ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ВЫСШАЯ ШКОЛА ЭКОНОМИКИ

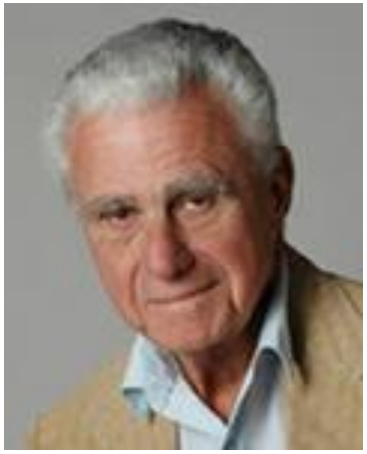
Алгоритмы и структуры данных

Семинар 8

Динамическое программирование 1

2019 - 2020

Динамическое программирование



***Richard Ernest Bellman*; 1920 — 1984**

Ричард Беллман

Принцип динамического программирования: для отыскания решения поставленной задачи решается похожая, но более простая. При этом осуществляется переход к еще более простым и так далее, пока не доходят до тривиальной.

Динамическое программирование обычно применяется к задачам, в которых искомый ответ состоит из частей, каждая из которых в свою очередь дает оптимальное решение некоторой подзадачи.

Динамическое программирование полезно, когда на разных путях многократно встречаются одни и те же подзадачи . Основной технический прием – запоминать решение подзадач на случай, если та же подзадача встретится вновь.

О термине Динамическое программирование

Словосочетание «динамическое программирование» впервые было использовано в 1940-х годах Р. Беллманом для описания процесса нахождения решения задачи, где ответ на одну задачу может быть получен только после решения задачи, «предшествующей» ей.

Центральный результат теории динамического программирования - уравнение Беллмана, которое переформулирует оптимизационную задачу в рекурсивной форме

Слово «программирование» в словосочетании «динамическое программирование» к традиционному программированию (написанию кода) почти никакого отношения не имеет.

Оно имеет смысл как в словосочетании «математическое программирование», которое является синонимом слова «оптимизация».

Слово «программа» в данном контексте скорее означает оптимальную последовательность действий для получения решения задачи.

Об оптимизации

В типичном случае динамическое программирование применяется к ***задачам оптимизации***. У такой задачи может быть много возможных решений, но требуется выбрать оптимальное решение, при котором значение некоторого параметра (целевой функции) будет минимальным или максимальным.

Оптимизация — в математике, информатике и исследовании операций - задача нахождения экстремума (минимума или максимума) целевой функции в некоторой области конечномерного векторного пространства, ограниченной набором линейных и/или нелинейных равенств и/или неравенств.

Задачи оптимизации могут решаться полным перебором, рекурсивно и т.п.

Часто задачи оптимизации не имеют точного решения (или его невозможно найти известными методами)

Еще об оптимизации

Оптимизация — процесс максимизации выгодных характеристик, соотношений (например, оптимизация производственных процессов и производства), и минимизации расходов.

Задача оптимизации сформулирована, если заданы:

– критерий оптимальности (экономический, технологические требования — выход продукта, содержание примесей в нем и др.);

– варьируемые параметры (например, температура, давление, величины входных потоков в процессах переработки горного и др. сырья), изменение которых позволяет влиять на эффективность процесса;

– математическая модель процесса;

– ограничения, связанные с экономическими и конструктивными условиями, возможностями аппаратуры, требованиями взрывобезопасности и др.

Сравнение ДП и алгоритмов «Разделяй и властвуй»

Алгоритмы типа «Разделяй и властвуй» делят задачу на независимые подзадачи, эти подзадачи – на более мелкие подзадачи, затем собирают решение основной задачи «снизу вверх».

Динамическое программирование применимо тогда, когда подзадачи не являются независимыми, т.е. когда у подзадач есть общие подподзадачи.

Алгоритм, основанный на ДП, решает каждую из подзадач один раз, запоминает решение в специальной таблице. Это позволяет не вычислять ответ к уже встречавшейся подзадаче.

®ДП применяется к задачам оптимизации. У таких задач может быть много возможных оптимальных решений.

Задачи, которые можно решать методом ДП

Динамическое программирование может применяться когда в задаче есть:

- перекрывающиеся подзадачи;
- оптимальная подструктура;
- возможность запоминания решения часто встречающихся подзадач.

Для таких задач ДП заменяет полный перебор или рекурсию.

