ЛКШ, ЛКШ.2018.Август B'
B', конспект лекции

Собрано 9 августа 2018 г. в 16:34

Содержание

Тема #1: Разбор задач практики

8 августа

1.1. Редакционное расстояние

Будем делать квадратную динамику $n \times m$. $dp[i][j]$ — ответ на задачу для префикса до i-го элемента первой строки и префикса до j-го элемента второй строки (всё не включительно).
База $dp[0][0] = 0$ (ну и еще можно обнести границы, если очень не хочется ставить ифы).
Переход. Смотрим на префиксы длин i и j. Первый надо преобразовать во второй.
Мы можем сделать так, чтобы последние символы этих строк были равны, а дальше сведемся к уже решенной задаче.
У нас есть три варианта действий:

1. Добавить в конец первого префикса $b[j-1]$. Свелись к $dp[i][j-1]$
2. Удалить символ из конца первого префикса. Свелись к $dp[i-1][j]$
3. Поменять последний символ первого префикса на $b[j-1]$. Свелись к $dp[i-1][j-1]$

За эти три варианта мы «платили» единицу за изменение. Надо не забыть ее прибавить.
Но возможен вариант, когда $a[i-1] = b[j-1]$. Тогда мы забесплатно свелись к $dp[i-1][j-1]$.

1.2. Ход королем

Банальная динамика вида «Черепашки».
$dp[i][j] = cost[i][j] + min(dp[i+1][j], dp[i][j-1], dp[i+1][j-1])$

1.3. Короля — в угол 3

Проще всего сделать динамику в обратном направлении (то есть начать считать с правой верхней клетки и окончить в левой нижней).
Играем, как и в задаче про поедание массива, всегда за первого. В нужные моменты нужно ставить знак минус. Чтобы найти «профит» от перехода в клетку i, j нужно взять $cost[i][j]$ и вычесть $dp[i][j]$ (так как ходить будет второй).

1.4. Задача Иосифа

Пусть мы знаем ответ на задачу для $n-1$ человека и он равен $dp[n-1]$. Но мы начинали считать с p-го. Значит, к этому ответу нужно прибавить p и взять по модулю n.

1.5. НОВП

1.5.1. $\mathcal{O}(n^4)$

Пусть $dp[i][j]$ — длина НОВП, заканчивающейся ровно в позиции i, j.
База. $dp[0][0] = 0$
Переход. Если $a[i] == b[j]$, то переберем все элементы квадрата $i \times j$, выберем из них такой, что i_1, j_1, $a[i_1] < a[i]$ и $dp[i_1][j_1]$ максимально.
n^2 состояний, n^2 переходов из каждого $\Rightarrow \mathcal{O}(n^4)$

1.5.2. $\mathcal{O}(n^3)$

$dp[i][j]$ — длина НОВП. заканчивающейся ровно в i-ом элементе массива a, и использующей элементы из массива b с индексами до j.
База. $dp[0][0] = 0$
Переход. Переход первый. Не берем $b[j]$. То есть $dp[i][j+1] = max(dp[i][j+1], dp[i][j])$
Переход второй. Берем $b[j]$. Этот переход возможен только если $b[j] > a[i]$. Тогда найдем минимальное i_1 такое, что $i_1 > i$ и $a[i_1] = b[j]$. Делаем $dp[i_1][j+1] = max(dp[i_1][j+1], dp[i][j]+1)$

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

Научимся искать такое i_1 быстрее. Например, это можно сделать бинпоиском. Для каждого $a[i]$ запомним индексы элементов a такие, что $a_i = b_j$ (за квадрат. Можем сжать координаты или использовать unordered_map). Запомнили в какой-нибудь массив. Теперь поиск i_1 работает за $\mathcal{O}(\log n)$, запусти upper_bound.

1.7. $\mathcal{O}(n^2)$

Заметим, что если перебирать сначала j, а внутри i, то результаты запросов $i_1, i_2 \ldots$ не убывают для каждого i. Таким образом, мы можем воспользоваться методом двух указателей (на самом деле одного указателя).

