Лекция 9

Динамическое программирование

План лекции

- Мемоизация и динамическое программирование
  - Рекурсивный алгоритм расчёта чисел Фибоначчи
  - Мемоизация
  - Динамическое программирование (общий принцип)
- Динамическое программирование (общий принцип)
  - Построение алгоритма ДП на основе рекурсивного
- Варианты ДП для задачи о рюкзаке
- Псевдополиномиальные алгоритмы

Числа Фибоначчи

- Задача: написать алгоритм / программу расчёта чисел Фибоначчи.
  - Вход: *n*.
  - Выход: *F*(*n*) – *n*-е число Фибоначчи.
- Числа Фибоначчи задаются рекуррентным соотношением:

$$F_0 = 0, \qquad F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}, \quad n \geqslant 2, \quad n \in \mathbb{Z}.$$

Числа Фибоначчи

Рекурсивный алгоритм F(n):

Если n=0, то вернуть 0.

Если n=1, то вернуть 1.

Иначе вернуть F(n-1)+F(n-2).

Оценим сложность алгоритма $T(n)$.

T(0) = T(1) = const

T(n) = T(n-1)+T(n-2)

Числа Фибоначчи

$T(n) \sim O(F(n))$.

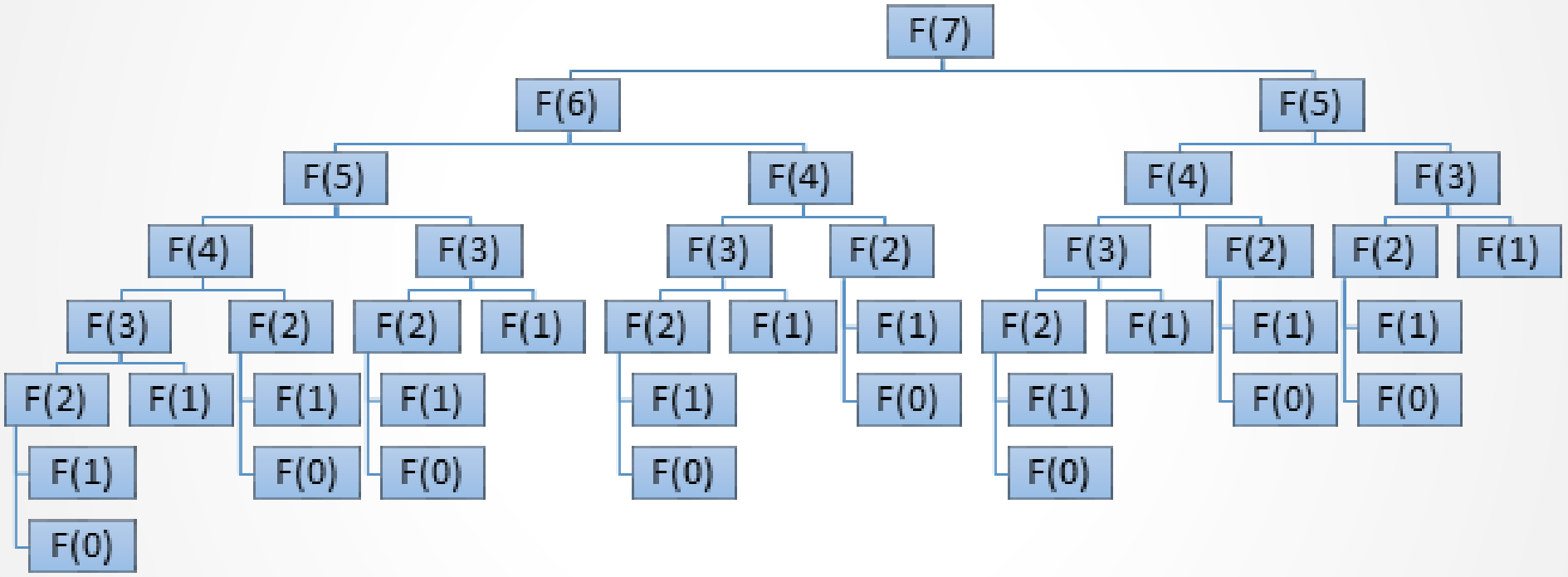
Но ведь $F_n \sim \frac{\varphi^n}{\sqrt{5}}$, где $\varphi = \frac{1+\sqrt{5}}{2}$

- коэффициент золотого сечения.