Динамическое программирование — это когда у нас есть задача, которую непонятно как решать, и мы разбиваем ее на меньшие задачи, которые тоже непонятно как решать. (с) А. Кумок.

Пример: простая задача о последовательностях

Последовательность Фибоначчи F_n задается формулами:

$$F_0 = 1, F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2} \quad \text{при } n > 1$$

Необходимо найти F_n по номеру n.

Логичный способ решения – с помощью рекурсии:

```
int F(int n)
{
        if (n < 2)
                return 1;
        else
                return F(n - 1) + F(n - 2);
}
```

Пример: простая задача о последовательностях (продолжение)

Однако при больших n программа будет работать долго: одни и те же промежуточные данные вычисляются по несколько раз — число операций нарастает с той же скоростью, с какой растут числа Фибоначчи — экспоненциально

Решение:

```
F[0] = 1; F[1] = 1;
for (i = 2; i < n; i++)
    F[i] = F[i - 1] + F[i - 2];
```

Это - **классическое решение для** динамического программирования:

- сначала решили все подзадачи (нашли все F_i для $i < n$),
- затем, зная решения подзадач, нашли ответ : $F_n = F_{n-1} + F_{n-2}$

(F_{n-1} и F_{n-2} найдены ранее).

Подходы к решению задач методом ДП

Обычно используются два подхода к решению задач ДП:

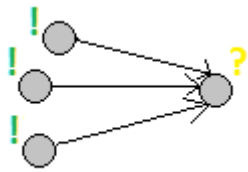
- **нисходящее** динамическое программирование: задача разбивается на подзадачи меньшего размера, они решаются и затем комбинируются для решения исходной задачи (рекурсия).

Часто используется запоминание для полученных решений подзадач.

- **восходящее** динамическое программирование: все подзадачи, которые впоследствии понадобятся для решения исходной задачи, просчитываются заранее и затем используются для построения решения исходной задачи (таблица).

  Этот способ лучше нисходящего программирования в смысле размера необходимой памяти и количества вызова функций, но иногда нелегко заранее выяснить, решение каких подзадач потребуется в дальнейшем.

Порядок пересчета ДП



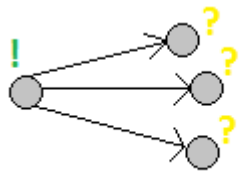
Существует три порядка пересчёта:

1) Прямой порядок:

Состояния последовательно пересчитывается исходя из уже посчитанных.

2) Обратный порядок:

Обновляются все состояния, зависящие от текущего состояния.



3) Ленивая динамика:

Рекурсивная мемоизированная функция пересчёта динамики. Заполним таблицу значениями, которые точно не могут быть получены при вычислениях. Например, -1 в некоторых задачах.

Если при расчетах обращаемся к клетке, значение которой не равно -1, берем это значение. Иначе – выполняем рекурсивный вызов.

Алгоритм ДП

1. Описать оптимальные решения, т.е. определить целевую функцию, которую надо минимизировать / максимизировать / др.
2. Составить рекуррентное соотношение, связывающее оптимальное значение целевой функции с оптимальными решениями подзадач, т.е. как текущее оптимальное решение зависит от предыдущих оптимальных решений
3. Если для решения будет использоваться таблица:
   - Чем определяются строки таблицы (переменная, объекты и т.п.)
   - Чем определяются столбцы таблицы
   - Какое значение в ячейках таблицы (в основном определяется п.2)
4. Задать начальные значения
5. Находить оптимальные решения подзадач, в итоге вычислить оптимальное значение целевой функции (восходящее ДП).
6. Пользуясь полученной информацией, построить (восстановить) оптимальное решение

Одномерная и многомерная динамика

Пример с числами Фибоначчи – пример одномерной динамики.

Многомерная динамика отличается от одномерной количеством измерений, т.е. количеством параметров в состоянии.