1.7.1. Другой $\mathcal{O}(n^2)$

Изначально сделаем все $dp[0..n][j]$ равными $dp[0..n][j-1]$.
Хотим обновить ответ, то есть найти такие НОВП, что заканчивается на $b[j]$.
Будем вести пересчет внешним циклом по j, внутренним по i. Бежим по i. Помним k, такое что $a[k] < b[j]$ и $dp[k][j-1]$ максимально. Обновляем k, если очередное $a[i] < b[j]$. Если $a[i] = b[j]$, то обновляем ответ значением $dp[k][j]+1$.
Можно заметить, что это можно делать одномерной динамикой.

1.8. Японский кроссворд

Построим вспомогательную строку S: сначала запишем 0, потом a_1 единиц, потом 0, потом a_2 единиц, ... в конце 0.
Например, для теста из условия 4 2 строка S будет выглядеть так 011110110
Насчитаем динамику $dp[i][j]$ — правда ли, что если мы находимся в позиции i строки S и позиции j нашей строки, то мы сможем дополнить строку до конца (как-нибудь).
В этой динамике нужно осторожно обработать случай $S[i] = 0$ (переход может быть в ту же позицию или на позицию дальше) и случаи крайних нулей (чтобы их не обрабатывать, можно записать в начало и конец исходной строки по нулю).
Представим эту динамику как ациклический граф. Тогда ребра — это переходы (0 или 1 ставим). Начальное состояние $[0, 0]$, конечное состояние $[m, n]$. Тогда, чтобы понять, можно ли поставить на позицию *pos* нолик (или единичку), нужно посмотреть, правда ли, что существует путь на этом графе, идущий из начала в конец, проходящий по этому ребру. Посмотреть, можно ли пройти в конец, можем, просто обратившись к соответствующему элементу массива динамики. Чтобы узнать, можем ли мы пройти из начала динамики к началу ребра, можно либо запомнить это (динамикой по обратным ребрам), либо посчитать ту же самую динамику для перевернутых

строк.

1.9. Приблизительное значение

Задача. Дан ряд чисел и число X. Сумма модулей числе в ряду не превосходит 10^4. Чисел тоже не больше 10^4. Хотим расставить между ними знаки так, чтобы сумма была как можно ближе к X.

1.9.1. Простая динамика

Можно сделать двумерную динамику $dp[pos][S]$ — правда ли, что можем набрать сумму ровно S, если берем первые pos элементов. База $dp[0][0] = 1$, переход $dp[pos + 1][S + a_{pos}] = dp[pos + 1][S - a_{pos}] = 1$.

Она работает за $\mathcal{O}(nW)$, где $W = \sum_{i=0}^{n} |a_i|$. И потребляет столько же памяти. По времени нормально, по памяти — нет.

1.9.2. Сводим к рюкзаку

Мы уже хорошо умеем сжимать память до $\mathcal{O}(W)$ в задаче о рюкзаке. А, оказывается, что наша задача довольно просто сводится к рюкзаку.

Что нас останавливает сделать рюкзак? То, что все предметы мы можем взять либо с плюсом, либо с минусом. А давайте изначально все предметы возьмем с минусом, и скажем, что теперь мы ищем вес $T = S + \sum_{i=0}^{n} a_i$.

А если мы захотим взять предмет с плюсом, то нужно прибавить $2a_i$.

Итого у нас есть n предметов $2a_i$, хотим набрать ими вес, как можно более близкий к T.

1.10. Профессор и телефоны/яйца/транзисторы

Задача. Есть N-этажное здание и K яиц. Известно, что начиная с какого-то этажа M яйца разбиваются, а раньше — нет. Нужно за минимальное число бросков найти это M. Гарантируется, что $1 < M < N$

$N \leqslant 10^5$.

1.10.1. $\mathcal{O}(N^2K)$

Будем делать динамику по количеству этажей и количеству яиц. $dp[N][K]$ — ответ на задачу. Будем поддерживать инвариант, что с первого этажа яйца не разбиваются, а с последнего разбиваются.

База: крайние значения ($N = 1,\ K = 1, \ldots$).

