Динамическое программирование

Горденко Мария Константиновна

Динамическое программирование

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. («простая» - это, например, с меньшей длиной входа)
- **Динамическое программирование** обычно применяется к задачам, в которых искомый ответ состоит из частей, каждая из которых в свою очередь дает оптимальное решение некоторой подзадачи, например, сумма решений подзадач.
- **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.

Richard Ernest Bellman; 1920 — 1984

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 задач, «предшествующих» 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.

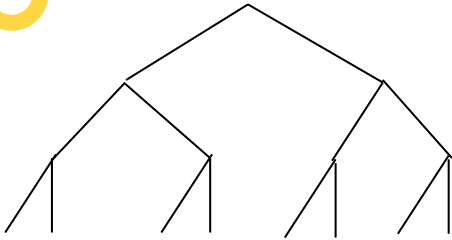
Об оптимизации

- 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 некоторой 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).

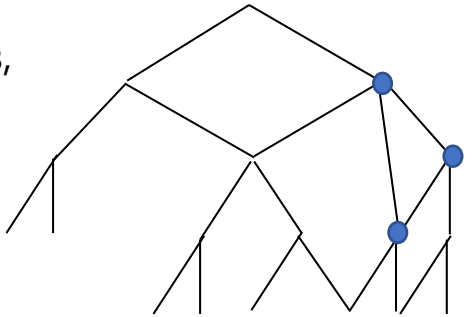
Еще об оптимизации

- **Оптимизация** — процесс максимизации целевой функции (выгодных характеристик, соотношений, например, оптимизация производственных процессов и производства), и минимизации расходов.
- Задача оптимизации сформулирована, если заданы:
  - целевая функция, т.е. критерий оптимальности (экономический – максимальная прибыль; 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 целевую функцию и тем самым эффективность процесса;
  - ограничения, связанные с экономическими и конструктивными условиями, возможностями аппаратуры, требованиями взрывобезопасности и др.;
  - математическая модель процесса.

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:
  - пересекающиеся подзадачи;
  - оптимальная подзадача;
  - возможность запоминания решений 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:

$F_0 = 1, F_1 = 1,$

$F_n = F_{n-1} + F_{n-2}$ при $n > 1$

Необходимо найти F_n по номеру n.

- Логичный способ решения – с помощью рекурсии:
- Некоторые вызовы происходят не один раз, т.е. рекурсия для подзадачи запускается несколько раз

$F_7 = F_6 + F_5$; $F_6 = F_5 + F_4$;

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

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

Пример: простая задача о последовательностях (продолжение)

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).

Проблема – иногда вычисляются «ненужные» значения.

Подходы к решению задач методом ДП

Обычно используются два подхода к решению задач ДП:

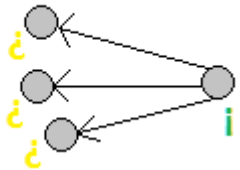
- **нисходящее** динамическое программирование: задача разбивается на подзадачи меньшего размера, они решаются и затем комбинируются для решения исходной задачи (рекурсия). Полученные решения подзадач можно сохранить и затем использовать, чтобы не вычислять их несколько раз
- **восходящее** динамическое программирование: все подзадачи, которые впоследствии понадобятся для решения исходной задачи, просчитываются заранее и затем используются для построения решения исходной задачи (итерация). 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:

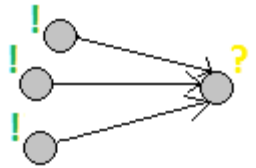
1) Обратный порядок:

   Обновляются все состояния, от которых зависит текущее состояние.



2) Прямой порядок:

   Состояния последовательно пересчитывается исходя из уже посчитанных.



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 (оно было вычислено при предыдущем рекурсивном вызове функции). Иначе – выполняем рекурсивный вызов. Об этом – на следующем семинаре

Алгоритм ДП

1. Описать оптимальные решения, т.е. определить целевую функцию, которую надо минимизировать / максимизировать /.
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, надо решить:
   - 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