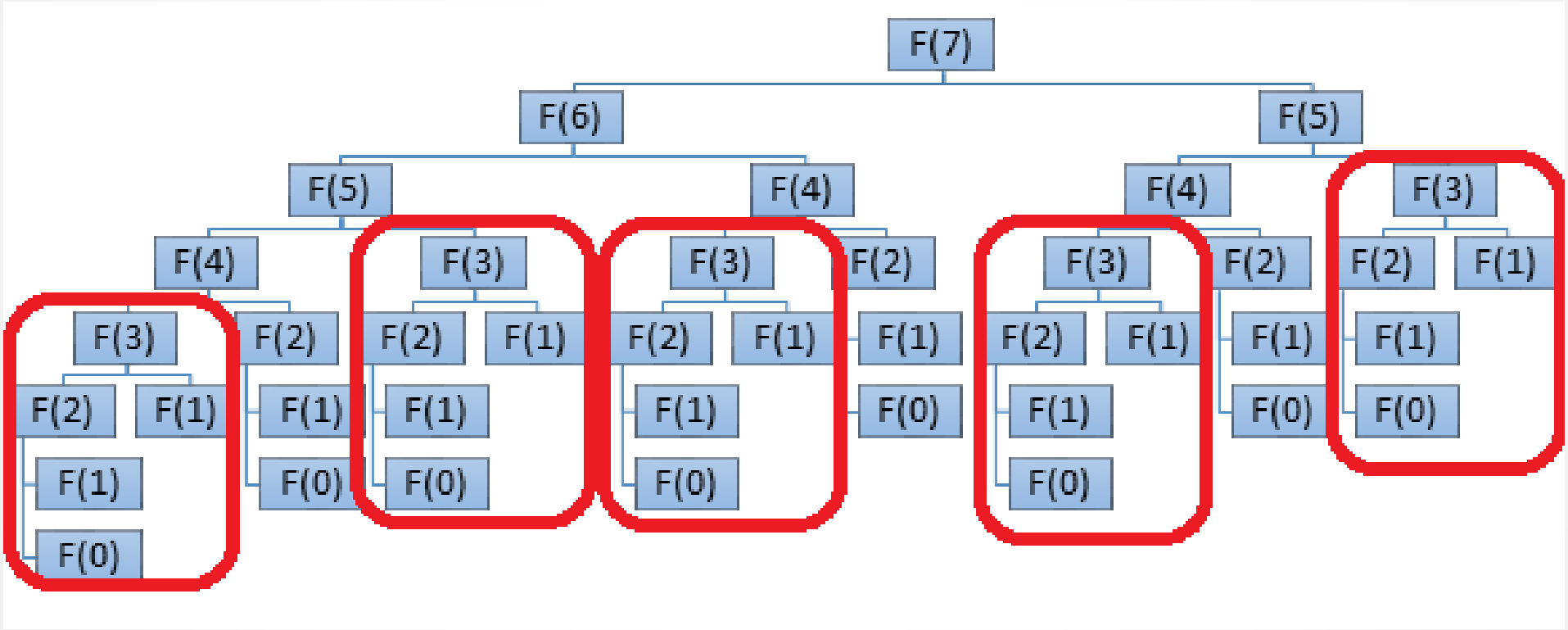
Получается, что наш алгоритм работает экспоненциальное время.

Можно ли быстрее???

Числа Фибоначчи



Числа Фибоначчи



Мемоизация

Идея: вычисленное значение сохраняем в структуру данных («*таблица*»), и при последующих обращениях не перевычисляем, а берём готовое значение. Этот приём называется «*мемоизация*».

Алгоритм:

Массив M[2..*n*], заполнить NULL.

Вернуть F_Memo(M,*n*).

Алгоритм F_Memo(M,*n*):

Если n=0, то вернуть 0.

Если n=1, то вернуть 1.