Переход: переберем этаж, с которого мы скинем яйцо. Пусть это T. Возможны два случая: яйцо разбилось (тогда свелись к задаче на нижних T этажах и минус одним яйцом) и яйцо не разбилось (свелись к задаче на верхних $N - T + 1$ этажах).

Код перехода:

```
1 for (int T = 2; T < N; ++T)
2     dp[N][K] = min(dp[N][K], max(dp[T][K - 1], dp[N - T + 1][K]));
```

NK состояний, за N переходим.

1.10.2. $\mathcal{O}(N^2 \log N)$

Если яиц хотя бы $\log N$, то их достаточно, чтобы запустить бинпоиск. Поэтому нужно посчитать только ответ для состояний, где $K < \log N$.

1.10.3. $\mathcal{O}(N \log^2 N)$

Посмотрим внимательно на переход. Нас интересует момент, когда максимум из двух величин $dp[T][K-1]$, $dp[N-T+1][K]$ минимален. Заметим, что при возрастании T первая величина растёт, а вторая убывает. Найдем бинпоиском такой момент, когда $dp[T][K-1] \leqslant dp[N-T+1][K]$. Оптимальный переход — либо найденное T, либо $T+1$.
$N \log N$ состояний, $\log N$ тратим на переход $\Rightarrow \mathcal{O}(N \log^2 N)$

1.10.4. $\mathcal{O}(N \log N)$

Пусть $T(N,K)$ — оптимальное T из прошлого пункта. Тогда заметим, что $T(N-1,K) \leqslant T(N,K) \leqslant T(N+1,K) \Rightarrow$ можем применить два указателя и получить $\mathcal{O}(N \log N)$

1.11. Динамика по дереву

TODO

Тема #2: Графы

8 августа

2.1. Хранение графа

Графом называют множество вершин, некоторые из которых соединены ребрами.
Как хранить граф? Есть несколько способов:

```
vector<int> gr[N]; // список смежности. Для каждой вершины храним в списке те, с которыми она смежна
set<int> gr[N]; // то же, что и список смежности, но на сете. Преимущество - можем за O(log n) удалить ребро из графа
bitset<N> gr[N]; // матрица смежности. Табличка n x n. В позиции i, j стоит 1, если есть ребро i->j
```

В первом и втором способах мы можем за $\mathcal{O}$(количество соседей) перебрать всех соседей вершины:

```
for (int u : gr[v]) {
    ...
}
```

В третьем способе так:

```
for (int u = 0; u < N; ++u) {
    if (gr[v][u] == 1) {
        ...
    }
}
```

2.2. Количество треугольников в графе

Треугольником в неориентированном графе называется тройка вершин, каждые две из которых попарно соединены ребром.
Хотим найти количество треугольников в графе.

2.2.1. $\mathcal{O}(V^3)$

Переберем все тройки вершин, для очередной тройки с помощью матрицы смежности проверим за $\mathcal{O}(1)$, треугольник ли она.

2.2.2. $\mathcal{O}(VE)$

Переберем все ребра. Смотрим на очередное ребро. Оно соединяет вершины u и v. Посмотрим всех соседей вершины u, для каждой проверим, является ли она соседом v. Если да — нашли треугольник. В конце не забыли поделить количество треугольников на 3.

2.2.3. $\mathcal{O}(E\sqrt{E})$

Если взять предыдущее решение, и перебирать соседей вершины, у которой меньше соседей, то мы, внезапно, получим асимптотику $\mathcal{O}(E\sqrt{E})$

2.3. Обход в ширину (BFS)