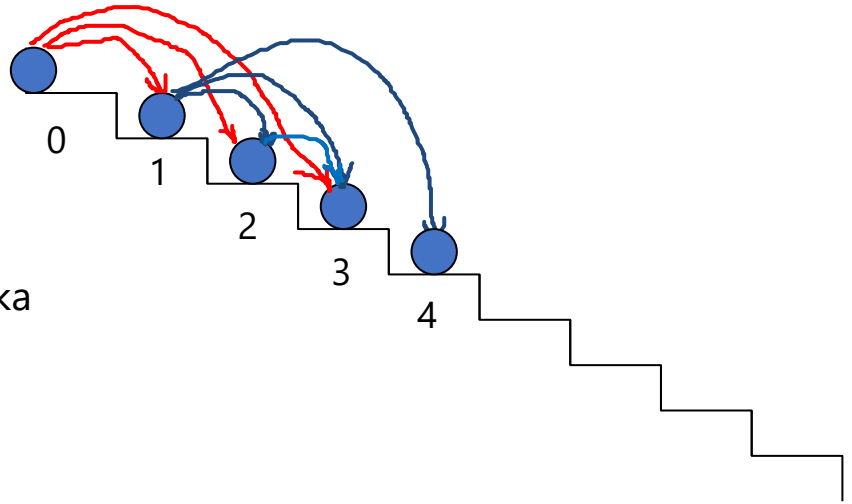
Одномерная и многомерная динамика

- Пример с числами Фибоначчи – пример одномерной динамики.
- Многомерная динамика отличается от одномерной количеством измерений, т.е. количеством изменяемых параметров в целевой функции.
- 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 возможных 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: F(4) = F(3)+F(2)+F(1),

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, пока это возможно,
  - иначе вычитать единицу, до тех пор, пока не получим 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.
- Для 82 по наивному решению получим: 82/2=41, затем 40 вычитаний. Лучше было бы 82-1=81, 81/3 = 9, 9/3=3, 3/3=1. – четыре операции.

2. Задача Калькулятор

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

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

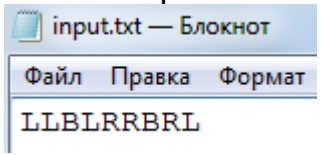
- ВАЖНО! все индексы должны быть целыми.
- Для формирования списка действий необходимо идти в обратном направлении и искать такой индекс i, что F(i) = F(N), где N — номер рассматриваемого элемента.
- Если i = N - 1, записываем в начало строки 1,

  если i = N/2 , записываем двойку,

  иначе — тройку.

Задача Поход (контест)

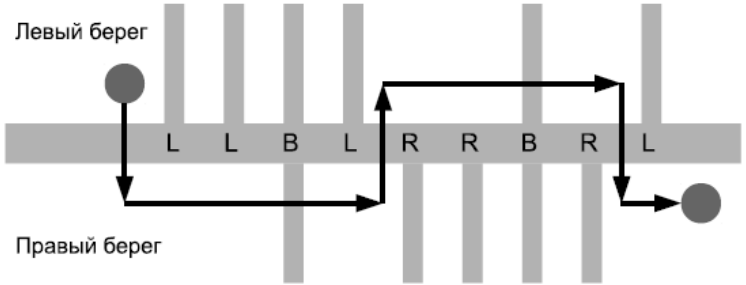
- Группа школьников решила сходить в поход вдоль Москвы-реки. У Москвы-реки существует множество притоков, которые могут впадать в нее как с правого, так и с левого берега. Школьники хотят начать поход в некоторой точке на левом берегу и закончить поход в некоторой точке на правом берегу, возможно, переправляясь 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)



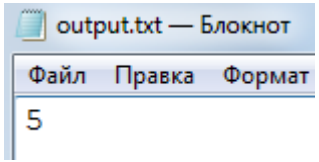
Пример

Вход	Ответ
LLBLRRBRL	5

Рисунок к приведенному выше примеру.



- Результат (output_bridge.txt)



Задача Поход (контест)

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).

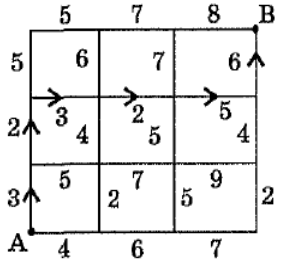
A

B

Черепашка

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

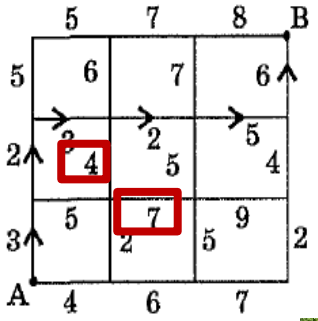
XXXXXX CCC
CCCXXX
CCXCXX
CCXXCX
CCXXXC и т.д.

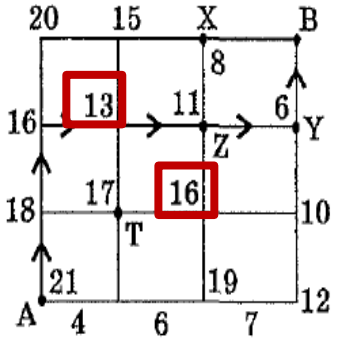


21 ед. времени

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

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

Еще пример:

$$T = \min(13+4, 16+7) = \min(17, 23) = 17$$

Количество операций для каждого узла:

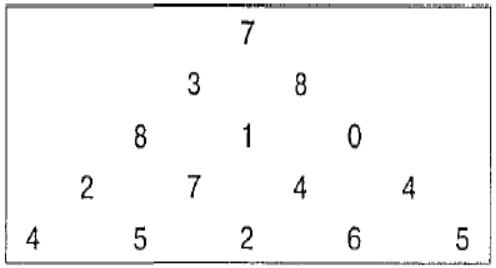
2 сложения + 1 сравнение = 3 операции.

