* [Динамическое программирование на поддеревьях](https://foxford.ru/lessons/32281/conspects/1)
* [Динамическое программирование на подмножествах](https://foxford.ru/lessons/32281/conspects/2)

**Динамическое программирование на поддеревьях**

Динамическое программирование на графах — это динамическое программирование, где каждой вершине графа сопоставляется значение функции. Наиболее часто возникает динамическое программирование на поддеревьях, где есть корневое дерево, и функция вычисляется от листьев к корню, то есть функция в каждой вершине вычисляется, если вычислены функции во всех потомках данной вершины. Для вычисления используется обход в глубину, запускаемый из корня, который в каждой вершине запускается из всех потомков данной вершины, затем вычисляет значение функции в данной вершине.

Если дано дерево без корня, то, как правило, необходимо подвесить его за любую вершину, выбрав её в качестве корня.

Рассмотрим пример задачи. Дан взвешенный граф, являющийся деревом с n вершинами. Необходимо найти две наиболее удалённые вершины.

Эту задачу можно решить при помощи алгоритма Флойда за O(n3). Если же перебирать каждую вершину, и от каждой вершины запускать алгоритм обхода в глубину для поиска самой удалённой от неё (пути в дереве можно искать при помощи обхода в глубину вместо обхода в ширину), то получится решение сложности O(n2). Но при помощи динамического программирования на поддереве можно решить эту задачу за O(n).

Пусть в графе n вершин, граф задан при помощи списков смежности

**vector<pair<int, int>> g**

, где g[u] — список смежности вершины u, состоящий из пар (конец ребра, вес ребра).

Функция динамического программирования будет такая: f(u) — длина максимального пути от вершины u до одного из листьев дерева (находящихся в поддереве данной вершины, то есть рассматриваются пути, которые спускаются от u в направлении корня).

Тогда ответ может быть найден одним из двух способов. Либо это путь, который начинается в корне и идёт к одному из листов, тогда его длина равна f(корень). Либо этот путь "перегибается" в какой-то вершине u, то есть вершина u является ближайшей к корню вершиной пути. Тогда этот путь состоит из двух частей, спускающихся из вершины u вниз, то есть нужно для вершины u взять два наибольших пути в её потомков.

**int ans = -INF;**

**void dfs(int u, int parent)**

**{**

**int max1 = -INF, max2 = -INF;**

**for (auto [v, wt]: g[u]) {**

**if (v != parent) {**

**dfs(v, u);**

**int curr\_dist = f[v] + wt;**

**if (curr\_dist > max1) {**

**max = max1;**

**max1 = curr\_dist;**

**}**

**else if (curr\_dist > max2)**

**max2 = curr\_dist;**

**}**

**}**

**if (max2 > -INF)**

**ans = max(ans, max1 + max2);**

**f[u] = max1;**

**}**

Параметр parent, передаваемый в функцию обхода в глубину, нужен для того, чтобы не перейти в родителя данной вершины (то есть в вершину, из которой мы пришли в данную вершину), поэтому dfs нужно вызывать, например, от вершины номер 0, а в качестве parent передать несуществующую вершину, например, так:

**dfs(0, -1);**

Также нужно учесть возможность пути, который начинается из корня и идёт вниз к листу (такой путь может оказаться ответом, если у вершины root ровно один потомок:

**ans = max(ans, f[0])**

**Максимальное независимое множество вершин**

Рассмотрим задачу - выбрать в дереве максимальное число вершин так, чтобы никакое ребро не соединяло две выбранные вершины (максимальное независимое множество вершин).

Будем решать задачу динамическим программированием по поддеревьям. Рассмотрим какое-то поддерево и попробуем выбрать в нём максимальное число независимых вершин. Если мы включим корень поддерева в выбранное множество, то прямых потомков этой вершины выбирать уже нельзя, а если не включим - то каждого потомка можно как выбрать, так и не выбрать.

