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

Собрано 7 августа 2018 г. в 20:27

Содержание

Тема #1: Рюкзаки

6 августа

1.1. Обычный рюкзак

Задача: даны предметы. У каждого предмета есть вес w_i. Для каждого веса u_i от 0 до W нужно определить, можем ли мы каким-то поднабором предметов набрать ровно u_i. Каждый предмет можно брать только 1 раз.

1.2. Решение динамикой

Давайте будем перебирить предметы в порядке входных данных и считать динамику «Правда ли, что предметами из префикса до i-го можем набрать вес u_j».
База: нулевым префиксом можем набрать вес 0 и только его.
Переход: пришел очередной предмет с весом w_i. Если раньше мы могли набрать вес u_i, то теперь мы можем набрать u_j и $u_j + w_i$.
Код:

```
dp[MAXN][MAXW]
dp[0][0] = 1
for i = 1..n:
    for w = 0..MAXW:
        dp[i][w] = dp[i - 1][w]
        if w - weight[i] >= 0: dp[i][w] |= dp[i - 1][w - weight[i]]
```

В итоге в n-й строке массива единички будут стоять в тех и только тех весах, которые можем набрать.
Это решение, как не сложно заметить, работает за $\mathcal{O}(nW)$ времени и потребляет столько же памяти.

1.2.1. Улучшаем память

Что мы по сути делаем каждый раз? Копируем предыдущий массив и вставляем единички в какие-то новые места. Но для этого нам достаточно поддерживать только один массив размера W.
И код от этого упростился:

```
dp[MAXW]
dp[0] = 1
for i = 1..n:
    for w = MAXW-weight[i]..0:
        if dp[w]: dp[w + weight[i]] = 1
```

Кстати, если мы хотим позволить брать каждый предмет много раз, то цикл по w нужно просто пустить в другую сторону.
Как теперь восстановить ответ (предметы, которые мы взяли в набор)?
Нужно для каждого момента времени просто запомнить, с помощью предмета мы пришли туда в первый раз:

```
prev[MAXW]
dp[MAXW]
dp[0] = 1
for i = 1..n:
    for w = MAXW-weight[i]..0:
        if dp[w] && !dp[w + weight[i]]:
            dp[w + weight[i]] = 1
            prev[w + weight[i]] = i
```

Теперь мы хотим понять, какие предметы нужны, чтобы набрать w:

```
get_items(w):
    ans = []
    while w != 0:
        ans.add(prev[w])
        w -= weight[prev[v]]
    return ans
```

Итак, асимптотика осталась прежней, но мы улучшили память до $\mathcal{O}(W)$.

1.2.2. Bitset

Побитовый сдвиг для типа long long в C++ и побитовое «ИЛИ» двух long long выполняются за $\mathcal{O}(1)$, хотя в long long живут целых 64 бита.

Это позволяет заменить массив bool на массив long long и определить операции побитового сдвига, побитового «ИЛИ» и много других для массива. Все эти операции будут выполняться за $\mathcal{O}(\frac{N}{64})$, где N — размер массива.

Именно такая идея реализована в bitset для C++.

Применим битсет в нашей задаче. По сути, при переходе мы делаем побитовое «ИЛИ» нашего массива с тем же массивом, сдвинутым на $weight[i]$.

```
bitset<MAXW> dp; // размер битсета должен быть известен на момент компиляции
dp[0] = 1
for (int i = 0; i < n; ++i) {
    dp |= dp << weight[i];
}
```

Да, это весь код. Асимптотика теперь $\mathcal{O}(\frac{nW}{64})$. Но в таком случае мы не умеем восстанавливать ответ.

1.3. Взвешенный рюкзак

Теперь у каждого предмета есть своя стоимость. Хотим унести предметы максимальной стоимости.

Возьмем ту же динамику, что была в первом пункте, но теперь $dp[i][j]$ будет обозначать максимальную стоимость предметов веса j, если мы берем отрезки только из префикса до i-го предмета.

```
dp[MAXN][MAXW] <-- inf
dp[0][0] = 0
for i = 1..n:
    for w = 0..MAXW:
        dp[i][w] = dp[i - 1][w]
        if w - weight[i] >= 0: dp[i][w] = min(dp[i - 1][w - weight[i]] + cost[i],
            dp[i][w])
```

Опять же, мы можем соптимизировать память до $\mathcal{O}(W)$, но в этом случае мы не сможем восстановить ответ.
В динамике с памятью за nW ответ восстанавливается точно так же, как и в прошлый раз: помним, какой предмет положили последним, потом восстанавливаем ответ в обратном порядке.

