

Алгоритмы и структуры данных. Коллоквиум

*Основано на лекциях и тренировках
по алгоритмам Михаила Густокашина*

Содержание

1	Способы задания графов	2
1.1	Матрица смежности	2
1.2	Список смежности	2
1.3	Список ребер	2
2	Обход в глубину	2
2.1	Обход в глубину	2
2.2	Связность	3
2.3	Поиск компонент связности в графе	4
2.4	Поиск цикла в графе	4
2.5	Проверка графа на двудольность	4
2.6	Диаметр и центр дерева	5
3	Задача построения дерева кратчайших расстояний	5
3.1	Обход в ширину	5
3.2	Алгоритм Дейкстры	6
3.3	Алгоритм Форда-Беллмана	8
3.4	Алгоритм Левита	8
3.5	Алгоритм Флойда	9
4	Задача union — find	9
4.1	Задача union — find	9
4.2	Наивная реализация	10
4.3	Реализация с использованием линейных списков	10
4.4	Система непересекающихся множеств	10
4.4.1	Сжатие путей	11
4.4.2	Объединение деревьев	11
4.5	Алгоритм Краскала	11
5	Дерево поиска	11
5.1	Поиск элемента	12
5.2	Вставка элемента	12
5.3	Удаление элемента	12
6	Дерево отрезков	13
6.1	Дерево отрезков	13
6.2	Операции на отрезке	14
6.2.1	Построение	14
6.2.2	Изменение	14
6.2.3	Сумма	14
6.3	Применение дерева отрезков	15
7	Декартово дерево	15
7.1	По явному ключу	15
7.2	По неявному ключу	16
7.3	Операции	16
7.3.1	Merge	16
7.3.2	Split	17
7.4	Применение декартового дерева	17
7.4.1	Вставка	17
7.4.2	Сумма на отрезке	17
7.4.3	Переворот	18

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

Сложность построения — $O(V^2)$

1.2 Список смежности

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

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 неориентированный, 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

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 $B_i - A_i$)

Сортировка ребер — $O(E \log E)$; поиск первого соседа в списке ребер для одной вершины — $O(\log E)$, для двух — $O(V \log E)$

Так как обычно в графах $E \geq V$, 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

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. Поиск циклов, сортировки

Асимптотика

Сложность зависит от представления графа: матрица смежности, список ребер или список смежности

Рассмотрим каждый вариант:

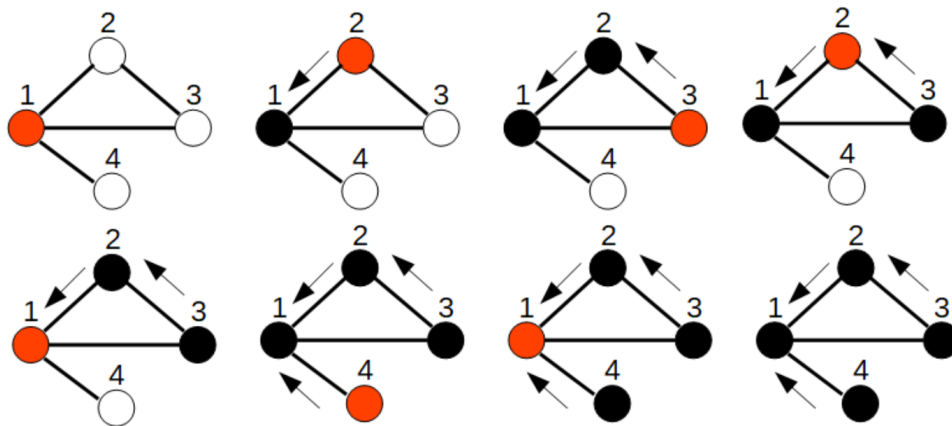
1. Список смежности — $O(V + E)$

- 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 $V \times V$

Пример



Псевдокод

```
void dfs(int v){
    used[v] = 1;
    for (auto to : gr[v]){
        if (!used[to]){
            dfs(to);
        }
    }
}
```

2.2 Связность

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

Проверка графа на связность

1. Запуск DFS от каждой из вершин графа

- Для каждого из обходов необходимо проверить, что посещены все вершины
- Сложность составит $O(V \times E)$