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)

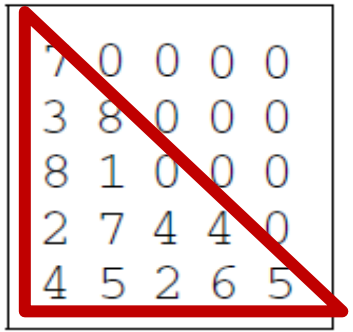
4. Задача Треугольник

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



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

Матрица D



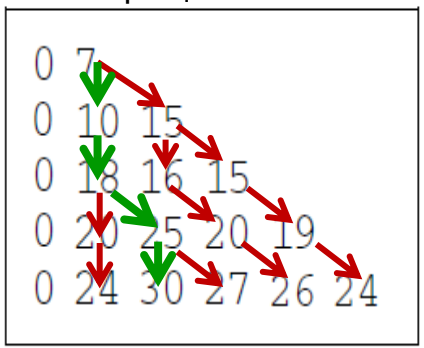
Треугольник

- Построим матрицу R – наибольшие суммы в каждом узле
- Выбираем мах в последней строке.
- Для восстановления пути выбираем максимальное значение из двух с предыдущей строки (выше и выше+левее)

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



Треугольник

```
R[1,1] = D[1,1]
for i = 2 to N do
  for j = 1 to i do
     R[i,j] = max(D[i,j] + R[i-1,j],
                  D[i,j] + R[i-1,j-1])
// R[i,j]=max(R[i-1,j], R[i-1,j-1]) + D[i,j]
```

```
R[2,1] = max(D[2,1] + R[1,1], D[2,1] + R[1,0]) =
             max(3+7, 3+0) = 10
```

```
R[2,2] = max(D[2,2] + R[1,2], D[2,2] + R[1,1]) =
             max(8+0, 8+7) = 15
```

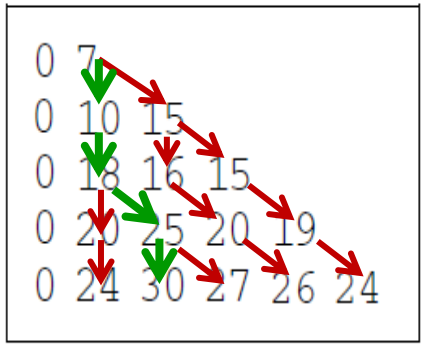
```
R[5,3] = max(D[5,3] + R[4,3], D[5,3] + R[4,2]) =
             max(2+20, 2+25) = 27
```

Выбрать max в последней строке

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



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

Длины подпоследовательностей

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

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

Восстановление подпоследовательности

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

Где разветвление решений?

При выполнении каких условий?

Начинаем с правого нижнего угла, используем формулу (когда куда можно перемещаться по таблице). CTCA, TCGA, ACGA,…

1. Если символы в строке и столбце совпадают, включаем их в подпоследовательность, перемещаемся по диагонали.
2. Если символы в строке и столбце не совпадают, переходим вверх [i-1, j] или влево [i, j-1] в направлении максимального изменения значения в этих соседних точках.

		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

Восстановление подпоследовательности

Будем собирать наидлиннейшую подпоследовательность **рекурсивно** по известной таблице длин подпоследовательностей с использованием формулы для $l[i,j]$

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?