1.4. Рюкзак на отрезке

А теперь нам даны Q запросов вида «а можно ли набрать вес w_i, если использовать только предметы с отрезка $[L_i;\ R_i]$».
Ну давайте сделаем снова динамику с $\mathcal{O}(nW)$ памяти, только для каждого u_i будем запоминать максимальное L, такое что мы можем набрать вес u_i предметами из отрезка $[L;\ i]$

```
dp[MAXN][MAXW]
dp[0][0] = INF
for i = 1..n:
    for w = 0..MAXW:
        dp[i][w] = dp[i - 1][w]
        if w - weight[i] >= 0: dp[i][w] = max(min(i, dp[i - 1][w - weight[i]]),
            dp[i][w])
```

Теперь, чтобы ответить на запрос, нужно посмотреть на $dp[R][w_i]$. Если это $\leqslant L$, то победа.
А еще можно сначала считать все запросы, отсортировать их и отвечать при подсчете динамики. Это позволит сократить память до $\mathcal{O}(W)$, а время останется таким же ($\mathcal{O}(nW + q)$), так как мы умеем сортировать подсчетом.

Тема #2: Наибольшая возрастающая подпоследовательность

6 августа

Нужно из последовательности выбрать подпоследовательность, которая будет (нестрого) монотонно возрастающей.

2.1. Квадратичное решение

Сделаем динамику $dp[pos][len]$ — минимальное такое a_i, что $i \leqslant pos$ и существует НВП длины len, кончающееся в элементе a_i.
База: $dp[0][0] = -INF,\ dp[0][..] = INF$
Переход: зафиксировали конкретные pos и len. Изначально $dp[pos][len] = dp[pos-1][len]$. Как мы можем изменить это значение? Поставив туда a_i. Когда мы можем это сделать? Всегда, когда $dp[pos-1][len-1] < a_i$ (или $\leqslant$, в зависимости от характера монотонности).
Код:

```
1 dp[0][0] = -INF
2 dp[0][1..n] = INF
3 for pos = 1..n:
4     for len = 1..n:
5         dp[pos][len] = dp[pos-1][len]
6         if dp[pos][len - 1] <= a:
7             dp[pos][len] = min(dp[pos][len], a[i])
```

Время работы и память $\mathcal{O}(n^2)$

2.2. Делаем линейную память

Давайте теперь для каждой длины хранить минимальный элемент, на который заканчивается НВП данной длины
Код:

```
1 dp[0] = -INF
2 dp[1..n] = INF
3 for i = 0..n - 1:
4     for len = 1..n:
5         if a[i] >= dp[len - 1]:
6             dp[len] = min(dp[len], a[i])
```

2.3. Получаем $\mathcal{O}(n \log n)$

Посмотрим на предыдущий алгоритм внимательнее. Сколько раз он сделает обновление?
Пусть $ub = dp.upper\_bound(a[i])$
Замечание 1. Массив dp отсортирован по возрастанию.
Замечание 2. Все элементы до ub остались прежними, так как они $\leqslant a[i]$.
Замечание 3. Все элементы после ub остались прежними, так как для них не выполнился бы if из 4 строчки алгоритма.
Таким образом, обновится значение в $dp[ub]$ и только оно!
Получаем следующий код:

```
dp[0] = -INF
dp[1..n] = INF
for i = 0..n - 1:
    ub = dp.upper_bound(a[i])
    dp[ub] = a[i]
```

Это решение для случая нестрогого возрастания. В случае строгого возрастания нужно еще сделать проверку на то, что $a[i] > dp[ub - 1]$

2.4. Восстанавливаем ответ

Для каждой клетки будем запоминать не только значение a_i, но и индекс этого элемента. Теперь, чтобы восстановить НВП, нужно для каждого i запомнить $dp[ub - 1].idx$

Тема #3: Игры

6 августа

3.1. Игра с камнями

Есть кучка из n камней и массив a. Игроки могут убрать из кучки любое число камней из a. Нужно определить, кто победит при правильной игре.
Сделаем динамику выигрышная-проигрышная позиция.
Позиция называется выигрышной, если у игрока, стоящего в этой позиции, есть выигрышная стратегия.
Позиция выигрышная, если из нее есть хотя бы один ход в проигрышную и проигрышная иначе.
База: $dp[0]$ = Проигрышная
Переход: Если какая-то из $pos - a_i$ выигрышная, то эта позиция выигрышная. Иначе — нет.