2. Запуск DFS от любой вершины

- Если от вершины 1 в неориентированном графе можно попасть в v и в u , то существует путь от v до u

- Так как есть путь из 1 в v , то в неориентированном графе существует и обратный путь из v в 1, а оттуда можно попасть в u
- Сложность такого подхода зависит от сложности DFS — 2.1

2.3 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 длиной V для хранения цвета каждой вершины
- При входе в DFS красим вершину в текущий цвет
- Если очередная вершина не покрашена, то счетчик количества компонент связности увеличивается и запускается DFS для этой вершины со значением цвета равным текущему значению счетчика

2.4 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

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

2.5 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. 2.3)

2.6 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

```
std::vector<node> find_diameter() {
    std::vector<node> ans, path;
    std::vector<int> used(gr_.size(), 0);

    std::function<void(node v)> dfs = [&] (node v) {
        used[v] = true;
        path.emplace_back(v);
        for (const auto& to : gr_[v]) {
            if (!used[to]) {
                dfs(to);
            }
        }
        if (ans.size() < path.size()) {
            ans = path;
        }
        path.pop_back();
    };
    dfs(0);
    used.assign(gr_.size(), 0);
    dfs(ans.back());
    return ans;
}
```

3 Задача построения дерева кратчайших расстояний

3.1 Обход в ширину

Наивный алгоритм

Создаем массив, заполняем его бесконечностью ($v + 1$, например). Для начальной вершины установим значение 0

Выполняем $v - 1$ шагов. 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 — $O(V^2 + E)$, так как мы сделаем V шагов, переберем V вершин, а также просмотрим все ребра

Реализация через очередь

В начале очереди находятся вершины текущего шага, а в конце — следующего

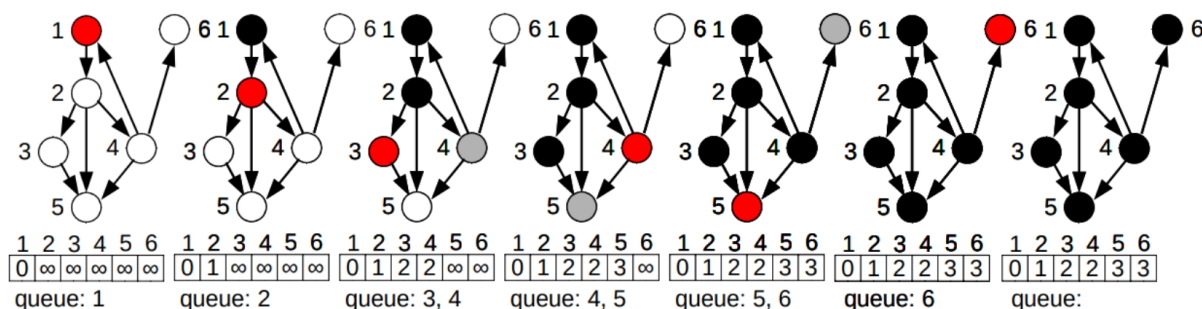
- Помещаем начальную вершину в очередь, а в массив расстояний ставим 0 для нее
- Берем первый (верхний) элемент из очереди и просматриваем соседей
- Если сосед не посещался, то добавляем его в очередь, помечаем вершину как посещенную
- Если сосед является пунктом назначения, алгоритм завершает работу

Время работы — $O(V + E)$

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* — обрабатываемые в данный момент (в начале очереди), *серый* — вершины, находящиеся в очереди