Поэтому будем насчитывать две функции динамического программирования.

f0(u) — максимальное независимое множество вершин, которое можно выбрать в поддереве вершины u, если не выбирать вершину u.

f1(u) — максимальное независимое множество вершин, которое можно выбрать в поддереве вершины u, если выбрать вершину u.

Формулы пересчёта такие (здесь v - любой потомок вершины u).

f0(u)=∑vmax(f0(v),f1(v))

f1(u)=1+∑vf0(v)

Если вершина u является листом, то f0(u)=0, а f1(u)=1, но при правильной реализации эти случаи даже не надо специально обрабатывать.

Дерево подвешивается за произвольную вершину root, а ответом является максимум из f0(root) и f1(root)

#### Динамическое программирование на подмножествах

### Простая задача на динамическое программирование по подмножествам

Рассмотрим следующую задачу. Имеется n студентов и n заданий. Задания нужно распределить по студентам, каждому студенту выдать задание, всем студентам выдать разные задания. При этом каждый студент любит одни задания и не любит другие, а именно, есть двумерный массив

**likes**

, где

**likes[i][j]**

 равно 1, если студент номер i хочет получить задание номер j, и равно 0 в противном случае. Студенты и задания будут нумероваться от 0 до n−1.

Необходимо посчитать количество способов распределить задания по студентам так, чтобы каждому студенты было выдано задание, которое ему нравится.

Непосредственный перебор всех способов распределить задания будет иметь сложность O(n!), но при помощи динамического программирования можно избавиться от факториала, хотя решение всё равно будет иметь экспоненциальную сложность.

А именно, пусть мы распределили первые k заданий по каким-то студентам. Выберем подмножество студентов, которое могло получить первые k заданий. Для каждого такого подмножества посчитаем количество способов раздать эти задания студентам.

Состояниями нашей динамики будет комбинация из числа k и подмножества студентов, которые получили эти задания. Здесь может показаться, что у динамики будет n×2n состояний, на самом деле состояний будет только 2n, так как каждому выбранному подмножеству студентов сопоставлено ровно одно значение k - а именно, число студентов в выбранном подмножестве.

Итак, пусть есть произвольное подмножество студентов s, сопоставим каждому подмножеству студентов функцию f(s), равную количеству способов распределить первые k заданий среди студентов из подмножества s, где k равно количеству элементов в s.

Начальное значение - для пустого множества студентов значение f равно 1 (пустое множество заданий по пустому множеству студентов распределяется единственным способом).

Обсудим подробности реализации. Кодировать множества будем при помощи битовых масок, то есть каждому подмножеству сопоставляется число из n бит. Минимальное значение числа - 0, оно соответствует пустому множеству, максимальное значение маски равно 2n−1, оно соответствует всему множеству. В программе на C++ это значение можно получить, как

**(1 << n) - 1**

. Элементы нумеруются с нуля, как биты чисел.

Далее нам необходимо работать с подмножествами, это делается при помощи битовых операций. Например:

а) проверить, принадлежит ли элемент номер i множеству mask:

**if (mask & (1 << i))**

б) добавить элемент номер i в множество mask:

**mask | (1 << i)**

в) удалить элемент номер i из множества mask (если он в него включён):

**mask ^ (1 << i)**

.

Для перебора всех подмножеств просто будем перебирать все значения mask в арифметическом порядке, начиная с 0. При таком переборе при пересчёте каждой маски гарантируется, что все подмаски данной маски (то есть все подмножества данного множества) уже были рассмотрены.

В данной задаче вычисление значения f(mask) произодится так. Переберём всех студентов j от 0 до n−1, входящих в данное подмножество, то есть в маске должен быть бит номер j. Назначим студенту j задание с максимальным номером k−1, если этот студент хочет получить это задание. Тогда к ответу нужно добавить значение функции на маске, полученной выбрасыванием из данной маски студента номер j.

Пример решения.