Рассмотрим еще примеры задач на одномерную динамику

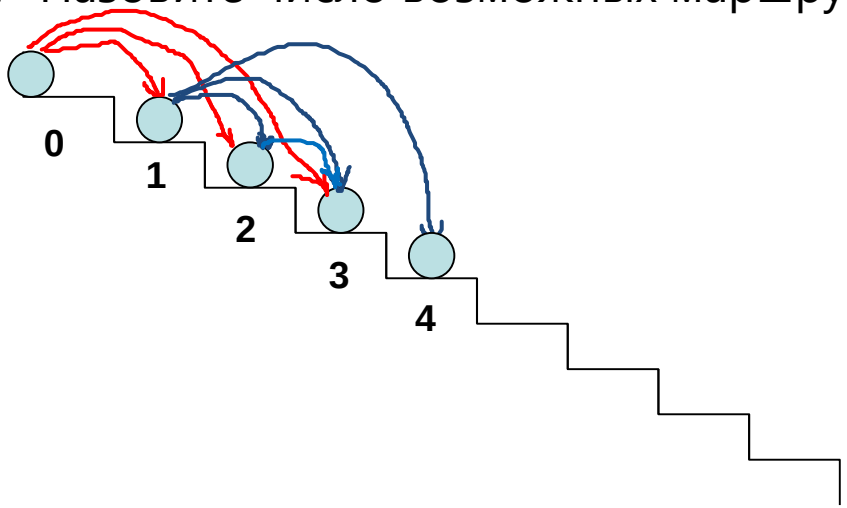
1. Задача «Лесенка и мячик»

На вершине лесенки, содержащей K ступенек, находится мячик, который начинает прыгать по ним вниз, к основанию. Мячик может прыгнуть на следующую ступеньку, на ступеньку через одну или через 2.

(Если пронумеровать ступеньки сверху вниз, то если мячик лежит на нулевой ступеньке, то он может переместиться на первую, вторую или третью.) Определить число всевозможных «маршрутов» мячика с вершины на землю.

Задание

1. Напишите рекуррентную формулу для количества маршрутов.
2. Назовите число возможных маршрутов для K=8.



1. Задача «Лесенка и мячик». Решение

На первую ступеньку можно попасть только одним образом — сделав прыжок с длиной равной единице. Всего 1 вариант.

На вторую ступеньку можно попасть сделав прыжок длиной 2, или с первой ступеньки — всего 2 варианта.

На третью ступеньку можно попасть сделав прыжок длиной три, с первой или со второй ступенек. Т.е. всего 4 варианта (0->3; 0->1->3; 0->2->3; 0->1->2->3).

Теперь рассмотрим четвёртую ступеньку. На неё можно попасть с первой ступеньки — по одному маршруту на каждый маршрут до неё, со второй или с третьей — аналогично. Иными словами, количество путей до 4-й ступеньки есть сумма маршрутов до 1-й, 2-й и 3-й ступенек.

Рекуррентная формула:

$$F(N) = F(N-1)+F(N-2)+F(N-3), \quad N = 4 \ldots K$$

Первые три ступеньки будем считать начальными состояниями.

$$F[1] = 1; \quad F[2] = 2; \quad F[3] = 4;$$

Номер ступеньки	1	2	3	4	5	6	7	8
Количество маршрутов	1	2	4	7	13	24	44	81

2. Задача Калькулятор

Имеется калькулятор, который выполняет три операции:

1. Прибавить к числу X единицу;
2. Умножить число X на 2;
3. Умножить число X на 3.

Определите, какое наименьшее число операций необходимо для того, чтобы получить из числа 1 заданное число N.

Выведите это число, и на следующей строке набор исполненных операций вида «111231».

2. Задача Калькулятор

Наивное решение состоит в том, чтобы

делить число на 3, пока это возможно,

иначе на 2, если это возможно,

иначе вычитать единицу,

до тех пор, пока оно не обратится в единицу.

Это неверное решение, т.к. оно исключает, например, возможность убавить число на единицу, а затем разделить на три, из-за чего на больших числах (например, 32718) возникают ошибки.

2. Задача Калькулятор

Правильное решение заключается в нахождении для каждого числа от 2 до N минимального количества действий на основе предыдущих элементов, иначе говоря:

$$F(N) = \min(F(N - 1), F(N/2), F(N/3)) + 1.$$

ВАЖНО! все индексы должны быть целыми.

Для формирования списка действий необходимо идти в обратном направлении и искать такой индекс i, что F(i) = F(N), где N — номер рассматриваемого элемента.

Если i = N - 1, записываем в начало строки 1,

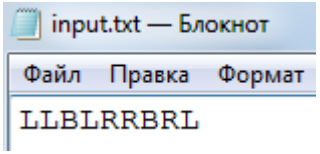
если i = N/2 , записываем двойку,

иначе — тройку.