3.2 Алгоритм Дейкстры

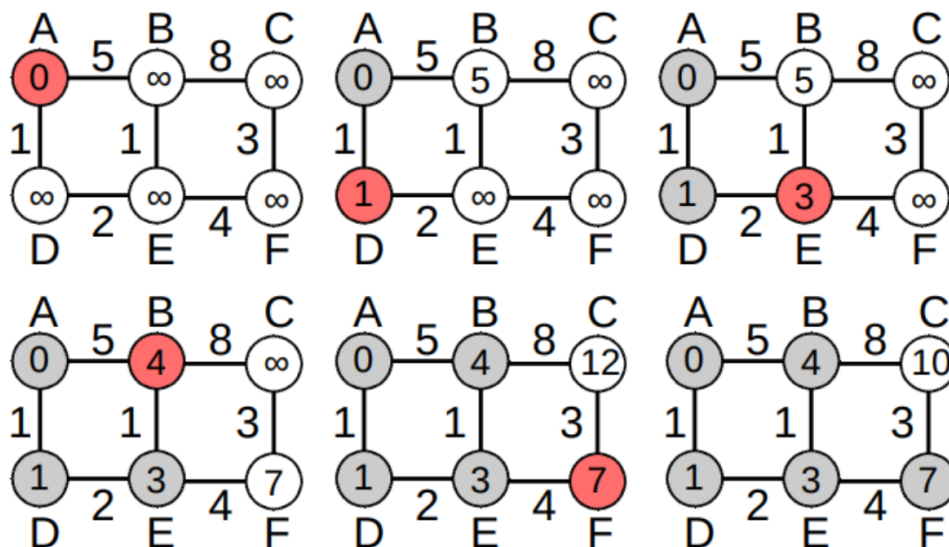
Случай для плотного графа

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 — ∞)
- **visited** — размером $v + 1$. Хранит информацию о том, обработана вершина или нет

Алгоритм состоит из $v - 1$ шага. На каждом шаге делаем следующее

1. Выбираем необработанную вершину v с минимальным расстоянием от начальной вершины. Помечаем v как обработанную
2. Производим релаксацию: рассматриваем все вершины, соседние с v (вес ребра $(u, v) = w$) и для соседней вершины u пытаемся улучшить значение **dist[u]**, т.е. $\text{dist}[u] = \min(\text{dist}[u], \text{dist}[v] + w)$



