Лекция

1 Persistent Data Structures

Для понимание этой части очень желательно знакомство с деревьями отрезков.

1.1 Введение

Для начал поймем, что такое `Persistent`. Пусть у нас есть массив a_0 и мы его меняем, пишим $a_0[i] := x$, старый массив не меняется создается новый массив a_1. Если мы пишем теперь $a_0[j] := y$, создается новый массив a_2, где в позиции i нет x-а.

Получается дерево версий. Старые версии структур живут вечно, мы их никогда не меняем. Мы только создаем новые структуры, новые версии.

Сейчас мы научимся делать любую структуру данных `Persistent`-ной. При этом память и время станут больше в $O(\log n)$ раз. Для бинарных деревьев увеличится только память.

1.2 Массив = Дерево отрезков, Дерево - это просто

Струтуры данных в основном состоят из массивов и бинарных деревьев.

Для начала представим массив в виде дерева отрезков. Дерево мы будем хранить не в другом массиве, а как произвольное бинарное дерево со ссылками на левое и правое поддеревья. Обращение к одному элементу массива (и чтение, и запись) теперь работают не за $O(1)$, а за $O(\log N)$.

Вернемся к деревьям. Как дерево сделать *Persistent*-ным? Рассмотрим следующий код, надеюсь, он полностью ответит на поставленный вопрос:

```
tree Add( tree t, int x )
{
  if (t == null)
    return new tree(null, null, x);
  if (x < t.x)
    return new tree(Add(t.left, x), t.right, t.x);
  else
    return new tree(t.left, Add(t.right, x), t.x);
}
```

Обратите внимание, старое дерево мы ни разу не поменяли. Подведем итог:

С этого момента весь мир для нас — деревья, а деревья мы умеем делать `Persistent`*. Каждая версия структуры данных — корень дерева. Мы храним массив версий, массив корней. Построенная нами конструкция — это несколько деревьев. Вместе они образуют ациклический граф. Одно дерево однозначно задается корнем.*

1.3 Garbage Collection

Если нам нужны все версии, все их нужно хранить. После k запросов, памяти будет использовано $O(k \log k)$.

Представим другую ситуацию: только 3-4 версии все еще нужны, память из под остальных можно очистить. В общем случае, если нужных версий в каждый момент времени $O(1)$, память можно сократить до $O(k)$ в каждый момент времени.

Способ 1:

Будем хранить число ссылок на каждую вершину. Если какая-то вершина является корнем, число ссылок на нее тоже нужно увеличить на 1.

Если какая-то версия нам уже не нужна, уменьшаем число ссылок на корень этой версии, если число ссылок стало нулем, эта вершина нам можно не нужна. Удалим ее, осовободим память. Удалять нужно рекурсивно, удаление вершины уменьшает число ссылок на другие вершины.

Способ 2:

Будем иногда (когда памяти осталось совсем мало и уже пора что-то делать) делать так: берем все живые корни и обходим их деревья, помечая, что все эти вершины живые. После этого очищаем память из-под всех вершин, не помеченных, как живые.

1.4 Демонстрация мощи слова Persistent. Решим задачку.

Многие задачи имеют `Offline` решение гораздо более простое, чем `Online`. `Persistent` структуры позволят нам многие `Offline` решения обобщить на `Online` случай.

Сделаем это на примере задачи *Даны n точек на плоскости, запрос = посчитать число точек в прямоугольнике.* Стороны прямоугольника, конечно, параллельны осям координат.

`Offline` решение этой задачи — одномерное дерево отрезков со сканирующей прямой. Пройдемся сканирующей прямой по точкам. Все деревья отрезков сделаем `Persistent` и сохраним в памяти. Произошло чудо, теперь, когда

нам приходит очередной прямоугольник, оба дерева отрезков, которые нужны для ответа на запрос уже посчитаны, к ним можно обратиться.

Добавлять новые точки мы так и не научились, но, если набор точек фиксирован, на запросы мы умеем отвечать `Online`.

1.5 Заключение

1. Многие структуры данных состоят из массивов и деревьев.
2. Дерево естественным образом делается персистентным (без потери времени, памяти в $\log N$ раз больше).
3. Массив — это тоже дерево. Дерево отрезков.

2 Перебор!

Для понимания этой части обязательно знакомство с рекурсией.

2.1 Читерские методы

2.1.1 Предподсчет

Представьте такую задачу: вам дан бинарный файл. Вы знаете, что это программа, которая для n от 1 до 500 умеет искать число кактусов из n вершин с точностью до изоморфизма. Эту программу написали участники контеста. Вы не знаете, правильно ли программа работает, какие ошибки в ней сделаны, у вас нет исходного кода программы.