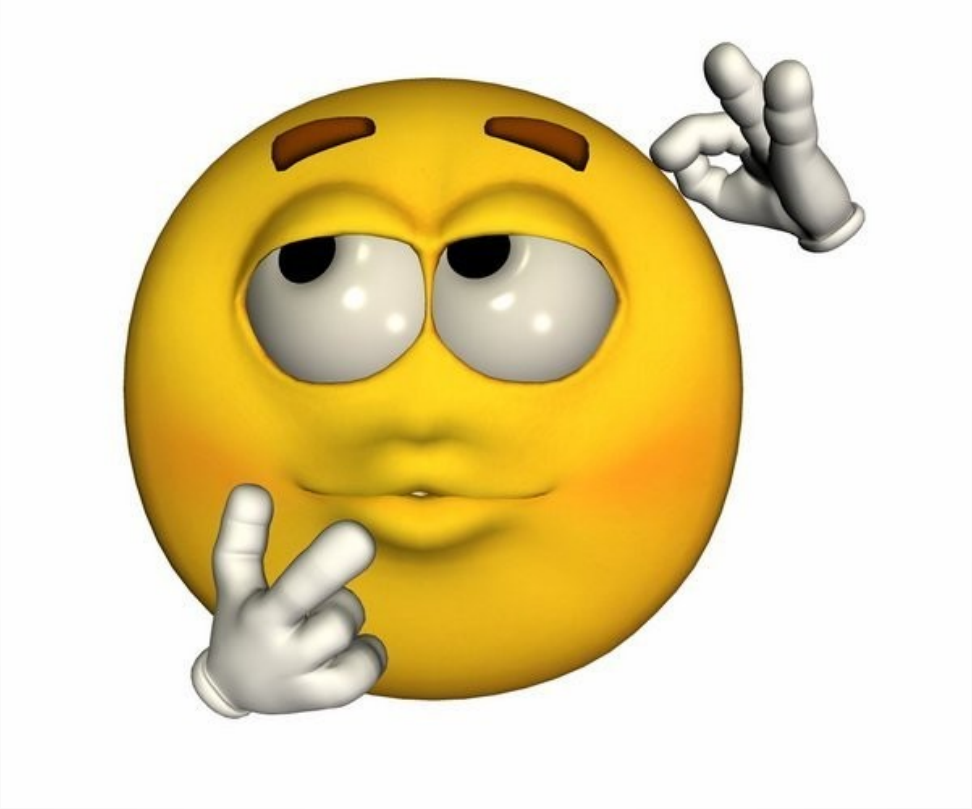
Иначе

Если M[*n*]=NULL, то M[*n*]= F_Memo(n-1)+F_Memo(n-2)

Вернуть M[*n*].

Числа Фибоначчи

Но нужна ли здесь рекурсия?



Числа Фибоначчи

Нерекурсивный алгоритм $F(n)$:

Массив F[0..n]

F[0] = 0; F[1] = 1;

Для i = 2 до n выполнять:

F[i] = F[i-1] + F[i-2]

Вернуть F[n].

Время: O(n). Память: O(n).

Числа Фибоначчи

Другой нерекурсивный алгоритм *F*(*n*):

A = 0; B = 1;

Для *i* = 2 до *n* выполнять:

C = A +B;

A = B;

B = C;

Вернуть C.

Динамическое программирование

Динамическое программирование

1. Разработать рекурсивную процедуру, вычисляющую решение для большой задачи на основе решений подзадач меньшего размера.
2. В цикле рассчитать решения подзадач от маленьких к большим.
3. Все построенные решения сохранять (в таблицу).
4. Для самых маленьких задач значения находятся с помощью другого (нерекурсивного) алгоритма.

Динамическое программирование

Когда стоит применять?

1. Построить рекурсивный алгоритм (декомпозиция задачи на более простые подзадачи).
2. Оценить сложность рекурсивного алгоритма.
3. Если сложность экспоненциальная, то провести анализ решаемых подзадач.
4. Если алгоритм многократно решает одни и те же подзадачи — следует использовать мемоизацию или ДП.

Динамическое программирование

Преобразование рекурсивного алгоритма в алгоритм ДП

1. Завести таблицу для хранения промежуточных результатов. Размерность таблицы равна количеству существенных аргументов рекурсивной функции. Размер по каждому измерению определяется допустимыми значениями соответствующего аргумента.

   Массив F[0..*n*]

Динамическое программирование

Преобразование рекурсивного алгоритма в алгоритм ДП

2. Все нерекурсивные выходы преобразовать в начальное заполнение таблицы.