3.2. Едим массив

У двоих игроков есть массив. За ход они съедают левый или правый элемент массива. Побеждает тот, у кого сумма съеденых элементов больше.
Давайте решать другую задачу. У них есть число A, изначально равное нулю, первый прибавляет съеденный элемент к сумме, второй — вычитает. Первый хочет максимизировать результат, а второй — минимизировать. Нужно сказать, какая сумма будет в конце. Это та же самая задача, так как если сумма положительна, то победил первый, а иначе второй.
Эту задачу тоже решим динамикой.
Пусть у нас будет $dp[i][j]$ — остался отрезок массива от i до j. Динамика будет по подотрезкам (то есть мы перебираем отрезки по возрастанию длины).
И давайте представим, что мы всегда играем за первого игрока (пытаемся максимизировать сумму). Этот результат будет отличаться от хода второго игрока только знаком (так как игра симметричная).
Тогда **база:** $dp[i][i] = a[i]$
Переход: $dp[i][j] = max(a[i] - dp[i+1][j], a[j] - dp[i][j-1])$
Асимтотика и память $\mathcal{O}(n^2)$. Память можно соптимизировать до $\mathcal{O}(n)$.

Тема #4: Ациклические графы

6 августа

Переходы динамики образуют ацикличный ориентированный граф.
Поэтому задачи динамики сводятся к задачам на ацикличном графе и наоборот.

4.1. Самый длинный путь

Хотим найти самый длинный путь в ацикличном орграфе. Длина пути — количество ребер в нем.
Сделаем динамику $dp[v]$ «какой самый длинный путь, начинающийся в вершине v».
Переход понятен: нужно взять максимум из тех, в кого ведут ребра из v, и прибавить единицу к ответу.
База тоже понятна: если из вершины нет ребер, то ответ для нее ноль.
Непонятно, в каком порядке пересчитывать. Это можно делать в порядке топологической сортировки, а можно применить подход, называемый ленивостью. На нем мы остановимся подробнее.
Для каждой вершины будем помнить, посещали ли мы её. Если не посещали, то насчитаем ответ и запомним его.
Если посещали, то вернем уже насчитанный ответ. Асимптотика не ухудшилась.

```
1  bool used[MAXN];
2  int dp[MAXN];
3
4  int lazy_dp(int v) {
5      if (used[v]) return dp[v];
6      used[v] = true;
7      dp[v] = 0;
8      for (int u : gr[v]) {
9          dp[v] = max(dp[v], lazy_dp(u));
10     }
11     return dp[v];
12 }
13
14 ...
15
16 int ans = 0;
17 for (int i = 0; i < n; ++i) {
18     ans = max(ans, lazy_dp(i));
19 }
```

4.2. Транзитивное замыкание

Дан ориентированный ациклический граф на V вершинах и E ребрах. Хотим построить матрицу достижимости (транзитивное замыкание). Это матрица $V \times V$, в клетке i, j которой стоит единица, если из i можно добраться до j. Будем считать, что i достижима из i.
Будем делать это ленивой динамикой. Для каждой вершины сохраним bitset вершин, достижимых из этой. При переходе надо делать побитовое «ИЛИ» с bitset'ом потомка.

Тема #5: Разбиения на слагаемые

6 августа

5.1. Разбиение на слагаемые не больше K

Дано число N и число K. Хотим найти количество способов разбить N на возрастающие слагаемые не больше K.
Сделаем динамику $dp[K][N]$ — ответ на задачу.
База: $dp[0][0] = 1$
Переход: мы можем либо взять K в разбиение, или нет. Если мы берем, то должны прибавить $dp[K-1][N-K]$ (хотим набрать число $N-K$ слагаемыми не больше $K-1$, так как слагаемые должны быть возрастающими)
Если мы не берем K в разбиение, то должны прибавить $dp[K-1][N]$.
Это $\mathcal{O}(n^2)$ времени и памяти. Но память можем соптимизировать до $\mathcal{O}(N)$, так как можем хранить только две последние строчки.

```
int dp[2][MAXN];

int prev = 0;
int cur = 1;
dp[prev][0] = 1;
for (int k = 1; k <= K; ++k, swap(cur, prev)) {
    for (int n = 0; n <= N; ++n) {
        dp[k][n] = dp[k - 1][n] + (n >= k ? dp[k - 1][n - k] : 0);
    }
}
```

5.2. Разбиение ровно на K слагаемых

Нам нужно найти количество способов разбить число N на K неупорядоченных различных слагаемых.
Будем делать динамику $dp[K][N]$ — ответ на задачу.
У нас будет два случая:

1. Среди слагаемых нет единицы. Тогда вычтем единицу из каждого слагаемого и сведемся к уже посчитанному $dp[K][N-K]$
2. Среди слагаемых есть единица. Тогда вычтем единицу из N, а так как среди других слагаемых не может быть единицы, вычтем её из оставшихся $K-1$. $dp[K-1][N-K]$

Итого $dp[K][N] = dp[K][N-K] + dp[K-1][N-K]$. И снова заметим, что можем хранить только две последние строки $\Rightarrow$ память $\mathcal{O}(N)$, время $\mathcal{O}(N^2)$.