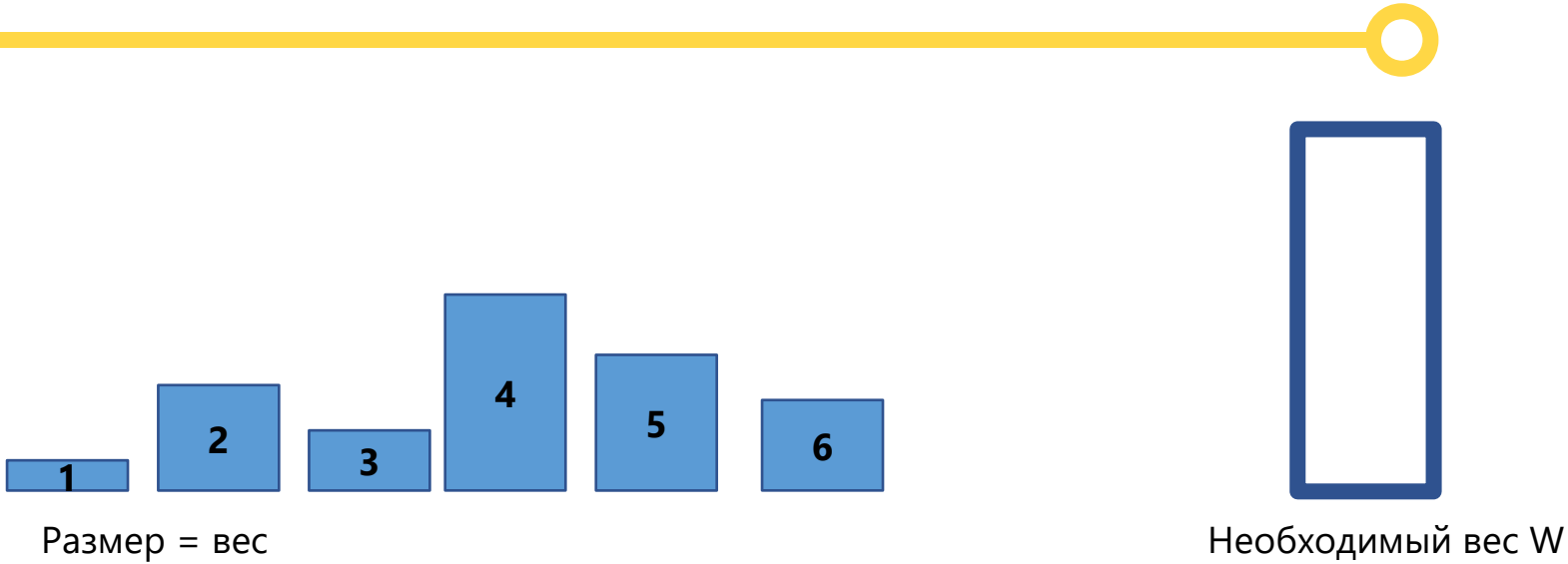
6. Задача о камнях

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

- Перебор всех вариантов
- Перебор с возвратом
- **Динамическое программирование восходящее (построение таблицы)**
- **Динамическое программирование нисходящее (Ленивая рекурсия)**
- **Жадный алгоритм**