F[0] = 0; F[1] = 1;

Динамическое программирование

Преобразование рекурсивного алгоритма в алгоритм ДП

3. Рекурсивные вызовы заменить на чтение из таблицы.
4. Вместо возврата вычисленного значения записывать его в таблицу.

Операции (3) и (4) внести в цикл заполнения таблицы.

Для $i = 2$ до n выполнять:

$$F[i] = F[i-1] + F[i-2]$$

Динамическое программирование

Преобразование рекурсивного алгоритма в алгоритм ДП

5. Возвращать одно из значений в таблице.

   Как правило, возвращается значение из угловой или граничной ячейки. Но могут быть и другие варианты.

   Вернуть F[*n*].

Динамическое программирование

Преобразование рекурсивного алгоритма в алгоритм ДП

5. Проанализировать возможность сократить необходимый объём памяти.

   A = 0; B = 1;

   ...

Варианты ДП для «Рюкзака»

Снова рассмотрим задачу о рюкзаке.

Пусть b — вместимость рюкзака, n — количество предметов.

Без потери общности можно считать веса и стоимости предметов целыми числами.

Построим рекурсивный алгоритм. Для того чтобы найти оптимальное решение полной задачи, рассмотрим 2 альтернативы

а) n-й предмет кладём в рюкзак => надо оставшиеся n-1 предмет оптимально уложить в рюкзак вместимостью $b - w_n$; получим решение стоимостью c^+;

б) n-й предмет НЕ кладём в рюкзак => надо оставшиеся n-1 предмет оптимально уложить в рюкзак исходной вместимостью b; получим решение стоимостью c^-.

Оптимальное решение задачи соответствует наилучшему из решений: $c^+ + c_n$ и c^-.

Варианты ДП для «Рюкзака»

Рекурсивная процедура решения задачи: Knapsack(*i*, *r*), где *i* — количество предметов, для которых решаем задачу ({1,..., *i*}), r — максимальный допустимый суммарный вес предметов, которые ещё можно положить в рюкзак. Возвращает стоимость оптимального решения для заданных *i* и *r*.

Для решения исходной задачи — вызов: Knapsack(*n*,*b*).

Knapsack(*i*,*r*)

if *i*=0 then return 0;

If w[*i*] <= *r* then return max{ *Knapsack*(*i*-1, *r*-w[*i*]) ; *Knapsack*(*i*-1, *r*) }

else return *Knapsack*(*i*-1, *r*)

Сложность рекурсивного алгоритма: $T(n)= 2T(n-1)+d \Rightarrow T(n)=O(2^n)$

Варианты ДП для «Рюкзака»

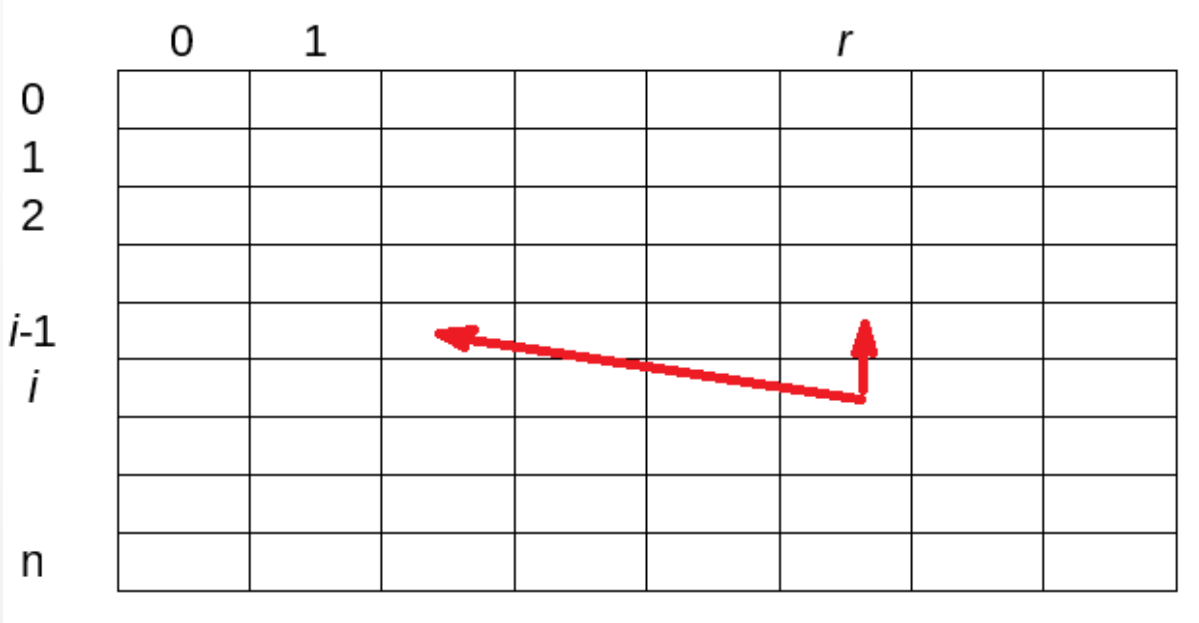
Применим метод динамического программирования. Создадим двумерную таблицу A. Значение A[i, r] = максимальная суммарная стоимость предметов из {1, …, *i* }, имеющих суммарный вес ≤ *r*.

