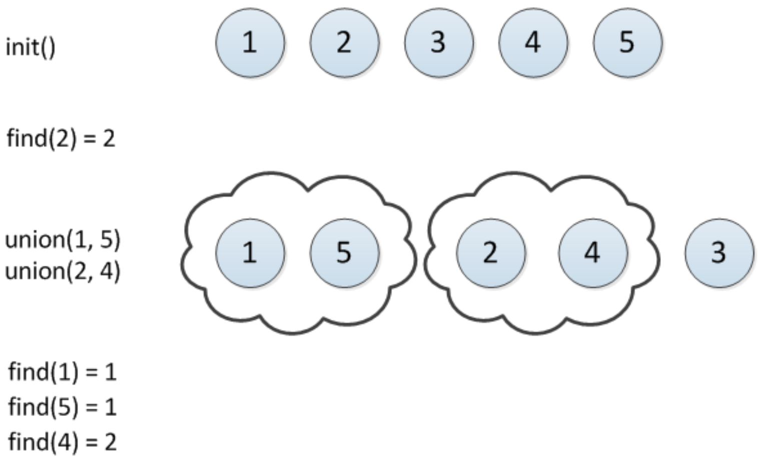
- Время: $O(V^3)$.
- Память: $O(V^2)$.

4 Задача union — find

4.1 Задача union — find

Пусть у нас есть N элементов, занумерованных от 1 до N. Изначально каждый элемент находится в своем отдельном множестве (также занумерованных от 1 до N). Нам необходимо поддерживать такие операции:

- `find(x)` — найти номер множетсва, в котором лежит x
- `union(x, y)` — объединить множества, содержащие x и y
- Можно добавить, но необязательно:
  - `make(x)` — создает новое множество, содержащее x



4.2 Наивная реализация

1. Создаем массив с индексами от 1 до N
   - Индекс означает номер элемента
   - Значение по индексу — номер множества, к которому относится элемент
2. Операция `find(x)`
   - Нужно вернуть содержимое ячейки с номером x
   - **Сложность** — $O(1)$
3. Операция `union(x, y)`
   - Узнаем номера множеств, содеражащих эти элементы (пусть это a и b соответственно)
   - Проходим по всему массиву и заменяем значения b на a
   - **Сложность одной операции** — $O(N)$
   - **Сложность объединения всех множеств** — $O(N^2)$

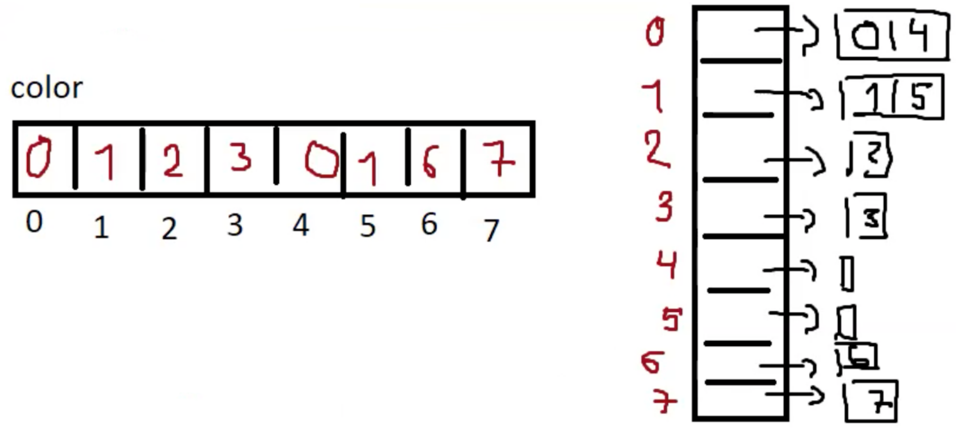
4.3 Реализация с использованием линейных списков

Идея этой реализации заключается в том, что мы заводим второй масиив, на индексах которого стоят номера множеств, а по индексу хранится список элементов соответствующего множества