Ваша задача — написать свою программу, работающую так же, как и данная вам на всех тестах. Решение этой задача — запустить данную вам программу на всех тестах от 1 до 500 и отправить предподсчитанный массив.

Это первое читерство. Имя его — **предподсчет**. Не забывайте про него.

2.1.2 Ленивое DP = перебор + запоминание

Чтобы написать динамику нужно написать перебор и добавить в него запоминание. Можно воспринимать это, как ленивую динамику, можно как оптимизацию к перебору. Часто, если не получается написать простую динамику по профилю, можно написать рекурсивный перебор и добавить в него отсечение “динамика по профилю”.

Для тех, кто плохо знаком с ленивой динамикой: рассмотрим задачу *найти число путей из s в t в ациклическом графе*. Будем для каждой

вершины v считать $f(v)$ — число путей из v в t. $f(v) = \sum_{x = \text{дитё } v} f(x)$.
Запускаем dfs из вершины s и рекурсивно считаем функцию f для всех вершин графа. Решение работает за $O(E)$. Чем это решение отличается от перебора всех путей из s в t? Ответ:

```
if (used[v])
  return f[v];
used[v] = 1
```

Благодаря этому `if`-у перебор превращается в динамику.

2.1.3 Жадность и перебор

Любую жадность можно улучшить до перебора. Жадность работает на некотором графе состояний. При этом в каждый момент времени выбирается максимальное по некоторой функции ребро. Можно вместо выбора максимума сделать сортировку и запустить перебор по всем ребрам начиная с максимального. Работать будет не хуже. А на маленьких тестах даже лучше.

`Пример`: решаем задачу о рюкзаке (набрать в рюкзак ограниченного веса вещи max суммарной стоимости) жадностью "брать каждый раз вещь с max удельной стоимсотью", чтобы превратить жадность в перебор, будем сортировать вещи по этому критерию и реализуем идею рекурсивно.

2.1.4 Ленивость перебора

Не перебирайте слишком много. Перебирайте значение объектов только, когда эти значения вам понадобились (для сравнения в `if`-е, например). Действуйте максимально лениво.

2.2 Постановка общей задачи

Давайте разберемся, что же такое перебор, какую задачу он решает, и чему будет посвящена эта часть лекции. "Перебором" мы будем решать 4 задачи на графе состояний.

- Проверка достижимости "хорошей" вершины
- Поиск невзвешенного кратчайшего пути
- Поиск взвешенного пути min веса
- Поиск взвешенного пути max веса

Последняя задача сводится к предыдущей домножением весов на -1, поэтому будем считать, что задач всего 3.

Поговорим о нашем графе. Граф, на котором, мы решаем описанные 3 задачи задан неявно. Он всегда настолько большой, что сохранить в памяти его не получается. Иначе мы бы просто написали соответственно `dfs`, `bfs` и `Ford-Bellman` или `Dijkstra`.

Многие из предложенных решений будут, как и положено "переборам", рекурсивными, но не все. Поговорим о рекурсии. Рекурсия порождает дерево рекурсии. Некоторые состояния этого дерева хочется склеить, так как это одна и та же вершина графа состояний. Рекурсивный же перебор — попытка написать алгоритмы `dfs`, `bfs` и `Ford-Bellman` рекурсивно. В отличии от `dfs`-а, `bfs` и `FB` написать рекурсивно не просто.

2.3 Меморизация (запоминание, хэширование состояний)

2.3.1 dfs

Для `dfs`-а это хэштаблица. Собственно мы получаем ленивую динамику.

2.3.2 bfs

Для `bfs`-а, если мы его пишем нерекурсивно все понятно — состояние добавляется в очередь только, если его еще нет в хэштаблице. Если же `bfs` реализуется через рекурсию, можно проверять, улучшилось расстояние до вершины, или нет. Углублять рекурсию имеет смысл только если расстояние до вершины улучшилось. Время работы такого алгоритма в худшем случае $O(VE)$ (т.к. для каждой вершины будет не более V улучшений).

2.3.3 FB

Рекурсивная реализается ни чем не отличается от "bfs"-а (см. выше)

2.4 Метод ветвей и границ для dfs-а

Рассмотрим теперь так называемый метод ветвей и границ на примере двух задач — поиск Гамильтонова пути в графе и поиск максимального по длине пути в невзвешенном графе.

