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

Собрано 7 августа 2018 г. в 17:53

Содержание

Тема #1: ПСП (продолжение)

5 августа

1.1. Другой способ посчитать количество ПСП

Давайте теперь занчением динамики $dp_{position,balance}$ будет количество способов дополнить до правильной скобочной последовательности из $2n$ скобок префикс, начиная с позиции *position*.
База: $dp_{2n,0} = 1,\ dp_{2n,??} = 0$.
Переход: $dp_{pos,bal} = dp_{pos+1,bal+1} + dp_{pos+1,bal-1}$
Обратите внимание, в каком порядке мы считаем динамику:

```
dp[2*n][0] = 1;
for (int pos = 2n - 1; pos >= 0; --pos) {
    for (int bal = 0; bal <= n; ++bal) {
        dp[pos][bal] = dp[pos + 1][bal + 1];
        if (bal != 0) dp[pos][bal] += dp[pos + 1][bal - 1];
    }
}
```

Теперь количество ПСП длины $2n$ лежит в $dp_{0,0}$. Но нам этот вариант динамики пригодится именно в виде «сколько есть способов дополнить префикс, начиная с позиции *position* и балансом *balance* до ПСП из $2n$ скобок».

1.2. Скобочная последовательность по номеру

Нужно вывести K-ю ПСП (в лексикографическом порядке). N фиксировано.
K нумеруется с нуля. Номер ПСП == количество ПСП, которые лексикографически меньше данной ПСП.
Если делаем это в тупую (выписываем все подряд), то получим экспоненту.
Воспользуемся динамикой из предыдущего пункта.
Пусть мы уже построили какой-то префикс и стоим в позиции *pos* с балансом *bal*. Будем считать не K лексикографическую ПСП среди всех, а K-ю среди все, что начинаются с данного префикса. Как определить, какая скобка должна идти дальше?
Давайте посмотрим, сколько существует способов дополнить нашу последовательность до ПСП, если мы поставим открывающую скобку. Это можем посмотреть, заглянув в $dp_{pos+1,bal+1}$. Пусть это A.
Все последовательности, у которых дальше идет закрывающая скобка лексикографически больше всех последовательностей, у которых идет открывающая.
Если K меньше, чем A, то нам нужно поставить открывающую скобку. Иначе нужно поставить закрывающую и вычесть из K A (все те, кто лексикографически меньше нас).
Получаем такой код:

```
K = ...
bal = 0
for i = 0..2 * n - 1:
    if K < dp[i + 1][bal + 1]:
        print('(')
        bal++
    else:
        print(')')
        K -= dp[i + 1][bal + 1]
        bal--
```

1.3. Номер по ПСП

Дано ПСП. Найти номер K этой ПСП.
Идем по ПСП. Поддерживаем баланс и текущее K (изначально оно 0). Если встретили открывающую, то просто пересчитываем баланс. Если встретили закрывающую, то мы больше всех, у которых на этой позиции стояла открывающая. Количество таких можем посчитать нашей динамикой.

1.4. Номер по ПСП из двух и более типов скобок

Сложность в том, чтобы посчитать сколько есть ПСП с таким префиксом, ведь у нас есть какой-то стек... Но ведь это почти никак не вляет на динамику! По стеку мы всегда однозначно определяем, какую закрывающую скобку поставить, а открывающую скобку можем выбрать одну из двух. Таким образом, единственное, что мы изменяем в прошлой динамике — домножение на два при переходе к следующей скобке.

1.5. Следующая ПСП

Следующая ПСП лексикографически больше $\Rightarrow$ есть момент, где в первой стоит открывающая скобка, а во второй закрывающая (а до этого они равны).
Мы хотим найти такую последнюю открывающую, которую можно поменять на закрывающую. Нашли первую открывающую с конца, при замене которой на закрывающую, баланс на префиксе останется неотрицательным (баланс в ней должен быть $\geqslant 2$). Меняем ее. Конец добиваем жадно (сначала ставим открывающие, а потом закрывающие).

1.6. А если несколько типов скобок?

WARNING! Здесь мы предполагаем, что скобки упорядочены следующим образом: ()[]{}.... При других порядках скобок этот алгоритм надо совсем чуть-чуть поменять.
Идем со стеком. Какую скобку можем заменить? Мы точно не можем заменить закрывающую скобку на другую закрывающую, так как тогда последовательность перестанет быть правильной.
У нас есть два случая:

1. Мы ставим открывающую скобку
2. Мы ставим закрывающую скобку

Как обычно, из всех вариантов замен, мы выберем самый ближний к концу. А если их несколько, то лексикографически наименьший.