Перебор с возвратом



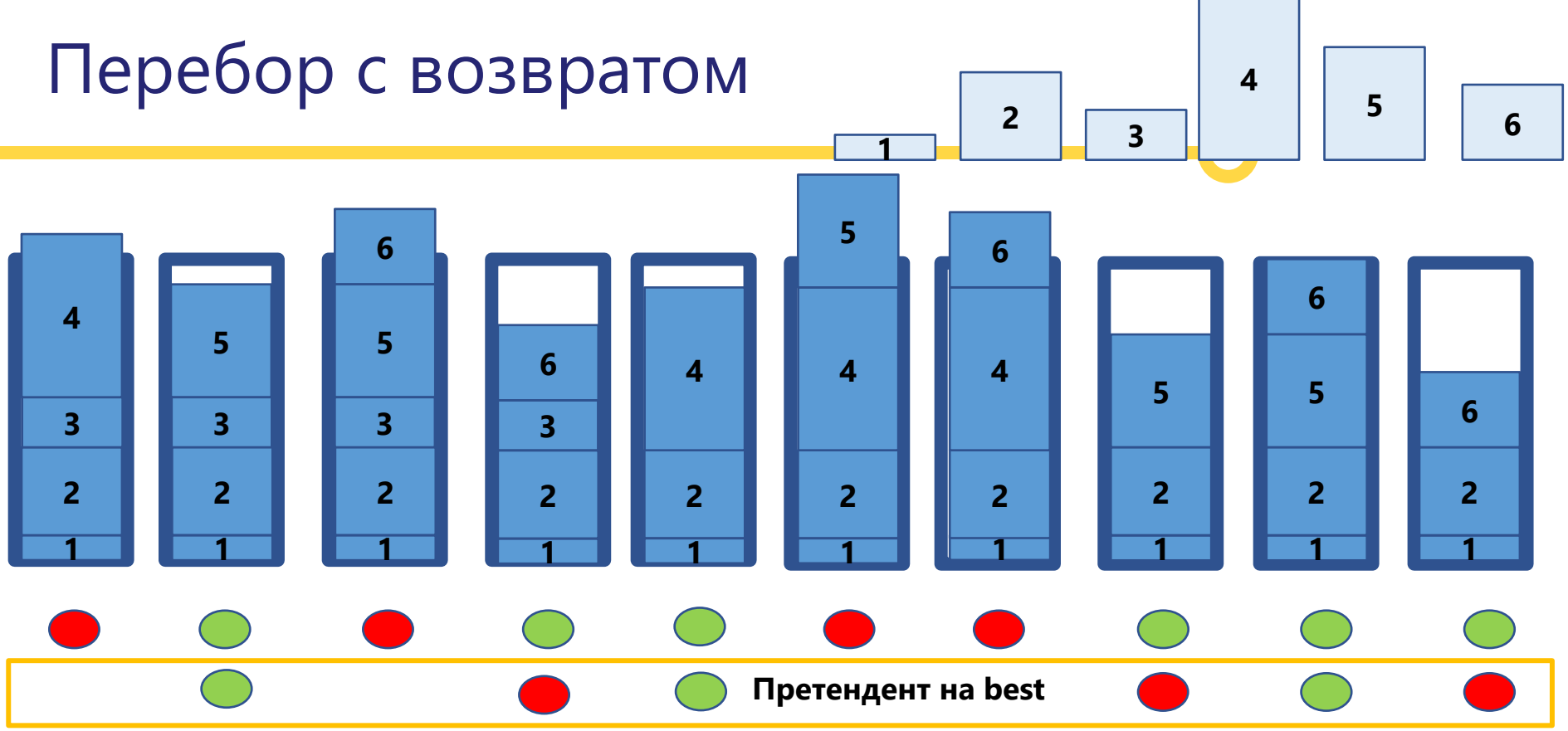
Размер = вес

Необходимый вес W

Надо собрать набор камней весом как можно ближе к W, но не больше W

Вариации – каждый предмет можно выбирать **только 1 раз** / произвольное число раз

Перебор с возвратом



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.

Строим таблицу ДП

- При построении таблицы и заполнении `A[i,j], i=1..N, j=1..W` выбираем максимальное значение из двух:
  - Тот же вес и предыдущий набор камней `A[i-1, j]`
  - Удалим из предыдущего набора `P[i]` килограмм (т. е. в `(i-1)`-й строке сдвинемся влево на `P[i]` столбцов, в столбец `j-P[i]`) и добавим к набору `A[i-1, j-P[i]]` `i`-й камень весом `P[i]`

```
Процедура Stones(P[1..N],W)
00  для i от 0 до N A[i,0] ← 0
01  для j от 1 до W A[0,j] ← 0
02  для i от 1 до N
03     для j от 1 до W
04            если j – P[i] >=0
05           то A[i,j] ← max(A[i-1,j], //оставим тот же набор камней
                          A[i-1,j-P[i]]+P[i]) //или заменим несколько камней на i-й
06           иначе A[i,j] ← A[i-1,j]
07  вернуть A[N,W]   // таблица ДП
```

Строим таблицу ДП

При построении таблицы и заполнении `A[i,j], i=1..N, j=1..W` выбираем максимальное значение из двух:

- Тот же вес и предыдущий набор камней `A[i-1, j]`
- Удалим из предыдущего набора `P[i]` килограмм (т. е. в `(i-1)`-й строке сдвинемся влево на `P[i]` столбцов, в столбец `j-P[i]`) и добавим к набору `A[i-1, j-P[i]]` `i`-й камень весом `P[i]`

$A_{i,0}=0, \quad 0 \le i \le N$

$A_{0,j}=0, \quad 0 \le j \le W$

$A_{i,j} = A_{i-1,j}, \quad P_i > j,\ 0 < j \le W$

$A_{i,j} = \max(A_{i-1,j}, A_{i-1,j-P[i]} + P_i), \quad P_i \le j,\ 0 < j \le W$

Пример таблицы ДП

Дано N = 5, W = 19, P = {5,7,9,11,13}

$A_{i,0} = 0$, $0 \le i \le N$

$A_{0,j} = 0$, $0 \le j \le W$

$A_{i,j} = A_{i-1,j}$, $P_i > j,\ 0 < j <= W$

$A_{i,j} = \max(A_{i-1,j},\ A_{i-1,j-P[i]} + P_i)$, $P_i \le j,\ 0 < j <= W$

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

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
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

```
02  для i от 1 до N
03   для j от 1 до W
04    если j – P[i] >=0
05     то A[i,j] ← max(A[i-1,j],A[i-1,j-P[i]]+P[i])
06     иначе A[i,j] ← A[i-1,j]
```

A[2,7]=max(A[1,7],A[1,7-7]+7)=max(5,0+7) = 7

A[2,12]=max(A[1,12],A[1,12-7]+7)=max(5,5+7) = 12

A[3,18]=max(A[2,18],A[2,18-11]+11)=max(16,7+11) = 18

A[5,19]=max(A[4,19],A[4,19-13]+13)=max(18,5+13) = 18

Какие камни входят в набор

Для определения набора камней можно использовать рекурсию, т.к. наборов может быть несколько.