Пусть нам дан граф на V вершинах и E ребрах, вес каждого ребра 1. Зафиксирована вершина v. Хотим найти кратчайшие расстояния от нее до всех других достижимых вершин.
Здесь нам на помощь придет алгоритм обхода графа в ширину.
Сначала посетим вершины, расстояние до которых 0 (это только стартовая вершина). Посмотрим на все ребра, смежные с ней. Это вершины, расстояние до которых 1.
На i-й итерации посещаем вершины, расстояние до которых i. Перебираем ребра из них исходящие. Если встретили вершину, которую до этого не обрабатывали, скажем, что расстояние до нее равно $i + 1$.
Утверждение. Этот алгоритм корректно найдет расстояния до вершин.
Доказательство. Очевидно, что те расстояния, которые нашел алгоритм не меньше кратчайших (он нашел корректные пути, длина любого пути не меньше длины кратчайшего).
Докажем, что они не больше кратчайших (индукция по фактическим расстояниям).
База: для стартовой вершины выполнено.
Переход: Пусть выполнено для уровня i, хотим доказать для уровня $i + 1$. Пусть для какой-то вершины, расстояние до которой $i + 1$, алгоритм посчитал расстояние неверно. Назовем эту вершину u. Тогда существует вершина v такая, что есть ребро $v \to u$ и расстояние до v равно i. Расстояние до v посчитано верно (по индукции). Значит, на i-м уровне BFS'а мы смотрели всех соседей v. Значит, мы рассмотрели ребро $v \to u$ и должны были обновить расстояние до u значением $i + 1$. Противоречие.

2.4. Немного кода

Обычно BFS пишут с очередью. Для очереди всегда выполняется инвариант, что вершины с меньшими расстояниями идут там раньше вершин с большими расстояниями.
В C++ есть структура данных queue. Она нам и нужна.
Приведем реализацию, где для каждой вершины посчитано расстояние.

```
1  void bfs(int v) {
2      queue<int> Q;
3      vector<int> dist(n, INF);
4      dist[v] = 0;
5      Q.push(v);
6      while (!Q.empty()) {
7          int v = Q.front();
8          Q.pop();
9          for (int u : gr[v]) {
10             if (dist[u] > dist[v] + 1) {
11                 dist[u] = dist[v] + 1;
12                 Q.push(u);
13             }
14         }
15     }
16 }
```

2.5. Упражнения

2.5.1. Кратчайший цикл в орграфе

Переберем все вершины, из каждой запустим BFS. Попробуем найти цикл до неё минимального

веса.
Для того, чтобы проверить, что есть цикл через вершину, нужно проверить, есть ли ребро из какой-то посещенной вершины BFS в нее. Кратчайший цикл проходит через первую такую вершину.

```
int min_cycle_with_v(int v) {
    queue<int> Q;
    vector<int> dist(n, INF);
    dist[v] = 0;
    Q.push(v);
    while (!Q.empty()) {
        int u = Q.front();
        Q.pop();
        for (int t : gr[u]) {
            if (dist[t] > dist[u] + 1) {
                dist[t] = dist[u] + 1;
                Q.push(t);
            }
            else if (t == v) {
    return dist[u] + 1;
            }
        }
    }
    return INF;
}
```

Так как мы V раз запускаем BFS, то алгоритм работает за $\mathcal{O}(VE)$

2.5.2. Кратчайший цикл в неорграфе

Здесь у нас проблема в том, что BFS считает одно ребро циклом.
Будем перебирать вершины. Давайте будем идти BFSом из вершины v. Пусть мы рассматриваем вершины на расстоянии i от стартовой. Для каждой посещенной вершины запомним, какая была предыдущая на пути до нее. При проходе BFS'ом не смотрим на обратные ребра. Если увидели ребро, ведущее на предыдущий или следующий уровень, то мы нашли цикл. Пусть это ребро соединяет вершины t, u. Не факт, что цикл проходит через v, но его длина не больше $d[u] + d[t] + 1$. А если он проходит через v, то мы посчитаем его фактическую длину. Таким образом, надо по всем запускам BFS'а выбрать минимум.

2.6. $0-1$ BFS

Теперь у нас веса ребер 0 и 1. Чтобы выполнялся инвариант обхода в ширину, мы должны добавить вершины, посещенные с помощью ребер веса 0 в начало очереди, а вершины, посещенные с помощью ребер веса 1 — в конец (как бы пропускать ребра веса 0 вне очереди).
Для этого есть структура данных дек (deque). Он умеет добавлять в начало/конец и удалять из начала/конца. Все за $\mathcal{O}(1)$

```
int min_cycle_with_v(int v) {
    deque<int> Q;
    vector<int> dist(n, INF);
    dist[v] = 0;
    Q.push(v);
    while (!Q.empty()) {
        int u = Q.front();
        Q.pop();
        for (auto edge : gr[u]) {
            if (dist[edge.to] > dist[u] + edge.w) {
                dist[edge.to] = dist[u] + edge.w;
                edge.w == 0 ? Q.push_front(edge.to) : Q.push_back(edge.to);
            }
        }
    }
    return INF;
}
```