Как определить, что к данной скобке мы можем применить случай 1? Очевидно, что если мы и будем заменять эту скобку, то на ближайшую б**о**льшую лексикографически открывающую скобку. Надо не забыть проверить, что у нас останется достаточно места, чтобы поставить скобки, соответствующие тем скобкам, что лежат в стеке.
Как определить, что мы можем применить случай 2? Нужно просто проверить, что скобка, соответствующая последней скобке из стека, лексикографически больше той, в которой мы стоим. Выбор скобки осуществляется однозначно.
Пусть теперь мы выбрали скобку, и на что ее менять. Поставим ее туда. Надо заполнить хвост. Пусть хвост состоит из k элементов и в стеке t элементов. Тогда жадно заполним первые $k-t$ элементов хвоста (поставить сначала открывающие круглые скобки, потом закрывающие их же). Последние t элементов будут соответствовать элементам стека.
Этот алгоритм работает за $\mathcal{O}(n)$

1.7. Сочетание по номеру

Даны числа n и k. Хотим научиться искать k-е сочитание из n элементов.
Будем по очереди вставлять элементы в сочетание. Будем действовать по тому же принципу, что и в задаче про скобки. Будем искать k-е сочетание, среди тех, что начинаются на такой же префикс, как и уже построенное.
Так как в сочетании нам не важен порядок элементов, мы можем строить его в возрастающем порядке.
Заметим, что единственное, что мы должны помнить об уже построенном префиксе сочетания, — последнее число в нем.
Сделаем вспомогательную динамику $dp_{pos,last}$ — сколько существует способов дополнить сочетание из pos элементов до k элементов, если можем использовать только числа, не меньшие $last$.
Ее посчитаем за $\mathcal{O}(nk)$
Хотим определить очередной элемент сочетания. Идем по i от $last+1$ до n, пока

$$\sum_{i=last+1}^{p} dp_{pos+1,i+1} < K$$

Берем подходящее p, вычитаем из K вышенаписанную сумму, запускаемся от следующей позиции.
Заметим, что предподсчет работает за $\mathcal{O}(nk)$, а само решение за $\mathcal{O}(k+n)$, так как $last$ на каждом уровне увеличивается.

1.8. Номер по сочетанию

Идем по перестановке. При встрече очередного элемента a_i прибавляем к ответу $\sum_{t=a_{i-1}}^{a_i-1} dp_{i+1,t+1}$.

1.9. Следующее сочетание

Отсортируем сочетание (за $\mathcal{O}(min(n+k, k\log k))$)
Будем идти с хвоста, пока не встретим такой элемент, который можем увеличить. Мы можем увеличить элемент a_i, если в его суффиксе нет какого-то элемента из $[a_i+1; n]$.
В случае сочетания это будет первый с конца такой элемент на позиции $k-j$, что $a_{k-j-1} \neq n-j$

(в 0-индексации). Хвост заполним жадно.

1.10. Перестановка по номеру

По опыту предыдущих задач, мы знаем, что нам нужно научиться отвечать на запрос «сколько существует способов дополнить префикс до полной перестановки». Но на этот запрос мы можем отвечать за $\mathcal{O}(1)$, с предподсчетом за $\mathcal{O}(n)$, так как существует $k!$ способов заполнить суффикс длины k k числами.
Пытаемся поставить i-е число. Для этого надо K поделить нацело на $(n-i)!$. При переходе не забудем отнять из K $(n-i)! \cdot \lfloor \frac{K}{(n-i)!} \rfloor$. Теперь нам нужна структура данных, которая умеет удалять k-ю порядковую статистику из множества. Пока что мы умеем делать это за $\mathcal{O}(n)$, но скоро научимся за $\mathcal{O}(\log n)$ с деревом отрезков.
Весь алгоритм работает за $\mathcal{O}(n^2)$ без ДО и $\mathcal{O}(n \log n)$ с ним.

1.11. Номер по перестановке

Идем по перестановке, встретили очередной элемент. Мы хотим узнать, сколько элементов в множестве меньше, чем он, и прибавить к ответу это число, помноженное на $(n-i)!$. И не забудем удалить элемент из множества.
$\mathcal{O}(n^2)$ без крутых структур и $\mathcal{O}(n \log n)$ с ними.