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 разреженного графа

Reminder. Разреженным 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. ($\text{d}[s] = 0$, 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[v] != inf){
    dist[v] = dist[u] + w
}
```

Применение алгоритма

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

Сложность — $O(V \times E)$

3.4 Алгоритм Левита

Пусть d_i — текущая длина кратчайшего пути до вершины i . Изначально, все элементы d , кроме s -го равны бесконечности, $\text{d}[s] = 0$

Разделим вершины на три множества:

- M_0 — вершины, расстояние до которых уже вычислено (возможно, не окончательно),
- M_1 — вершины, расстояние до которых вычисляется. Это множество в свою очередь делится на две очереди:
 1. M'_1 — основная очередь,
 2. M''_1 — срочная очередь;
- M_2 — вершины, расстояние до которых еще не вычислено.

Изначально все вершины, кроме s помещаются в множество M_2 . Вершина s помещается в множество M_1 (в любую из очередей)

Работа алгоритма: выбирается вершина u из M_1 . Если очередь M''_1 не пуста, то вершина берется из нее, иначе из M'_1 . Для каждого ребра (u, v) и весом w возможны три случая:

- $v \in M_2$, то v переводится в конец очереди M'_1 . И производится релаксация ребра (u, v) :
 $\text{d}[v] = \text{d}[u] + w$,
- $v \in M_1$, то происходит релаксация ребра (u, v) ,
- $v \in M_0$. Если при этом $\text{d}[v] > \text{d}[u] + w$, то происходит релаксация ребра (u, v) и вершина v помещается в M''_1 ; иначе ничего не делаем

В конце шага помещаем вершину u в множество M_0

Алгоритм заканчивает работу, когда множество M_1 становится пустым

Сложность — $O(V^2 E)$, так как при обработке каждой вершины приходится обрабатывать $n - 1$ ребро

3.5 Алгоритм Флойда

Работаем с матрицей смежности, которую будем превращать в матрицу кратчайших расстояний

Алгоритм пройдет V шагов. Перед k -м шагом на позиции `dist[from][to]` стоит длина кратчайшего пути из `from` в `to`, проходящего только через промежуточные вершины с номерами, меньше k

На первом шаге матрица смежности равна матрице `dist`, а отсутствующие ребра помечены как `inf`

При переходе от $k - 1$ шага к k нужно пересчитать матрицу расстояний, т.е. добавить в нее кратчайшие пути, проходящие через k . При пересчете пути из `from` в `to` возможны случаи:

1. Кратчайший путь из `from` в `to` не проходит через k , поэтому мы его не меняем
2. Кратчайший путь из `from` в `to` проходит через k : делим этот путь на две части — от `from` до k и от k до `to`. Тогда длина нового пути равна `dist[from][k] + dist[k][to]`

Применение алгоритма

Алгоритм применяется в задаче поиска кратчайших путей от каждой вершины до каждой. Его использование проще в написании, чем алгоритм Дейкстры

Сложность — $O(V^3)$, т.е. перебор всевозможных `from` и `to`

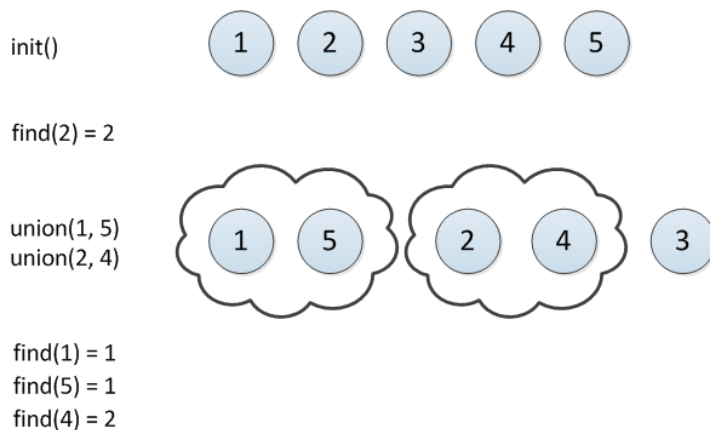
Требуется памяти — V^3 , однако, если хранить две матрицы, тогда требуемая память сократится до V^2

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 множества, 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, содержащих 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 — $O(N^2)$

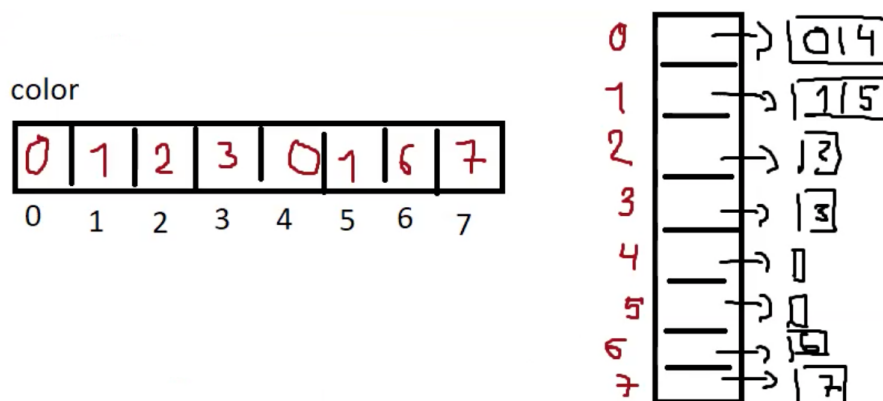
4.3 Реализация с использованием линейных списков

Идея этой реализации заключается в том, что мы заводим второй массив, 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

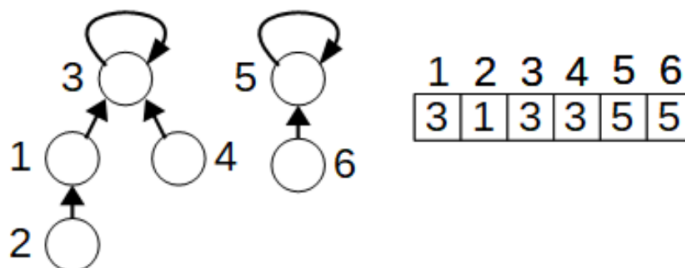
Так как мы храним второй массив, то мы *теряем в памяти*, однако алгоритм будет работать *быстрее*, за $O(N \log N)$, так как делаем $O(\log N)$ шагов на каждом шаге $O(N)$ для объединения множеств



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

5 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
```

5.1 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

5.2 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
```

5.3 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