Суть метода ветвей и границ в том, что не все ветви рекурсии нужно обходить, некоторые можно отсекать. Поводом для отсечения ветвей являются границы (оценки на то, насколько это поддерево интересно или неинтересно).

Для решения задачи прежде всего определим состояние: множество уже посещенных вершин и последняя вершина в пути. Состояний всего $O(2^n n)$, ребер $O(2^n n^2)$. У нас появился новый граф, граф состояний. Используем меморизацию (запоминание). В обоих задачах для каждого состояния нам важно только были мы уже в таком, или еще нет. Итого наше переборное решение работает за $O(2^n n^2)$.

Теперь новое, список отсечений:

1. **Отсечение по времени:** добавить его — никогда не лишнее.
2. **Отсечение по ответу:** это самое главное отсечение. $CurrentValue + F(State) \geqslant Answer$, $F(State)$ = количество достижимых вершин из текущего конца пути. Если мы ищем Гамильтонов путь, $Answer$ изначально равен $V - 1$, иначе 0.
3. **Жадность, сортировка ребер:** давайте перебирать ребра не в произвольном, а в каком-то "хорошем" порядке. Один из вариантов такого порядка — сортировка по $F(State)$. Для задачи про Гамильтонов цикл хорошо известна более лучшая жадность — порядок возрастания степени вершин. Интересный факт: задача про обход конем шахматной доски $N \times N$ для $N \leqslant 300$ решается без перебора, жадно. Для этого достаточно начать из случайной клетки доски, а жадность улучшить такой эвристикой: если степени равны, пытаемся максимально прижаться к краю доски.
4. **Перебор только k лучших ребер:** это первая из рассмотренных нами оптимизация (отсечение по времени не в счет), которая может привести к вердикту `Wrong Answer`. Тем не менее мощь оптимизации велика. Мы только пока не знаем, какое k лучше взять.
5. **Iterative Increasing of k:** будем перебирать k во внешнем цикле `for (k=1;;k++)`

Это попытка дать общую схему, которую легко обобщить на похожие задачи. Конкретно про Гамильтонов цикл можно добаыить следующее: помогает поиск мостов за $O(E)$ на каждом шаге рекурсии. Про путь max длины добавлю: для получения впечетляющих результатов достаточно первых двух из перечисленных оптимизаций.

Важно: Как работает отсечение по ответу? Нужно иметь функцию оценки $F(State)$: если $CurrentValue + F(State) \geqslant Answer$, можно оборвать рекурсию. Кроме этого нужно собственно иметь хорошее приближение ответа — $Answer$. Поиск в глубину сперва работает как жадность. Доходит до самого низа, получается оценка на ответ — текущий $Answer$.

2.5 dfs может быть не хуже bfs-а

Зададимся вопросом: мы ищем кратчайший путь, как его можно найти `dfs`-м, и чем `dfs` может оказаться лучше нерекурсивного `bfs`-а с очередью? Главное преимущество `dfs`-а — отсечение по ответу. Недостаток — то, что в каждую вершину мы вынуждены заходить несколько раз (каждый раз при этом расстояние до вершины улучшается).

Решение проблемы = `Iterative Deepening`. Давайте дадим `dfs`-у ограничение по глубине `MAX_DEP`. И будем перебирать эту константу внешним циклом `for (MAX_DEP=1;; MAX_DEP++)`. Это может ухудшить отсечение по ответу (иногда отсечение по ответу хорошо работает, только если мы позволяем `dfs`-у доходить до глубоких вершин графа). Мы получили новый алгоритм: `dfs + Iterative Deepening`.

Объясним, почему новый алгоритм по времени работы не хуже `bfs`-а. Казалось бы, мы должны каждый раз, для каждого нового `MAX_DEP` проделывать ту же работу, что и на предыдущей итерации, а `bfs` все делает только один раз. Заметим, что рост числа состояний экспоненциален от глубины. Т.е. `bfs` работает за $Num(Ans)$, а `dfs + Iterative Deepening` примерно за $Num(Ans)+Num(Ans)/2+Num(Ans)/4+\cdots$, что асимптотически не превосходит $Num(Ans)$.

2.6 bfs = dfs = dfs + Iterative Deepening

`dfs + Iterative Deepening` лучше `bfs`-а тем, что в нем есть отсечение по ответу. Давайте в `bfs` добавим такое же отсечение. И в `dfs + Iterative Deepening`, и в `bfs`, и в `dfs` отсечение по ответу работает по-разному, т.к. вершины перебираются в разном порядке, чтобы это ничем не портило первые 2 алгоритма, давайте сперва для улучшения оценки ответа запустим `dfs` на 0.5 секунд, а потом уже собственно алгоритм.