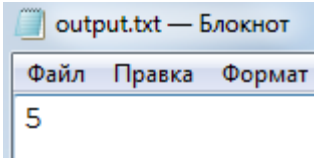
Задача Поход

Группа школьников решила сходить в поход вдоль Москвы-реки. У Москвы-реки существует множество притоков, которые могут впадать в нее как с правого, так и с левого берега. Школьники хотят начать поход в некоторой точке на левом берегу и закончить поход в некоторой точке на правом берегу, возможно, переправляясь через реки несколько раз. Как известно, переправа как через реку, так и через приток представляет собой определенную сложность, поэтому они хотят минимизировать число совершенных переправ. Школьники заранее изучили карту и записали, в какой последовательности в Москву-реку впадают притоки на всем их маршруте. Помогите школьникам по данному описанию притоков определить минимальное количество переправ, которое им придется совершить во время похода.

Вход (input_bridge.txt)



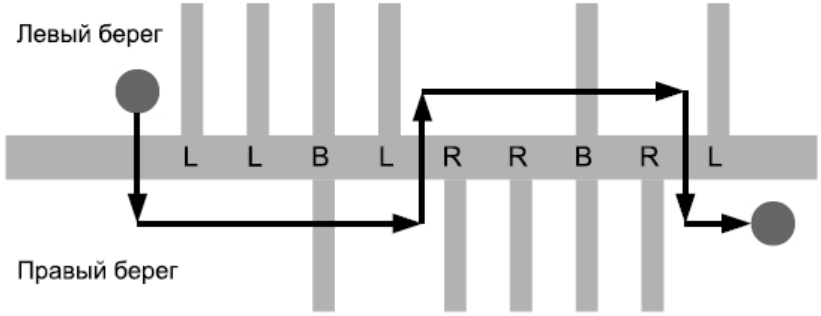
Результат (output_bridge.txt)



Пример

Вход	Ответ
LLBLRRBRL	5

Рисунок к приведенному выше примеру.

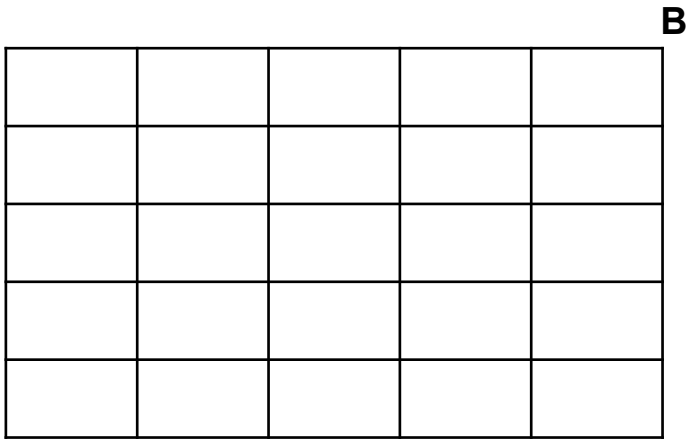


3. Задача Черепашка

Черепашке надо попасть из пункта A в пункт B. Шаг можно выполнять только на север или на восток. Длины пути (время в пути) между пунктами (узлами) заданы

Найти кратчайший путь (минимальное время).

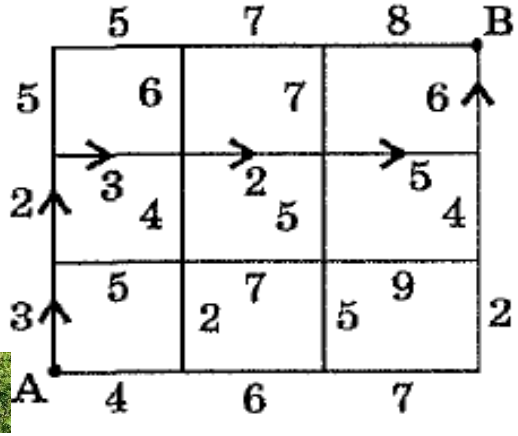
A



Черепашка

XXXXXX CCC
CCCXXX
CCXCXX
CCXXCX
CCXXXC и т.д.