```

6 Дерево отрезков

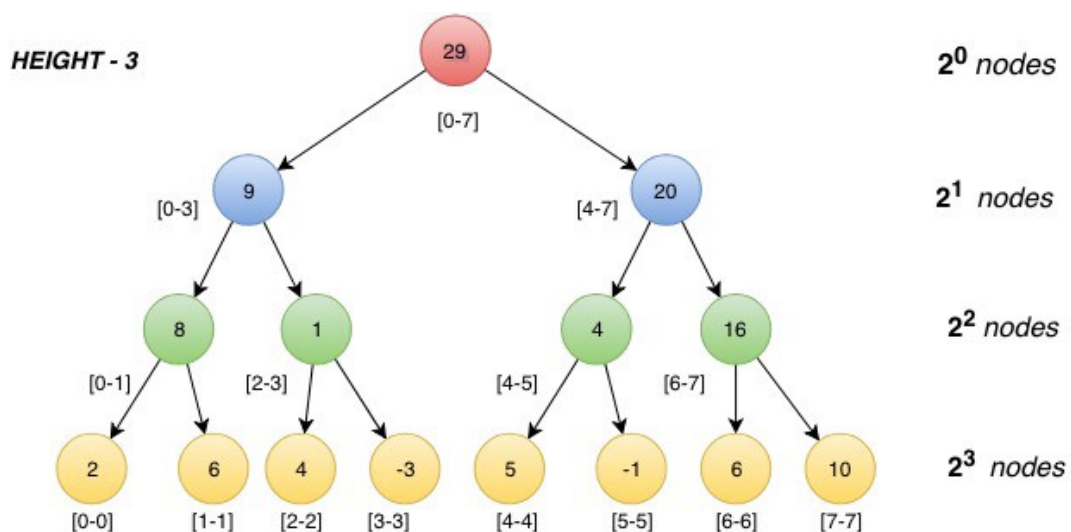
6.1 Дерево отрезков

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

6.2 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](#)

6.2.1 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;
    }
}
```

6.2.2 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 — $O(\log n)$

6.2.3 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

6.3 Применение дерева отрезков

Дерево отрезков может применяться в таких задачах, как

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

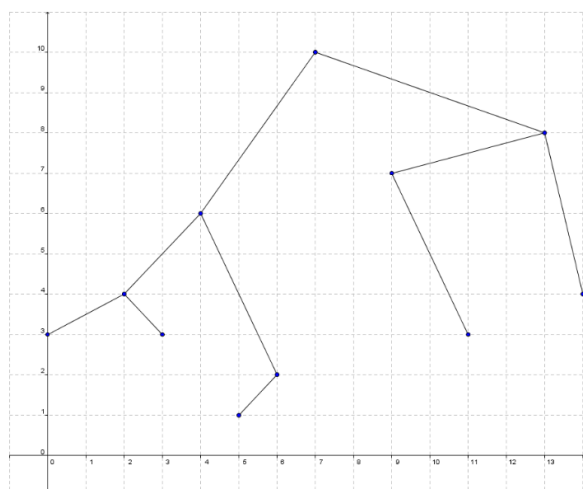
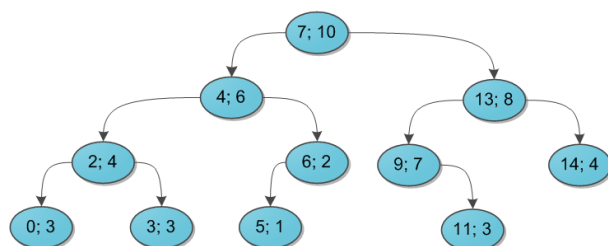
7 Декартово дерево

7.1 По явному ключу

Имеется бинарное дерево. Каждый его узел (обозначим как ключ x) дополняем случайным числом y . По ключам x образуется бинарное дерево поиска, а структура, полученная по y , обладает свойством двоичной кучи максимумов

Еще способ:

- Нанесём на плоскость набор из n точек. Их x назовем *ключом*, а y *приоритетом*
- Выберем самую верхнюю точку (с наибольшим y , а если таких несколько — любую) и назовём её *корнем*
- От всех вершин, лежащих слева (с меньшим x) от корня, рекурсивно запустим этот же процесс. Если слева была хоть одна вершина, то присоединим корень левой части в качестве левого сына текущего корня
- Аналогично, запустимся от правой части и добавим корню правого сына



Одно и то же дерево в обычном представлении и в виде декартового дерева

При добавлении второго ключа y представление дерева становится однозначным

Например, правым сыном корня будет самая верхняя вершина, находящаяся справа от корня, а левым — самая верхняя вершина слева от корня

Сложность — $O(\log N)$, так как координаты y выбираются случайно, то дерево получается сбалансированным и его высота равна $O(\log N)$

7.2 По неявному ключу

Работа с ДД по неявному пригодится в случае, если есть такие операции:

1. выводить сумму на произвольном отрезке
2. «переворачивать» произвольный отрезок, то есть переставлять элементы с l по r в обратном порядке, не меняя остальные

Так как во второй операции невозможно быстро поддерживать ключи актуальными, то делаем следующее:

- Убираем ключи
- Вместе с каждой вершиной храним размер ее поддерева
- Ключ (позицию элемента) восстанавливаем как число элементов слева от него
- Размер деревьев поддерживаем с помощью вспомогательной функции, вызываемой после каждого изменения дерева:

```
int size(Node *v) { return v ? v->size : 0; }

void upd(Node *v) { v->size = 1 + size(v->l) + size(v->r); }
```

Сложность — $O(\log N)$

7.3 Операции

Создадим структуру `Node`, в которой будем хранить ключ, приоритет и указатели на левого и правого сына

Сначала описывается реализация по явному ключу, после — по неявному

7.3.1 Merge

Принимает два дерева (два корня, L и R), про которые известно, что в левом все вершины имеют меньший ключ, чем все в правом. Их нужно объединить в одно дерево так, чтобы по ключам это было всё ещё дерево, а по приоритетам — куча

Сначала выберем, какая вершина будет корнем. Возьмем ту, у которой приоритет больше

Пусть это будет левый корень. Тогда левый сын корня итогового дерева должен быть левым сыном L . С правым сыном сложнее: возможно, его нужно смержить с R . Поэтому рекурсивно сделаем `merge(l->r, r)` и запишем результат в качестве правого сына

```
Node* merge (Node *l, Node *r) {
    if (!l) return r;
    if (!r) return l;
    if (l->prior > r->prior) {
        l->r = merge(l->r, r);
        return l;
    }
    else {
        r->l = merge(l, r->l);
        return r;
    }
}
```

По неявному ключу операция не изменится, так как нигде не использует ключи

7.3.2 Split

Принимает дерево и ключ x , по которому его нужно разделить на два: L должно иметь все ключи не больше x , а R должно иметь все ключи больше x

В этой функции мы сначала решим, в каком из деревьев должен быть корень, а потом рекурсивно разделим его правую или левую половину и присоединим, куда надо:

```
typedef pair<Node*, Node*> Pair;
```

```
Pair split (Node *p, int x) {
    if (!p) return {0, 0};
    if (p->key <= x) {
        Pair q = split(p->r, x);
        p->r = q.first;
        return {p, q.second};
    }
    else {
        Pair q = split(p->l, x);
        p->l = q.second;
        return {q.first, p};
    }
}
```

По неявному ключу операция немного изменяется. Теперь нужно использовать позицию корня вместо ключа. `Split` теперь выполняет такую команду, как «вырежи первые k элементов»

```
pair<Node*, Node*> split(Node *p, int k) {
    if (!p) return {0, 0};
    if (size(p->l) + 1 <= k) {
        auto [l, r] = split(p->r, k - size(p->l) - 1);
        p->r = l;
        upd(p);
        return {p, r};
    }
    else {
        auto [l, r] = split(p->l, k);
        p->l = r;
        upd(p);
        return {l, p};
    }
}
```

7.4 Применение декартового дерева

Операции `Merge` и `Split` позволяют реализовать следующие операции

7.4.1 Вставка

Например, чтобы добавить число x в дерево, мы можем разрезать его по x через `split`, создать новую вершину с одним числом x , и склеить через `merge` три получившихся дерева

7.4.2 Сумма на отрезке

- При `merge` и `split` надо будет поддерживать эту сумму актуальной
- Можно написать вспомогательную функцию `upd`, которую будем вызывать при обновлении детей вершины
- В `merge` и `split` теперь можно просто вызывать `upd` перед тем, как вернуть вершину
- При запросе суммы нужно вырезать нужный отрезок и запросить эту сумму

7.4.3 Переворот

Добавим флаг `rev` в каждую вершину, означающий, что подотрезок был перевернут

Когда мы встретим такую вершину, мы поменяем местами ссылки на её детей, а им самим передадим эту метку

Тогда функция `reverse` будет такой: вырезаем нужный отрезок, меняем флаг