ВЫСШАЯ ШКОЛА ЭКОНОМИКИ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

Построение и анализ алгоритмов

Семинар 8

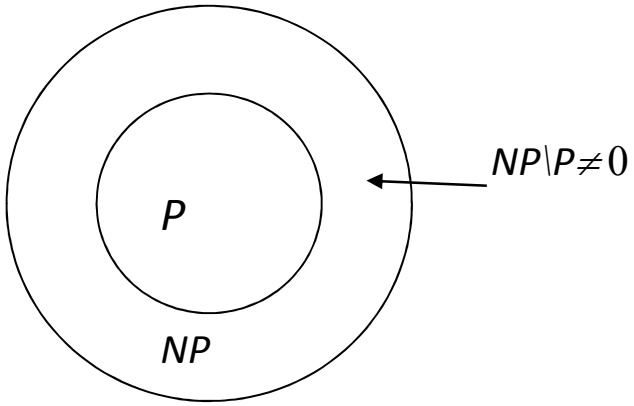
Задача о рюкзаке

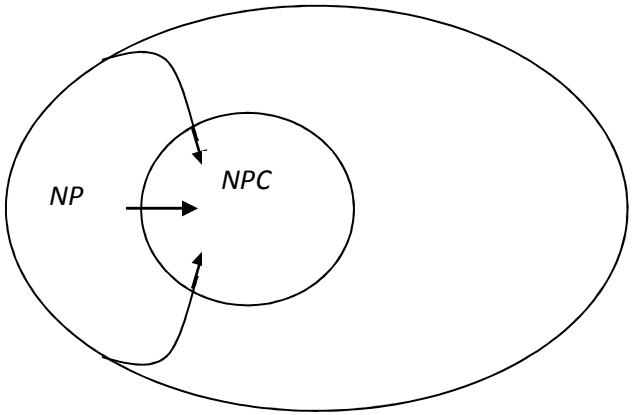
Классы задач

Класс **P** — класс задач с полиномиальной сложностью или класс полиномиально-решаемых задач.

Класс **NP** — класс задач с полиномиально проверяемым решением или класс полиномиально-проверяемых задач

Класс **NPC** или класс **NP**-полных задач





Задачи класса P

Задача называется полиномиальной, т. е. относится к классу P, если существует константа k и алгоритм, решающий эту задачу за время $O(n^k)$, где n есть длина входа алгоритма в битах.

Это задачи, решаемые за реальное время.

Преимущества :

— для большинства реальных задач из класса P константа k меньше или равна 6, что позволяет прогнозировать реальные времена решения задач для практически значимых размерностей входов;

— задачи класса P обладают инвариантностью (в смысле полиномиального времени) в рамках достаточно широкого класса моделей вычислений;

— класс обладает свойством естественной замкнутости, т. к. сумма или произведение полиномов также есть полином.

«Практически разрешимые задачи»

Задачи класса NP

Некоторый алгоритм получает решение некоторой задачи. Соответствует ли полученный ответ поставленной задаче, и насколько быстро можно проверить его правильность

Задача относится к классу NP, если ее решение, полученное некоторым алгоритмом, может быть проверено с полиномиальной временной сложностью.

Алгоритм, проверяющий решение этой задачи относится к классу P

Проблема P = NP

Основная проблема теории сложности. можно ли все задачи, решение которых проверяется с полиномиальной сложностью, решить также за полиномиальное время?

Очевидно, что любая задача, принадлежащая классу P, принадлежит и классу NP, т. к. она может быть полиномиально проверена, при этом задача проверки решения может состоять просто в повторном решении задачи.

Отсутствуют теоретические доказательства как совпадения классов P и NP, так и их несовпадения

Класс NPC задач

Понятие NP – полноты основывается на понятии сводимости одной задачи к другой

Определение класса NPC или класса NP-полных задач требует выполнения следующих двух условий:

—во-первых, задача должна принадлежать классу NP,

— во-вторых, к этой задаче должны полиномиально сводиться *все* задачи из класса NP

Класс NPC задач

Теорема: Если существует задача, принадлежащая классу NPC, для которой существует полиномиальный алгоритм решения, то класс P совпадает с классом NP

В настоящее время доказано существование сотен NP-полных задач, но ни для одной из них пока не удалось найти полиномиального алгоритма решения.

Для многих из них предложены приближенные полиномиальные алгоритмы

Задачи

Решаются задачи, в которых ставятся вопросы типа:

- «Перечислите все возможные варианты …»,
- «Сколько существует способов …»,
- «Есть ли способ …»,
- «Существует ли объект…» и т. п.

Примеры задач

- Расстановка n ферзей на доске n*n
- Обход шахматной доски конем
- Задача коммивояжера
- Латинский квадрат
- Задача о назначениях
- Задача об упаковке рюкзака

и т.д.

Подходы к решению

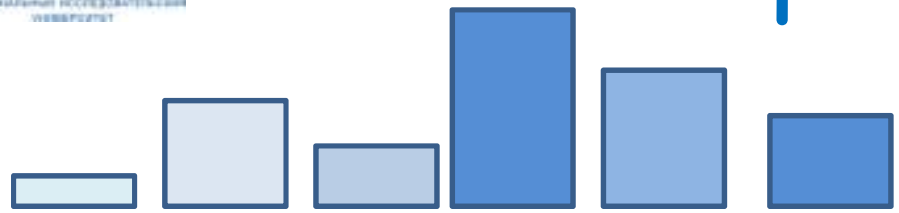
- Проводится **исчерпывающий поиск** в некотором множестве всех возможных вариантов, среди которых находятся решения конкретной задачи.
- Два метода организации исчерпывающего поиска:

1. **Перебор (поиск) с возвратом** (*Backtracking*).
   Модификации метода (представление данных, особенности реализации):
   - метод ветвей и границ (branch and bound),
   - поиск в глубину (depth first search),
   - метод проб и ошибок и т. д.
2. **метод решета** из множества возможных вариантов исключаются все элементы, не являющиеся решениями (вместо конструктивного построения решений задачи).