2.7 Улучшенный bfs, мажорирующий все другие методы

Давайте подведем итог всем отсечениям, которые мы узнали и добавим еще одно, последнее. Получится отличный алгоритм.

1. Сперва запускаем `dfs` без `Iterative Deepening` на 0.5 секунд. Используем все оптимизации для `dfs`-а, которые мы проходили выше.
2. Теперь пишем `bfs`. Нерекурсивный. Ограничиваем `bfs`-у размер очереди.
3. Внутри `bfs`-а не забываем использовать отсечение по ответу. Это почти самое важное.
4. В очереди в каждый момент времени оставляем только лучшие состояния (жадность). Можно использовать кучу (очередь с приоритетами), можно, как только очередь переполнится в 2 раза, сортировать все состояние и оставлять нужное число лучших.
5. `Iterative Deepening` по `max` размеру очереди. Размер очереди каждый раз удваивается.
6. Отсечение по времени. Оно нам поможет :-)

Замечание: 5-й оптимайз почти мажорирует 1-й, если размер очереди начинать перебирать с единицы.

2.8 AB-отсечение

Про него я сегодня рассказывать ничего не буду :-)

3 Метод откатов

Для понимание этой части очень желательно знание СНМ (Системы Непересекающихся Множеств).

Сейчас мы решим несколько задач. Задачи будут усложняться. Все задачи имеют почти одно и то же условие *Дан граф, он как-то меняется. После каждого изменения нужно говорить, сколько компонент связности в графе.*

3.1 Задача 1: ребра добавляются

СНМ в явном виде. Решение работает в `Online`.

3.2 Задача 2: ребра удаляются

Сперва удалим все ребра, а потом будем их добавлять в обратном порядке. Решение работает только в `Offline`.

3.3 Задача 3: ребра сперва добавляются, а потом удаляются

Разбиваем задачу на 2 части, решаем их отдельно. Решение работает только в `Offline`.

3.4 Задача 4: удалять можно только ребро добавленное последним

Есть 2 решения.

Решение 1: `Persistent` :-)

Решение 2: Нам нужна только одная версия, текущая.

Память меняется. Мы меняем какие-то ячейки памяти, но мы помним, какие. Можно все изменения значений в памяти хранить в стэке (храним ячейку и старое значение) и откатывать по необходимости состояние памяти.

Асимптотика СНМ при этом портится до $O(\log N)$, т.к. нам важно, сколько 1 запрос работает в худшем случае в реальном времени, без аморатизации.

Давайте подчеркнем главную идею: удаление ребра мы не делаем, мы читерим, откатываемся к предыдущему состоянию, в котором уже все посчитано.

Решение полностью `Online`.

3.5 Задача 5: удалять можно произвольное существующее ребро

Собственно об этом я и хотел рассказать. Решение предыдущей задачи можно назвать стэком. Решением этой задачи будет дерево отрезков. Идея решения не изменится — важно не делать `Del`, а откатываться назад.

Каждому запросу `Add` соответствует парный ему `Del`. И наоборот, каждому `Del` соответствует парный ему `Add`.

Построим на запросах дерево отрезков. Пускай, мы хотим знать число компонент связности после всех запросов с номерами $[L..R]$. Какие-то `Add`-ы уже точно можно сделать, какие-то точно известно, что отменятся

парным `Del`-ом. А про какие-то `Add`-ы мы пока не можем сказать ничего определенного. Храним все такие "неопределенные" `Add`-ы.

При рекурсивном спуске по дереву отрезков будем некоторые "неопределенные" `Add`-ы делать определенными. Эффекта два — мы больше не тащим их вглубь рекурсии, и нужно как-то поменять граф.

После рекурсивного спуска в конце концов следует рекурсивный подъем. При подъеме нужно отменить все изменения, которые мы сделали при спуске. Тут есть, как и в прошлой задаче, 2 пути — `Persistent` и хранить стэк изменений памяти.

Асимптотика решения ухудшилась уже до $O(k \log^2 k)$. Тем не менее такому времени для настолько сложной задачи можно порадоваться. Заметьте, решение работает только в `Offline`.

4 Разбор как продолжение лекции

Данная лекция задумывалась таким образом, что на разбор задач останется какой-то теоретический материал. Например, на разборе будут рассказаны быстрые решения NP-задач "число подклик в графе" и "минимальное вершинное покрытие произвольного графа".