Предполагается, что таблица хранится глобально

```
Процедура Way(i,j)
00  если A[i,j] = 0 выход
01  если A[i,j] ≠ A[i-1,j]
02    то {Way(i-1,j-P[i]); вывод P[i]}
03    иначе Way(i-1,j)
06  всё
```

Где разветвление решения?

При выполнении каких условий?

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

Что же такое Динамическое программирование?

- При динамическом программировании хранятся результаты вычислений, чтобы вычисления (одни и те же вызовы рекурсивных функций, например) не повторять.
- В простом случае это вызовы функции "снизу вверх" - f(1), f(2), f(3) и т.д.
- В более продвинутом случае это рекурсия с сохранением результатов вычислений, чтобы строить не всю таблицу (N-мерную гипертаблицу), а только нужную её часть.

Что же такое Динамическое программирование?

Например, надо вычислить f(x) = x > 3 ? f(x-2) + f(x-4) : 1

1. **Решение рекурсией** для f(7):

   ```
   f(7) = f(5) + f(3) = f(3) + f(1) + f(3) = 1 + 1 + 1 = 3.
   f(3) вычисляется дважды.
   ```

2. **Решение динамическое, в лоб**:

   ```
   f(0) = 1
   f(1) = 1
   f(2) = 1
   f(3) = 1
   f(4) = f(2) + f(0) = 1 + 1 = 2
   f(5) = f(3) + f(1) = 1 + 1 = 2
   f(6) = f(4) + f(2) = 2 + 1 = 3
   f(7) = f(5) + f(3) = 2 + 1 = 3
   f(3) вычисляли один раз, но f(0), f(2), f(4), f(6) - лишние вычисления.
   ```

Что же такое Динамическое программирование?

Например, у вас есть f(x) = x > 3 ? f(x-2) + f(x-4) : 1

3. **Решение динамическое**, как рекурсия с сохранением результата:

```
f(7)
= f(5) + f(3)
= f(5) + 1 // сохранили f(3) = 1.
= f(3) + f(1) + 1 // f(3) не вычисляем, подставляем сохранённое значение
= 1 + f(1) + 1
= 1 + 1 + 1 = 3
```

ЗАДАЧА О РЮКЗАКЕ

Постановка задачи

- Имеются наборы
  - объектов $o_1, o_2, \ldots, o_N$
  - весов (объемов) объектов $w_1, w_2, \ldots, w_N$
  - стоимостей объектов $s_1, s_2, \ldots, s_N$.
- Необходимо упаковать рюкзак весом (объемом) W так, чтобы его стоимость S была максимальной.

Разновидности

- Каждый предмет из множества можно выбирать произвольное количество раз.
- ***Каждый предмет можно использовать только один раз***
- И др

Подходы к решению

- Полный перебор
- Перебор с возвратом
- ***Динамическое программирование***
- ***Жадный алгоритм***

1. Решение методом ДП. Строим таблицу ДП

- Таблица ДП:
  - Номер строки – номер предмета,
  - Номер столбца – вес рюкзака
  - Ячейка таблицы [i, j] – максимальная стоимость набора предметов из $o_1..o_i$ для рюкзака весом j
  - Нулевые строка и столбец заполнены нулями
  - Нижняя ячейка столбца дает максимальную стоимость для данного размера рюкзака при использовании всех предметов

- Обозначим предметы парой чисел $[w_i, s_i]$. Рассмотрим пример заполнения таблицы для 6 предметов, $W=10$.
- Заполним таблицу по строкам.
- Подзадачи:
  - Вес рюкзака с 1 до заданного веса j
  - Количество предметов с 1 до заданного количества предметов i
- В ячейке таблицы – оптимальная (максимальная) стоимость упаковки рюкзака веса j при использовании i предметов

Рекуррентное соотношение (однократный выбор предмета)

$S_{i,w}$ - максимальная стоимость, которую можно набрать при весе рюкзака w (вес предметов в рюкзаке не более w) из первых i предметов

Рекуррентное соотношение:

$$S_{i,0} = 0, \quad 0 \le i \le n$$

$$S_{0,w} = 0, \quad 0 \le w \le W$$

Дополнительные строка сверху, столбец слева

$$S_{i,w} = \begin{cases} \max(S_{i-1,w}, (S_{i-1,w-w_i} + s_i)), & w_i \le w \\ S_{i-1,w}, & w_i > w \end{cases}, \quad 0 \le w \le W$$

Очередной предмет не можем положить в рюкзак, даже если вынем все из него (т.е. вес очередного предмета больше веса, который может поднять рюкзак)

Выбираем максимум стоимости из двух вариантов:
1) Не добавляем очередной предмет, не меняем упаковку
2) К лучшей упаковке рюкзака весом $w - w_i$ добавляем очередной предмет весом w_i и добавляем к ее стоимости стоимость очередного предмета