Тогда при вызове `union(x, y)` мы делаем так: `max(x, y) += min(x, y)`

Сложность

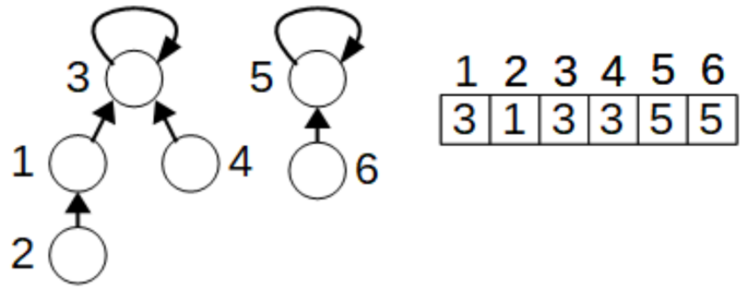
Так как мы храним второй масиив, то мы *теряем в памяти*, однако алгоритм будет работать *быстрее*, за $O(N \log N)$, так как делаем $O(\log N)$ шагов на кажом шаге $O(N)$ для объединения множеств



4.4 Система непересекающихся множеств

Идея заключается в хранении каждого множества в виде дерева

Мы имеем один единственный массив предков `p`. Индексы в нем — номера элементов, а по индексу хранится номер предка



В данном примере, петли означают, что это корень дерева, а в `p` на позицию с номером корня записывается значение самого корня

Вместо этого, в `p` на позицию корня можно поставить -1, суть не изменится

Тогда операции `find(x)` и `union(x, y)` можно реализовать так:

```
int find(x){
    while (p[v] != -1){
        v = p[v]
    }
    return v
}

void union(x, y){
    p[find(x)] = find(y)
}
```

Сложность — $O(N^2)$

4.4.1 Сжатие путей

Идея заключается в том, что когда мы найдём искомого предка p множества (с помощью `find(x)`), то запомним, что у вершины x и всех пройденных по пути вершин — именно этот предок p. Проще всего это сделать, перенаправив их parent[] на эту вершину p

Теперь в массиве предков для каждой вершины там может храниться не непосредственный предок, а предок предка, предок предка предка, и т.д

Сложность — $O(\log N)$

4.4.2 Объединение деревьев

Идея: нужно подвешивать дерево с большей глубиной к дереву с меньшей глубиной

Для каждого дерева храним его глубину в массиве

Сложность — $O(\log N)$

4.5 Алгоритм Краскала

- Отсортируем все ребра по возрастанию
- Каждая вершина находится в своем отдельном множестве
- В остовное дерево берем те ребра, которые соединяют разные множества вершин
- При добавлении ребра происходит объединение множеств

Алгоритм используется для построения минимального остовного дерева в графе

Реализовать представленный алгоритм проще всего с помощью СНМ. Необходимо отсортировать ребра по неубыванию по их весам. Далее мы каждую вершину можем поместить в свое собственное дерево, то есть, создаем некоторое множество подграфов. Дальше итерируемся по всем ребрам в отсортированном порядке и смотрим, принадлежат ли инцидентные вершины текущего ребра разным подграфам с помощью функции find() или нет, если оба конца лежат в разных компонентах, то объединяем два разных подграфа в один с помощью функции union().

Итоговая сложность составит $O(E\log V + V + E) = O(E\log V)$, где $O(E\log V)$ - сортировка, $O(V)$ — создание отдельных множеств, $O(1)$ — find() и union()