Путь – 6 шагов, по 3 на север и восток
Количество путей – число сочетаний из 6 по 3, т.е. 20



21 ед. времени

Для вычисления времени по одному из 20 вариантов требуется выполнить

- 5 операций сложения
- 1 операцию сравнения

Для вычисления времени по всем 20 вариантам требуется выполнить

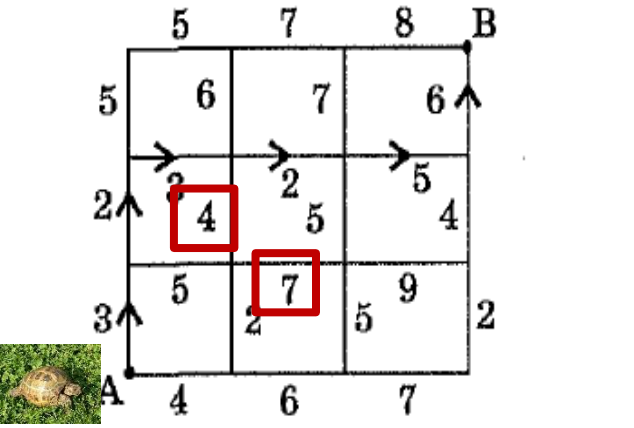
- 100 операций сложения
- 19 операций сравнения

При N=8 количество вариантов 12870
Для вычисления времени по одному из 12870 вариантов требуется выполнить
15 операций сложения

Для вычисления времени по всем 12870 вариантам требуется выполнить
193050 операций сложения
12869 операций сравнения

А если N=30 ?

Черепашка

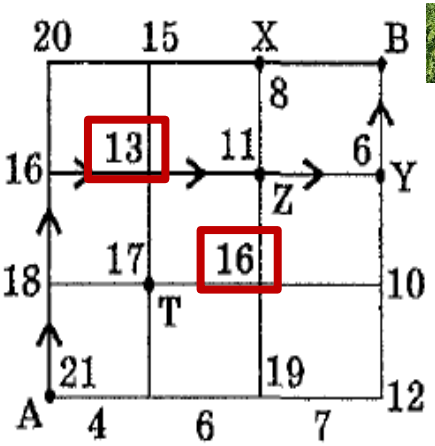


Начнем строить путь от пункта B

Каждому узлу присвоим вес, равный минимальному времени движения Черепашки от этого пункта до пункта B

X = 8; Y = 6; Z= min(8+7, 6+5) = min(15, 11) = 11

Еще пример:
T= min(13+4, 16+7) = min(17, 23) = 17



Количество операций для каждого узла:

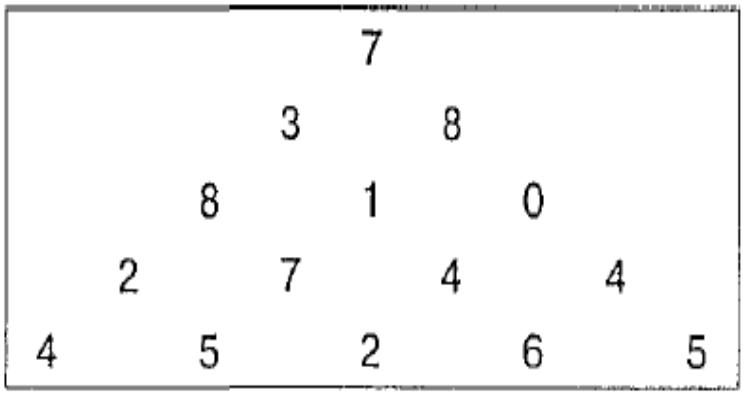
2 сложения + 1 сравнение = 3 операции.

Всего операций 2*3*N

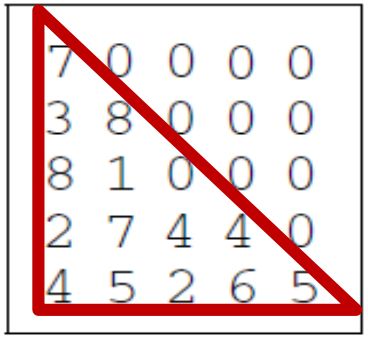
Восстанавливаем маршрут, выбирая лучший путь из A в B (стрелки), т.е. перемещаемся в том направлении, где в узле меньшее число (расстояние от узла до B)

4. Задача Треугольник

Дано: числовой треугольник из N строк
Требуется: Вычислить наибольшую сумму чисел, расположенных на пути, начинающемся в верхней точке треугольника и заканчивающемся на основании треугольника