**const int MAX = 20;**

**int likes[MAX][MAX];**

**unsigned long long f[1 << MAX];**

**f[0] = 1;**

**for (int mask = 1; mask < (1 << n); ++mask) {**

**f[mask] = 0;**

**int count\_ones = 0;**

**for (int j = 0; j < n; ++j) {**

**if (mask & (1 << j))**

**count\_ones += 1;**

**int last\_task = count\_ones - 1;**

**for (int j = 0; j < n; ++j)**

**if (likes[j][last\_task] && (mask & (1 << j)))**

**f[mask] += f[mask ^ (1 << j)];**

**}**

**}**

**cout << f[(1 << n) - 1] << endl;**

Сложность этого решения будет O(n2n). Обратим внимание на то, что массив f будет большим, поэтому его лучше делать глобальным (объявлять вне функции main), иначе программе может не хватить величины стека для хранения локального массива.

### Решение задач о гамильтоновых путях и циклах, задачи коммивояжёра

Ранее предлагалось решать задачи коммивояжёра при помощи перебора, то есть сложность решения была O(n!). Между тем можно избавиться от факториала, если использовать динамическое программирование. А именно, когда мы переходим к очередной вершине, то нам неважно, каким именно способом мы обошли все уже посещённые вершины, важна либо возможность сделать это (если решается задача о существовании гамильтонова пути), или количество способов обойти эти вершины (если решается задача о количестве гамильтоновых путей), или минимальная длина пути, проходящего через эти вершины (в задаче коммивояжёра).

Но при переходе к от меньшей маски к большей необходимо добавить к пути одну новую вершину. При этом вершина не может быть добавлена к произвольной из посещённых вершин, а только к той вершине, которая является конечной для пути. Поэтому состоянием динамики должно быть не только множество посещённых вершин, но и вершина, в которой закончился путь, то есть это будет массив

**int f[1 << n][n]**

Первый параметр элемента массива - это маска посещённых вершин, а второй параметр — номер последней вершины на пути. Эта вершина должна входить в маску.

В зависимости от задачи бывают разные вариации этой динамики. Например, в задаче коммивояжёра и в задаче построения цикла необходимо обойти все вершины, поэтому цикл можно разомкнуть в любой вершине. Выделим какую-либо вершину, например, вершину с номером n−1, и будем считать, что эта вершина всегда является начальной для пути и уже обойдена, то есть к любой маске нужно добавить эту вершину. Тогда массив будет размером

**int f[1 << (n - 1)][n]**

После завершения заполнения динамики нужно проверить, можно ли замкнуть цикл, то есть нужно рассмотреть такие состояния, у которых конечная вершина номер i соединена с выбранной начальной вершиной номер n−1.

Для восстановления ответа в подобных задачах для каждого состояния динамики нужно запоминать предка - номер предпоследней вершины, то есть вершина, которая станет последней, если выкинуть последнюю вершину. Тем самым для каждого состояния понятно, из какого состояния оно получено.

### Построение максимального паросочетания

Рассмотрим следующую задачу - дан неориентированный невзвешенный граф, нужно выбрать в нём максимальное множество рёбер, не имеющих попарно общих вершин.

У этой задачи есть эффективное полиномиальное решение при помощи алгоритма Куна, но мы рассмотрим решение методом динамического программирования.

Пусть mask - произвольное подмножество, а f[mask] - максимальное паросочетание в выбранном подмножестве вершин.

Выберем в этом подмножестве какой-то элемент, например, самую большую по номеру вершину, входящую в mask, то есть такое максимальное i, что

**(mask & (1 << i))**

 - не 0.

Тогда возможны варианты

а) Вершина i не входит в паросочетание, тогда выбросим её из mask, и рассмотрим

**f[mask ^ (1 << i)]**

.

б) Вершина j входит в паросочетание, тогда переберём парную ей вершину. Это такое j, что j входит в mask и вершины i и j соединены ребром. Тогда выбросим из маски две вершины i и j и ответ будет равен

**1 + f[mask ^ (1 << i) ^ (1 << j)]**

.

Из этих вариантов нужно выбрать вариант с наибольшим значением.