5 Дерево отрезков

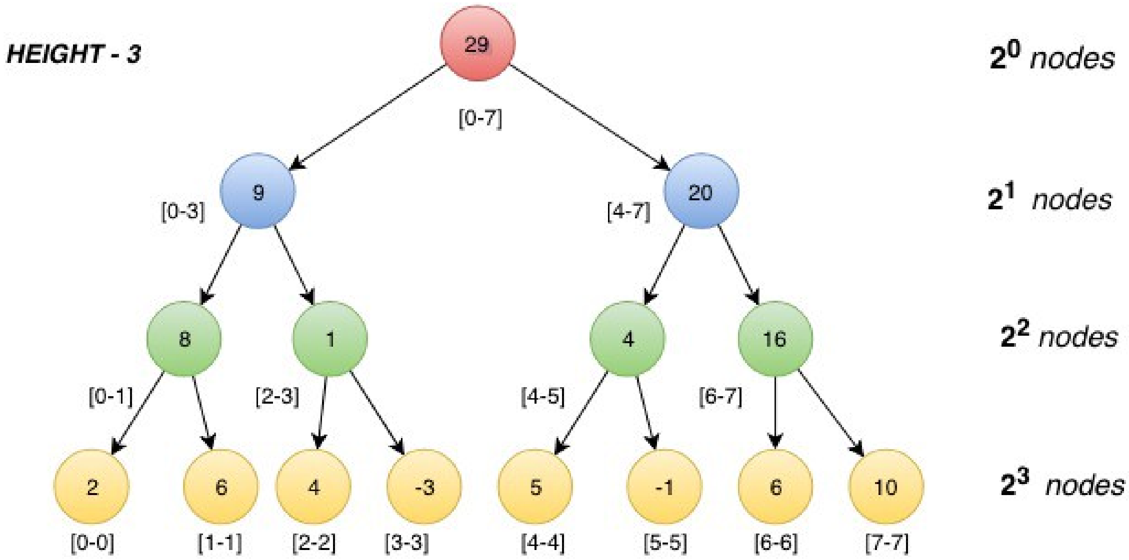
5.1 Дерево отрезков

Дан массив a из n целых чисел, и требуется отвечать на запросы двух типов:

1. Изменить значение в ячейке (т. е. реагировать на присвоение `a[k] = x`)
2. Вывести сумму элементов a_i на отрезке с l по r

Несколько изменим массив

- Посчитаем сумму всего массива и где-нибудь запишем
- Разделим его пополам, посчитаем сумму на половинах и тоже где-нибудь запишем
- Каждую половину разделим пополам ещё раз, и т.д., пока не придём к отрезкам длины 1



Корень этого дерева соответствует отрезку $[0, n)$, а каждая вершина (не считая листьев) имеет ровно двух сыновей, которые тоже соответствуют каким-то отрезкам

5.2 Операции на отрезке

Здесь представлены операции на отрезке, реализованные с помощью указателей. Эта реализация не самая эффективная, но самая простая, решающая большинство задач. Подробнее о других реализациях тут и тут

5.2.1 Построение

Строить дерево отрезков можно рекурсивным конструктором, который создает детей, пока не доходит до листьев

Если изначально массив не нулевой, то можно параллельно с проведением ссылок насчитывать суммы

Сложность — $O(n)$

```
Segtree(int lb, int rb) : lb(lb), rb(rb) {
    if (lb + 1 == rb)
        s = a[lb];
    else {
        int t = (lb + rb) / 2;
        l = new Segtree(lb, t);
        r = new Segtree(t, rb);
        s = l->s + r->s;
    }
}
```

5.2.2 Изменение

Для запроса прибавления будем рекурсивно спускаться вниз, пока не дойдем до листа, соответствующего элементу k, и на всех промежуточных вершинах прибавим x:

```
void add(int k, int x) {
    s += x;
    if (l){
        if (k < l->rb)
            l->add(k, x);
        else
            r->add(k, x);
    }
}
```

Сложность — $O(\log n)$

5.2.3 Сумма

Нужно делать разбор случаев, как отрезок запроса пересекается с отрезком вершины:

1. Если лежит полностью в отрезке запроса, вывести сумму
2. Если не пересекается с отрезком запроса, вывести ноль
3. else: рекурсивно запускаемся от детей

```
int sum(int lq, int rq) {
    if (lb >= lq && rb <= rq)
        return s;
    if (max(lb, lq) >= min(rb, rq))
        return 0;
    return l->sum(lq, rq) + r->sum(lq, rq);
}
```