Матрица D



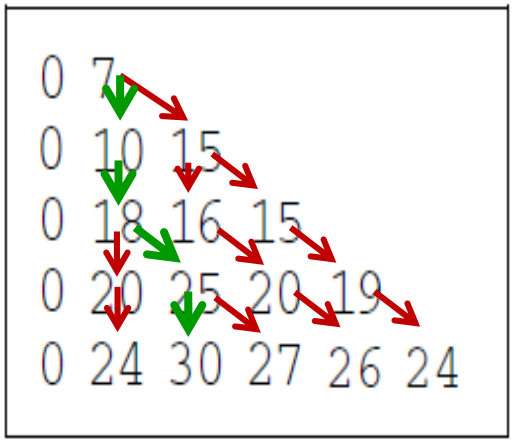
Ограничения и правила
1) Каждый шаг на пути – вниз влево или вниз вправо
2) Число строк от 1 до 100
3) Треугольник составлен из целых чисел от 0 до 99

Треугольник

Матрица D

7	0	0	0	0
3	8	0	0	0
8	1	0	0	0
2	7	4	4	0
4	5	2	6	5

Матрица R



Выбрать мах в последней строке.
Для восстановления пути выбирать максимальное значение из двух с предыдущей строки (выше и выше+левее)

5. Задача Наибольшая общая подпоследовательность строк

Подпоследовательность - это список элементов, в которых играет роль их порядок. Определенный элемент может появляться в подпоследовательности несколько раз.

Подпоследовательностью Z строки X является строка X, возможно, с удаленными элементами. Например, если X является строкой ABC, то она имеет 8 подпоследовательностей:

ABC, AB, AC, BC, A, B, C и пустая строка.

Если X и Y являются строками, то Z является общей подпоследовательностью X и Y, если она является подпоследовательностью обеих строк.

Понятия "подпоследовательность" и "подстрока" не эквивалентны. Подстрока - это подпоследовательность, в которой все символы выбраны из смежных позиций в строке.

Пример: строка BCDACDBACDB

подстрока CDACD

подпоследовательность CDCDACD

Постановка задачи

Задача: Для двух заданных строк X и Y найти все наидлиннейшие общие подпоследовательности Z[1..k] строк X и Y, где k – длина наидлиннейшей общей подпоследовательности.

Будем решать задачу поэтапно.

Сначала определим длину наидлиннейших подпоследовательностей k.

Затем попытаемся восстановить одну подпоследовательность.

И наконец, найдем все наидлиннейшие подпоследовательности.

Длина подпоследовательности

Для начала выясним, что представляет собой подзадача.

Будем использовать префиксы строк X[1..m] и Y[1..n], обозначим их

$$X_i = x_1 x_2 x_3 \ldots x_i, \quad i = 0..m$$

$$Y_j = y_1 y_2 y_3 \ldots y_j, \quad j = 0..n$$

X_0 и Y_0 - пустые строки.

Наидлиннейшая общая подпоследовательность двух строк содержит наидлиннейшие общие подпоследовательности префиксов этих строк.

Рассмотрим префиксы X_i и Y_j. Обозначим длину наидлиннейшей общей подпоследовательности префиксов как $l[i, j]$.

(Длина наидлиннейшей общей подпоследовательности строк X и Y равна $l[m, n]$)

- Если один из префиксов равен нулю, то наидлиннейшая общая подпоследовательность – пустая строка

$$l[0, j] = l[i, 0] = 0, \quad i = 0..m; \quad j = 0..n$$

Длина подпоследовательности

- Если i>0 и j>0, а $x_i = y_j$, то символ x_i (или y_j) добавляется к наидлиннейшей подпоследовательности, т.е. она увеличивается на один символ.

  Далее переходим к рассмотрению префиксов X_{i-1} и Y_{j-1}

$$l[i, j] = l[i - 1, j - 1] + 1$$

Пример:

Префикс X_i	xxxxxxxxxD	xxxxxD
Префикс Y_j	yyyyD	yyyyyyyyyyyyD

Символ D войдет в подпоследовательность наибольшей длины, далее перейдем к сравнению префиксов X_{i-1} и Y_{j-1}

Префикс X_{i-1}	xxxxxxxxx	xxxxx
Префикс Y_{j-1}	yyyy	yyyyyyyyyyyy

Длина подпоследовательности

- Если i>0 и j>0, а $x_i \neq y_j$, то надо рассматривать одну из двух пар префиксов: X_{i-1} и Y_j или X_i и Y_{j-1}. Тогда

$$l[i,j] = \max (l [i - 1, j] , l [i, j - 1])$$

Пример:

Префикс X_i	xxxxxxxxxA	xxxxxA
Префикс Y_j	yyyyD	yyyyyyyyyyyyD

Перейдем к сравнению префиксов X_{i-1} и Y_j или X_i и Y_{j-1}.