Пример – 6 предметов

W(*емкость рюкзака*)		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Объекты предметы		0	0	0	0	0	0	0	0	0	0
[3,55]	0	0	0	55	55	55	55	55	55	55	55
[2,80]	0	0	80	80	80	135	135	135	135	135	135
[4,60]	0	0	80	80	80	135	140	140	140	195	195
[1,45]	0	45	**80**	125	125	**135**	180				
[3,**105**]	0	45	80	125	150	**185**	230				
[1,50]											

$$S_{5,5} = \max(S_{4,5}, (S_{4,5-3} + s_5))$$

$$(w_5 = 3) \le (w = 5)$$

$$S_{5,5} = \max(135, (80 + 105)) = 185$$

Пример – 6 предметов

	W	1	2	3	4	5	6	7	8	9	10
объекты		0	0	0	0	0	0	0	0	0	0
[3,55]	0	0	0	55	55	55	55	55	55	55	55
[2,80]	0	0	**80**	80	80	135	**135**	135	135	135	135
[4, 60]	0	0	80	80	80	135	**140**	140	140	195	195
[1,45]	0	45	**80**	125	125	**135**	180	185	185	195	240
[3, 105]	0	45	80	125	150	**185**	230	230	240	285	290
[1,50]	0	50	95	130	175	200	235	280	280	290	335

Пример – 6 предметов. Что положили в рюкзак

	W	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
объекты		0	0	0	0	0	0	0	0	0	0
[3,55]	0	0	0	55	55	55	55	55	55	55	55
[2,80]	0	0	**80**	80	80	135	**135**	135	135	135	135
[**4,60**]	0	0	80	80	80	135	**140**	140	140	195	195
[1,45]	0	45	**80**	125	125	**135**	180	185	185	195	240
[3,**105**]	0	45	80	125	150	**185**	230	230	240	285	290
[1,50]	0	50	95	130	175	200	235	280	280	290	335

2. Решение с помощью рекурсии

Не все значения надо заполнять.

- Для W(6,10) надо только W(5,10) и W(5,9)
- Для W(5,10) надо только W(4,10) и W(4,7)
- Для W(5,9) надо только W(4,9) и W(4,6)

НО! Использовать рекурсию напрямую нельзя, т.к. появляется большое число перекрывающихся задач (ячейки заполняются несколько раз) и время выполнения может стать экспоненциальным.

При использовании динамического программирования сверху (запоминания) получаем псевдополиномиальную сложность O(n * W)

Пример – 6 предметов, рекурсия сверху

W(*емкость рюкзака*)		1	2	3	4	5	6	7	8	9	10
Объекты предметы		0	0	0	0	0	0	0	0	0	0
[3,55]	0	0	0	55	55	55	55	55	55	55	55
[2,80]	0	0	80	80	80	135	135	135	135	135	135
[4,60]	0	0	80	80	80	135	140	140	140	195	195
[1,45]	0	45	80	125	125	135	180	185	185	195	240
[3,**105**]	0	45	80	125	150	185	230	230	240	285	290
[1,50]	0	50	95	130	175	200	235	280	280	290	335

Черные цифры – не нужно вычислять,
красные цифры – вычисляются повторно,
синие цифры вычисляются однократно

Ленивая динамика

Сначала заполним таблицу символами, которые точно не могут появиться при расчетах, (например, -1), с помощью которых станет понятно, вычислялась ячейка ранее или нет.

Перед вызовом рекурсии:

если (значение в ячейке равно -1)
{вызываем рекурсию и ответ храним в таблице}
иначе
{используем значение из ячейки (без вызова рекурсии)}.

W(вес рюкзака)		1	2	3	4	5	6	7	8	9	10
предметы		0	0	0	0	0	0	0	0	0	0
[3,55]	0	0	0	55	55	55	55	55	55	55	55
[2,80]	0	0	80	80	80	135	135	135	135	135	135
[4,60]	0	-1	-1	-1	-1	135	140	140	140	195	195
[1,45]	0	-1	-1	-1	-1	-1	180	185	-1	195	240
[3,**105**]	0	-1	-1	-1	-1	-1	-1	-1	-1	285	290
[1,50]	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	335

3. Приближенное решение, или жадный алгоритм

Простой жадный алгоритм, выбирающий наилучшее отношение стоимости к размеру.