- Инициализация:
  - A[0, r] = 0
  - A[i, 0] = 0 для всех i=0, …, *n*

Варианты ДП для «Рюкзака»

- Заполняем ячейки таблицы, в цикле по i=1,..., *n* и по *r*.
  - Если $w_i \le r$, то A[*i*, *r*] =max { A[*i* - 1, *r* - w_i] + c_i , A[*i* - 1, *r*] }.

    Первый вариант соответствует ситуации «кладём i-й предмет в рюкзак»; второй вариант - «не кладём».
  - Иначе A[*i*, *r*] =A[*i* - 1, *r*]. // Без вариантов — не кладём.
  - Во вспомогательной таблице *P* запоминаем, какой из вариантов выбора значения использован: P[*i*,*r*] = 1, если кладём предмет; 0 — если не кладём.



Варианты ДП для «Рюкзака»

- Строки: 0..*n*. Колонки: 0..*b*.
- Стоимость оптимального решения = значение в самой правой нижней ячейке: A[*n*,*b*].
- Для реконструкции решения проходим по P от ячейки [*n*,*b*] до верхней строки. Для текущей ячейки [*i*,*r*]:
  - Если P[*i*,*r*] = 1, то включаем *i*-й предмет в решение и переходим к ячейке [*i*-1, $r-w_i$].
  - Если P[*i*,*r*] = 0, то не включаем *i*-й предмет в решение и переходим к ячейке [*i*-1, *r*].

Временная и ёмкостная сложность: O(*nb*).

Варианты ДП для «Рюкзака»

Альтернативный (*двойственный*) вариант:

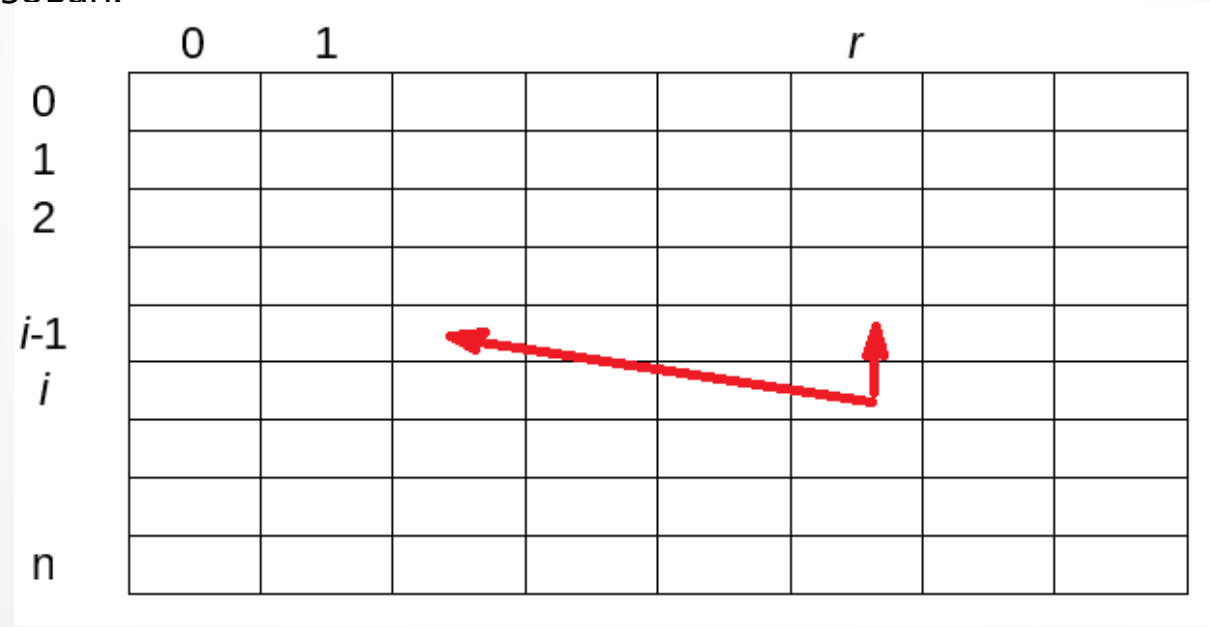
запоминать в A[*i*, *r*] не максимальную стоимость предметов суммарного веса *r*, а минимальный суммарный вес предметов из {1, …, *i* }, имеющих суммарную стоимость ровно *r* (и суммарный вес, не превышающий *b*).