Докажем, что каждая вершина попадет в дек не более двух раз.
Пусть мы в первый раз перешли в вершину из вершины, которая была на уровне i. Если это было ребро веса 0, то расстояние до вершины больше не улучшится $\Rightarrow$ вершина будет в деке ровно 1 раз. Если это было ребро веса 1, то расстояние до вершины стало $i+1$. Тогда, чтобы улучшить расстояние до вершины, мы можем сделать его только i (так как у нас выполняется инвариант, что в БФСе вершины идут в порядке увеличения расстояния, то есть расстояние до нашей вершины не может быть меньше i).
Если мы встречаем вершину во второй раз, то можно либо проверить это и сделать continue, либо ничего не делать (от того, что мы дважды прорелаксируем ребра из нее ничего не ухудшится, только константа станет чуть больше).

2.7. $1-K$ BFS за $\mathcal{O}(EK)$

Теперь у нас веса ребер от 1 До K. Скажем, что у нас не одно ребро веса w, а w ребер веса 1. Таким образом, каждое из E ребер мы разбили на $\leqslant K$ ребер $\Rightarrow$ $\mathcal{O}(EK)$

2.8. $1-K$ BFS за $\mathcal{O}(VK+E)$

Сделаем умнее: будем делать почти такой же BFS как и раньше, только честно хранить вершины на уровнях. Для каждого из VK уровней (их столько, так как кратчайший путь проходит по каждой вершине не более 1 раза) создадим очередь/вектор, который будет содержать вершины, до которых ровно такое расстаяние. Изначально в нулевой очереди стартовая вершина.
Рассматриваем очередное ребро. Если расстояние до вершины улучшилось, то добавим ее в соответствующую очередь. Каждая вершина побывает не более, чем в K очерядях, но здесь, в отличие от случая $0-1$, нам важно не запускать обход из вершины, если мы ее уже обработали, потому что иначе время станет $\mathcal{O}(VK+EK)$

```
void bfs(int v) {
    vector<vector<int> > Q(V * K);
    vector<int> dist(V, INF);
    dist[v] = 0;
    Q[0].push_back(v);
    for (int i = 0; i < V * K; ++i) {
        while (!Q[i].empty()) {
            int u = Q[i].back();
            Q[i].pop_back();
            if (dist[u] != i) continue;
            for (auto edge : gr[u]) {
                if (dist[edge.to] > dist[u] + edge.w) {
                    dist[edge.to] = dist[u] + edge.w;
                    Q[dist[edge.to]].push_back(edge.to);
                }
            }
        }
    }
}
```

Это решение работает за $\mathcal{O}(E + VK)$ и потребляет $\mathcal{O}(VK)$ памяти.

2.9. $1 - K$ BFS с $K + 1$ очередями

Заметим, что в каждый момент времени мы можем использовать только $K+1$ очередь (текущая и K достижимых по ребрам). Значит, нам не нужно VK очередей.
Вот так это можно написать просто:

```
void bfs(int v) {
    vector<vector<int> > Q(K + 1);
    vector<int> dist(V, INF);
    dist[v] = 0;
    Q[0].push_back(v);
    for (int i = 0, start_queue = 0; i < V * K; ++i, start_queue = (start_queue +
        1) % (K + 1)) {
        while (!Q[start_queue].empty()) {
            int u = Q[start_queue].back();
            Q[start_queue].pop_back();
            if (dist[u] != i) continue;
            for (auto edge : gr[u]) {
                if (dist[edge.to] > dist[u] + edge.w) {
                    dist[edge.to] = dist[u] + edge.w;
                    Q[(start_queue + edge.w) % (K + 1)].push_back(edge.to);
                }
            }
        }
    }
}
```