1) Каждый объект представляется в виде пары [размер, стоимость].
2) Удельные стоимости = стоимость/размер.
3) Сортируем объекты по удельной стоимости по убыванию
4) Заполняем рюкзак последовательно элементами сортированного списка
5) Если объект не входит, отбрасываем его и переходим к следующему

Это – не оптимальный алгоритм

Пример жадной упаковки

Предметы те же

Номер	1	2	3	4	5	6
Вес	3	2	4	1	3	1
Стоимость	55	80	60	45	105	50
Удельная стоимость	18.3	40	15	45	35	50

Отсортированные по убыванию удельной стоимости

Номер	6	4	2	5	1	3
Вес	1	1	2	3	3	4
Стоимость	50	45	80	105	55	60
Удельная стоимость	50	45	40	35	18.3	15

Собираем рюкзак весом 10

Номер	6	4	2	5	1	3
Сумма Весов	1	2	4	7	10	4
Сумма Стоимостей	50	95	175	280	335	60

Еще пример жадной упаковки

Предметы

Номер	1	2	3	4	5	6
Вес	2	3	4	5	6	7
Стоимость	20	30	40	50	60	70
Удельная стоимость	10	10	10	10	10	10

Отсортированные по убыванию удельной стоимости

Номер	1	2	3	4	5	6
Вес	2	3	4	5	6	7
Стоимость	20	30	40	50	60	70
Удельная стоимость	10	10	10	10	10	10

Собираем рюкзак весом 25

Номер	1	2	3	4	5	6
Сумма Весов	2	5	9	14	20	7
Сумма Стоимостей	20	50	90	140	200	70

Пример 3 жадной упаковки

Предметы

Номер	1	2	3	4	5	6
Вес	2	3	4	7	6	5
Стоимость	20	30	40	70	60	50
Удельная стоимость	10	10	10	10	10	10

Отсортированные по убыванию удельной стоимости

Номер	1	2	3	4	5	6
Вес	2	3	4	7	6	5
Стоимость	20	30	40	70	60	50
Удельная стоимость	10	10	10	10	10	10

Собираем рюкзак весом 25

Номер	1	2	3	4	5	6
Сумма Весов	2	5	9	16	22	5
Сумма Стоимостей	20	50	90	160	220	50

Пример 4 жадной упаковки

Предметы

Номер	1	2	3	4	5	6
Вес	7	6	5	4	3	2
Стоимость	70	60	50	40	30	20
Удельная стоимость	10	10	10	10	10	10

Отсортированные по убыванию удельной стоимости

Номер	1	2	3	4	5	6
Вес	7	6	5	4	3	2
Стоимость	70	60	50	40	30	20
Удельная стоимость	10	10	10	10	10	10

Собираем рюкзак весом 25

Номер	1	2	3	4	5	6
Сумма Весов	7	13	18	22	25	2
Сумма Стоимостей	70	130	180	220	250	20

Пример 5 жадной упаковки

Предметы

Номер	1	2	3	4	5	6
Вес	7	6	5	4	3	2
Стоимость	70	60	50	40	30	20
Удельная стоимость	10	10	10	10	10	10

Отсортированные по убыванию удельной стоимости

Номер	1	2	3	4	5	6
Вес	7	6	5	4	3	2
Стоимость	70	60	50	40	30	20
Удельная стоимость	10	10	10	10	10	10

Собираем рюкзак весом 20

Номер	1	2	3	4	5	6
Сумма Весов	7	13	18	4	3	20
Сумма Стоимостей	70	130	180	40	30	200

ЗАДАЧА О ШАХМАТНОМ КОНЕ

Постановка задачи

Шахматный конь стоит в клетке `(1, 1)` на доске размера `N x M`. Требуется подсчитать количество способов добраться до клетки `(N, M)` передвигаясь четырьмя типами шагов:

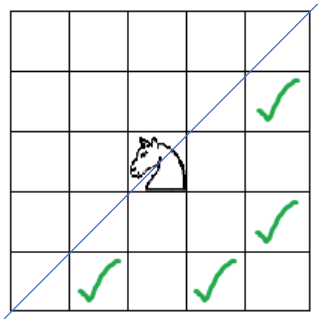


Таблица ДП

Строим таблицу `A[1..N, 1..M]`

1) Начальное значение в клетке `[1,1]` – 0 (количество ходов до клетки `[1,1]`)
2) Значение `A[i,j]` – количество способов добраться до клетки `(i, j)`
3) Для вычисления `A[i,j]` используем 4 клетки

```
A[i,j] = A[i-2,j-1] + A[i-2,j+1] + A[i-1,j-2] + A[i+1,j-2]
```

Проблема – при заполнении таблицы ДП не все 4 значения могут быть известны.

Решение

1) Выбрать пути заполнения таблицы ДП
2) Ленивая динамика
   - Заполним всю таблицу значениями -1
   - Запускаем рекурсию. Если рекурсивно вызывается значение в `A[i,j]`, не равное -1, используем значение `A[i,j]` без вызова рекурсии.
3) Ответ – в `A[N,M]`