- Инициализация:
  - A[0, r] = $+\infty$ для всех $r>0$;
  - A[i, 0] = 0 для всех i=0, …, *n*

Варианты ДП для «Рюкзака»

- Заполняем ячейки таблицы, в цикле по i=1,..., *n* и по *r*.
  - Если $c_i \le r$, то A[*i*, *r*] =min { A[*i* - 1, *r* - c_i] + w_i , A[*i* - 1, *r*] }.

    Первый вариант соответствует ситуации «кладём i-й предмет в рюкзак»; второй вариант - «не кладём».
  - Иначе A[*i*, *r*] =A[*i* - 1, *r*].
  - Во вспомогательной таблице *P* запоминаем, какой из вариантов выбора значения использован.



Варианты ДП для «Рюкзака»

- Стоимость оптимального решения = значение в самой правой ячейке нижней строки, не превышающее b.
- Для реконструкции решения проходим по P от этой ячейки до верхней строки.

Каковы должны быть размеры таблицы?

Строки индексируются от 0 до n.

Колонки соотвествуют возможным суммарным стоимостям. В идеале должны индексироваться от 0 до c^*, где c^* - стоимость оптимального решения. Проблема: мы не знаем значение c^*. Поэтому вместо c^* используем оценку сверху: $\bar{c}$. Простая, но грубая оценка: $\bar{c} = n\, c_{max}$.

Тогда таблица имеет размер $(n\text{-}1) \times (n\, c_{max} - 1)$.

Временная и ёмкостная сложность: $O(n^2\, c_{max})$.

При более тонкой оценке $\bar{c}$ можно получить сложность $O(n\, c_{max})$. Для этого надо вместо $\bar{c} = n\, c_{max}$ использовать для $\bar{c}$ стоимость решения, найденного жадным алгоритмом. Для него справедливо: $c' \geq c^*/2$.

Варианты ДП для «Рюкзака»

Итак, временная и ёмкостная сложность:

- Первый вариант: $O(nb)$.
- Второй вариант: $O(n^2 c_{max})$.

Это *псевдополиномиальный* алгоритм — экспоненциальный в общем случае (почему?), но полиномиальный при ограниченных числовых параметрах.

Псевдополиномальные алгоритмы

Рассмотрим задачи с числовыми параметрами, т. е. задачи, для которых входные данные содержат набор чисел, который может варьироваться для одного и того же размера входа.

Примеры: «Упаковка ящиков», «Рюкзак», «Коммивояжёр».

Определение. NP-трудная задача называется *NP-трудной в сильном смысле*, если она либо не содержит числовых параметров, либо является NP-трудной даже при ограниченных (полиномом от длины входа) значениях параметров.

Остальные NP-трудные задачи называются *NP-трудными в слабом смысле*.

NP-трудными в сильном смысле являются: «Упаковка ящиков», «Коммивояжёр» :(

Задача «Рюкзак» является NP-трудной в слабом смысле.