1.12. Следующая перестановка

Пойдем с хвоста, пока не встретим такую позицию i, что $a_i < a_{i+1}$. Теперь изменять будем только суффикс, начиная с i-го элемента. Для элемента a_i за $\mathcal{O}(n-i)$ (проходом по суффиксу) найдем минимальное такое a_j, что $a_j > a_i$. Поменяем a_j и a_i местами, отсортируем суффикс $[i+1; n]$ в порядке возрастания. Сортировать можем за $\mathcal{O}(n)$ подсчетом.
Итого $\mathcal{O}(n)$
В C++ есть функция next_permutation. Вы не поверите, что она делает!

Тема #2: Динамическое программирование

5 августа

2.1. Подпалиндром

Хотим найти в строке наибольшую подстроку, которая является палиндромом.
Будем делать квадратичную динамику. Пересчитывать ее будем по возрастанию длины.
База: подстроки длины 0 и 1 — палиндромы.
Переход: хотим понять, является ли палиндромом подстрока длины len, начинающаяся в позиции i. Для этого нужно сравнить символы на позициях i и $i + len - 1$. Если они не равны, то палиндромом наша подстрока точно не является. Иначе, нужно посмотреть на уже подсчитанное значение динамики от длины $len - 2$ и позиции $i + 1$.

```
dp[0][...] = true
dp[1][...] = true
for (int len = 2; len <= n; ++len)
    for (int i = 0; i <= n - len)
        dp[len][i] = (s[i] == s[i + len - 1]) && dp[len - 2][i + 1]
```

Чтобы найти ответ, нужно пробежаться по массиву и выбрать максимальное такое len, что в этой строке есть хотя бы одно true
Если подпалиндромом мы считаем подпоследовательность строки, являющуюся палиндромом, то тут будет похожая динамика. $dp[len][L]$ — длина наибольшего подпалиндрома на подстроке $[L;\ L + len - 1]$.
База: $dp[1][..] = 1$, $dp[0][..] = 0$
Переход: У нас есть три варианта:

1. Взять два крайних символа (если они равны) и запуститься от внутреннего отрезка
2. Сказать, что самый левый символ отрезка не входит в палиндром
3. Сказать, что самый правый символ отрезка не входит в палиндром

Код перехода такой:

```
dp[len][L] = 0
if (s[L] == s[L + len - 1]) dp[len][L] = dp[len - 2][L + 1] + 2;
dp[len][L] = max(dp[len][L], dp[len - 1][L]);
dp[len][L] = max(dp[len][L], dp[len - 1][L + 1]);
```

2.2. Свертка

Дана строка. Мы можем сворачивать ее по следующему правилу:
aaaaaaahahehaheaaaaaaahahehahe → 2(7(a)2(hahe))
Нужно получить минимальную по длине строку.
Будем решать это динамикой по подстрокам. Хотим узнать ответ для подстроки $s_{i...j}$.
У нас есть три варианта:

1. Не сворачивать эту подстроку

2. Не сворачивать подстроку полностью. Перебрать место разбиения, посмотреть в уже посчитанные результаты от двух половин.

3. Применить свертку ко всей подстроке. Для этого переберем делители $j-i+1$, попытаемся свернуть по каждому из них.

С первыми двумя случаями все ясно — первый работает за $\mathcal{O}(1)$, второй за $\mathcal{O}(j-i+1)$ и с этим ничего не поделаешь. Но непонятно, как жить с третьим случаем, ведь мы не умеем быстро определять, можно ли свернуть подстроку на блоки по i элементов.
Для этого подсчитаем еще одну динамику:

2.2.1. LCP

Двумерная динамика $LCP_{i,j}$ — длина наибольшего общего префикса суффиксов строки s, начинающихся с элементов i и j.
База: $LCP_{..,n} = LCP_{n,..} = 0$.
Переход:

$$LCP_{i,j} = \begin{cases} 0 & s_i \neq s_j \\ dp_{i+1,j+1} & \text{иначе} \end{cases}$$

LCP считается за $\mathcal{O}(n^2)$
С помощью этой динамики мы можем проверить, правда ли, что подстроку на позициях $i \dots j$ можно побить на k одинаковых кусков. Для этого нужно лишь убедиться, что

$$LCP_{i,i+len} \geqslant j - (i + len)$$

Не забудем посмотреть значение динамики $dp_{i,i+len-1}$, так как, возможно, свертываемую часть тоже можно свернуть.
Итого динамика работает за $\mathcal{O}(n^3)$

2.3. Recover

Дана ПСП, некоторые скобки заменены на вопросы. Нужно сказать, сколько есть способов восстановить ПСП.
Возьмем динамику из лекции прошлого дня (про подсчет количества ПСП через баланс и длину). Будем делать все то же самое, только если скобка определена, то сделаем один переход, а не два.