Префикс X_{i-1}	xxxxxxxxx	xxxxx
Префикс Y_j	yyyyD	yyyyyyyyyyyyD
или		
Префикс X_i	xxxxxxxxxA	xxxxxA
Префикс Y_{j-1}	yyyy	yyyyyyyyyyyy

Длина подпоследовательности

$$l[i,j] = \begin{cases} 0, & \text{при } i = 0 \text{ или } j = 0 \\ l[i-1, j-1] + 1 & \text{при } x_i = y_j, i > 0, j > 0 \\ max(l[i-1, j], l[i, j-1]) & \text{при } x_i \neq y_j, i > 0, j > 0 \end{cases}$$

Эти значения удобно сохранить в двумерной таблице.

		j	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
i	x_i	y_j		G	T	A	C	C	G	T	C	A
0			0	0	0	0	0	0	0	0	0	0
1	C		0	0	0	0	1	1	1	1	1	1
2	A		0	0	0	1	1	1	1	1	1	2
3	T		0	0	1	1	1	1	1	2	2	2
4	C		0	0	1	1	2	2	2	2	3	3
5	G		0	1	1	1	2	2	3	3	3	3
6	A		0	1	1	2	2	2	3	3	3	4

Сама подпоследовательность

Будем собирать наидлиннейшую подпоследовательность рекурсивно по известной таблице длин подпоследовательностей

Потребуется m+n рекурсивных вызовов.

```
Assemble_LCS(X,Y,l,i,j)
//Вход – строки X,Y, массив l, индексы i,j строк X,Y и массива l
01 Если l[i,j] = 0 вернуть пустую строку
02   иначе
03     Если x_i = y_j  вернуть Assemble_LCS(X,Y,l,i-1,j-1) + x_i
04        иначе
05            Если l[i-1,j] < l[i,j-1] вернуть Assemble_LCS(X,Y,l,i,j-1)
06              иначе вернуть Assemble_LCS(X,Y,l,i-1,j)
```

Как получить все варианты наидлиннейших последовательностей?

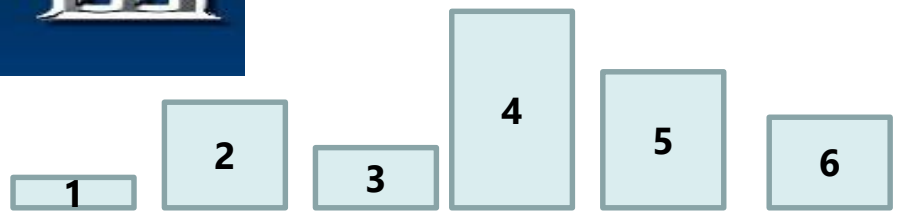
6. ЗАДАЧА О КАМНЯХ

Из камней весом p_1, p_2,..., p_N, набрать вес W или, если это невозможно, максимально близкий к W снизу вес.

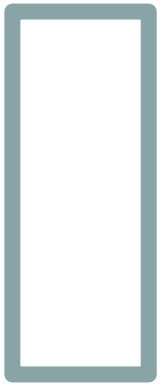
Подходы к решению

1) Перебор всех вариантов
2) Перебор с возвратом
3) **Динамическое программирование**