Сложность — $O(\log n)$. На каждом уровне дерева отрезков, наша рекурсивная функция могла посетить максимум четыре отрезка; тогда, учитывая оценку $O(\log n)$ для высоты дерева, мы получаем асимптотику времени работы алгоритма

5.3 Применение дерева отрезков

Дерево отрезков может применяться в таких задачах, как

- поиск суммы на подотрезке
- поиск минимума/максимума на отрезкее
- массовые изменения массивов (например, добавление элемента ко всем элементам сразу)

6 Дерево поиска

Бинарное дерево поиска — дерево, для которого выполняются следующие свойства:

- У каждой вершины не более двух детей
- Все вершины обладают *ключами*, на которых определена операция сравнения (например, целые числа или строки)
- У всех вершин *левого* поддерева вершины v ключи *не больше*, чем ключ v
- У всех вершин *правого* поддерева вершины v ключи *больше*, чем ключ v
- Оба поддерева — левое и правое — являются двоичными деревьями поиска

В *небинарных* (~~нетрадиционных~~) деревьях количество детей может быть больше двух, и при этом в «более левых» поддеревьях ключи должны быть меньше, чем «более правых»

Для работы с деревьями поиска нужно создать структуру

```
struct Node:
  T key                  // key of the node
  Node left              // pointer to the left child
  Node right             // pointer to the right child
  Node parent            // pointer to the parent
```

6.1 Поиск элемента

Нужна функция, принимающая корень дерева и искомый ключ

- Для каждого узла сравниваем значение его ключа с искомым ключом
- Если ключи одинаковы, то функция возвращает текущий узел
- В противном случае функция вызывается рекурсивно для левого или правого поддерева

```
Node search(x : Node, k : T):
   if x == null or k == x.key
      return x
   if k < x.key
      return search(x.left, k)
   else
      return search(x.right, k)
```

Сложность в худшем случае — $O(h)$ (h — высота дерева), так как узлы, которые посещает функция образуют нисходящее дерево. Такое возможно, когда дерево является «бамбуком»

Сложность при оптимизации — $O(\log N)$. Если изменить способ хранения дерева, например сразу при проходе до какого-то ключа записать его как ключ ко всем вершинам в пути, то сложность снизится

6.2 Вставка элемента

Почти то же самое, что поиск элемента, но теперь при обнаружении у элемента отсутствия ребенка нужно подвесить на него вставляемый элемент

```
Node insert(x : Node, z : T):         // x - root of the subtree, z - key to be inserted
   if x == null
      return Node(z)                  // attach a Node with key = z
   else if z < x.key
      x.left = insert(x.left, z)
   else if z > x.key
      x.right = insert(x.right, z)
   return x
```

6.3 Удаление элемента

Рассмотрим три случая при рекурсивной реализации

1. Удаляемый элемент находится в *левом* поддереве текущего поддерева
   - тогда нужно рекурсивно удалить элемент из нужного поддерева
2. Удаляемый элемент находится в *правом* поддереве
   - тогда нужно рекурсивно удалить элемент из нужного поддерева
3. Удаляемый элемент находится в *корне*, то два случая:
   - имеет два дочерних узла
     - нужно заменить его минимальным элементом из правого поддерева и рекурсивно удалить этот минимальный элемент из правого поддерева
   - имеет один дочерний узел
     - нужно заменить удаляемый элемент потомком

```
Node delete(root : Node, z : T): // root of subtree, key to delete
  if root == null
    return root
  if z < root.key
    root.left = delete(root.left, z)
  else if z > root.key
    root.right = delete(root.right, z)
  else if root.left != null and root.right != null
    root.key = minimum(root.right).key
    root.right = delete(root.right, root.key)
  else
    if root.left != null
      root = root.left
    else if root.right != null
      root = root.right
    else
      root = null
  return root
```

7 LCA (TBA)