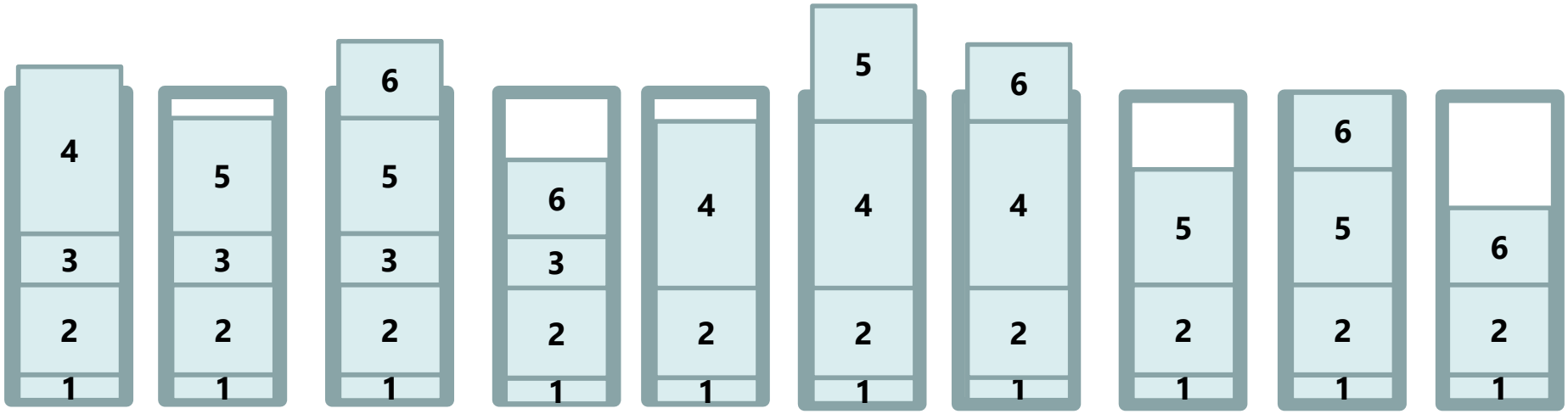
Перебор с возвратом

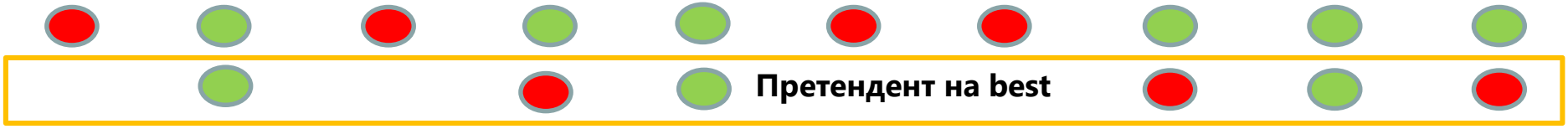


Размер = вес



Необходимый вес





допустимо

Не допустимо / не best

И т.д. – останавливаться нельзя!!!

Метод ДП

Для определения максимально возможного веса (<=W) набора камней можно использовать метод динамического программирования и построить таблицу.

Строим таблицу `A[0..N, 0..W]`, в которой

- номера строк – это номера камней
- Номера столбцов – максимально возможный вес набора из доступных камней
- Элемент `A[i,j]` – решение задачи о камнях при `N=i, W=j`.

В нулевой столбец и нулевую строку запишем нули.

Заполняем таблицу по строкам.

- В строке 1 – набираем вес, имея только первый камень.
- В строке 2 – набираем вес, имея только первый и второй камень.
- В строке `i` – набираем вес, имея только камни с первого по `i`-тый.

Пример таблицы ДП

Дано N = 5, W = 19, P = {5,7,9,11,13}

(можно не использовать нулевую строку, а предварительно заполнить первую строку. До W<P[1] нулями, далее – значением P[i])

Pi	*i* / *j*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
0	*0*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	*1*	0	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5
7	*2*	0	0	0	0	0	5	5	7	7	7	7	12	12	12	12	12	12	12	12
9	*3*	0	0	0	0	0	5	5	7	7	9	9	9	12	12	14	14	16	16	16	16
11	*4*	0	0	0	0	0	5	5	7	7	9	9	11	12	12	14	14	16	16	18	18
13	*5*	0	0	0	0	0	5	5	7	7	9	9	11	12	13	14	14	16	16	18	18

Какие камни входят в набор

Для определения набора камней можно использовать рекурсию.
Предполагается, что таблица хранится глобально

Pi
0
5
7
9
11
13

i	*j*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
2	0	0	0	0	0	5	5	7	7	7	7	7	12	12	12	12	12	12	12	12
3	0	0	0	0	0	5	5	7	7	9	9	9	12	12	14	14	16	16	16	16
4	0	0	0	0	0	5	5	7	7	9	9	11	12	12	14	14	16	16	18	18
5	0	0	0	0	0	5	5	7	7	9	9	11	12	13	14	14	16	16	18	18

Контест / работа на семинаре

В контесте 6 задач

1) Поход
2) Задача о камнях
3) Поиск длины максимальной подпоследовательности двух строк
4) Поиск максимальной подпоследовательности двух строк
5) Интернет
6) Экзамены