Подход к решению – перебор с возвратом

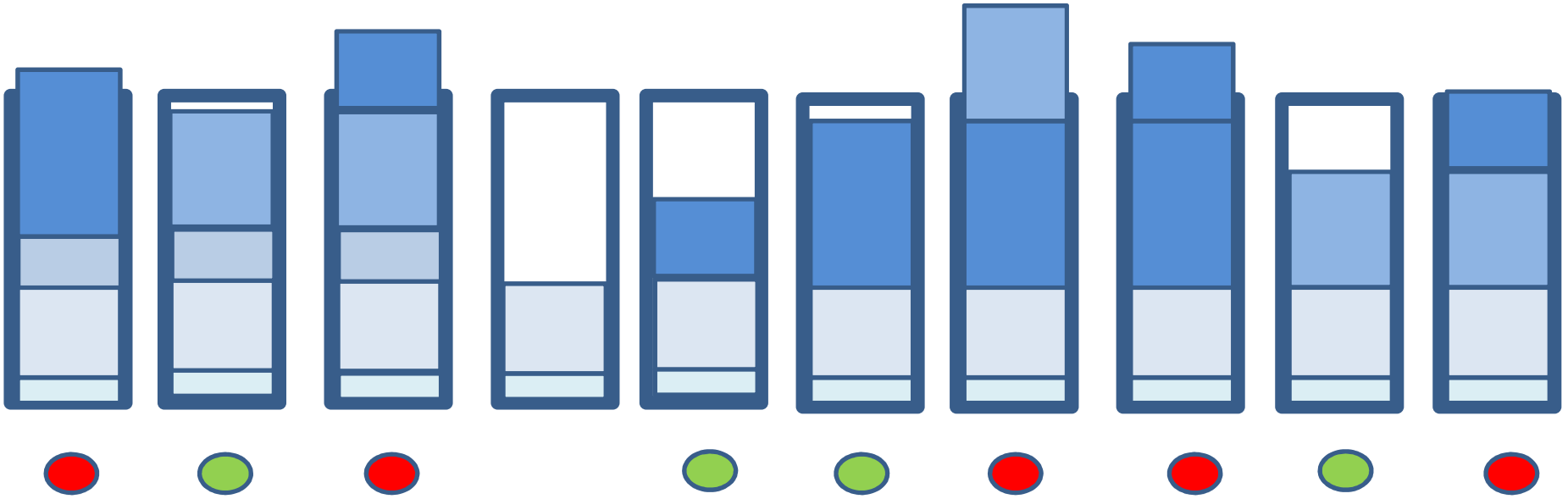
- Решение: последовательное расширение частичного решения.
- Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше.
- Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют.
- Для ускорения метода надо организовать вычисления таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты (эвристики).

Пример



Цвет – стоимость, размер - вес

Рюкзак заданного размера



И т.д.

Возвратная рекурсия

- Соединение метода перебора с возвратом и рекурсии определяет специфический способ реализации рекурсивных вычислений и называется **возвратной рекурсией**. Это соединение двух эффективных методов реализации переборных алгоритмов.

- При использовании возвратной рекурсии отпадает необходимость непосредственно организовывать возвраты и отслеживать правильность их осуществления. Они, как правило, становятся встроенной частью механизма выполнения рекурсивных вызовов.

- Ценность метода «перебор с возвратом + рекурсия»:
  - программы решения многих задач строятся по единой схеме,
  - они компактны и просты для понимания и усвоения соответствующих идей.

Постановка задачи в общем виде

Пусть $M_0, M_1, ..., M_{n-1}$ - n конечных линейно упорядоченных множеств,

G - совокупность ограничений (условий), ставящих в соответствие векторам вида

$$v = (v_0, v_1, ..., v_k)^T \quad (v_j \in \notin M_j;\ j = 0, 1, ... k;\ k<=n-1),$$

булево значение $G(v) \in \{истина, ложь\}$.

Векторы $v = (v_0, v_1, ..., v_k)^T$, для которых $G(v) = истина$, назовем **частичными решениями**.

Пусть, далее, существует конкретное правило P, в соответствии с которым некоторые из частичных решений могут объявляться **полными решениями**.

Тогда возможна постановка следующих поисковых задач:

1. Найти все полные решения или установить отсутствие таковых.
2. Найти хотя бы одно полное решение или установить его отсутствие.

Задание правила *P*

Часто задание правила *P* на частичных решениях имеет вид

$$P(v_0, v_1, \ldots, v_k) = \begin{cases} полное\ решение, & при\ k = n-1 \\ неполное\ решение, & при\ k < n-1 \end{cases}$$

Общий метод решения таких задач состоит в последовательном покомпонентном наращивании вектора *v* слева направо, начиная с v_0, и последующих испытаниях его ограничениями *G* и правилом *P*.

Схема 1. Нерекурсивный вариант схемы перебора с возвратом для нахождения всех решений

j ←0 // все решения, если есть, завершение при j=0
while j>=0 do
{ if (просмотрены не все элементы *Mj*)
{ *w* ←очередной не просмотренный элемент *Mj*
Элемент *w* объявляется просмотренным
if G(v_0, v_1, ..., v_{j-1},*w*)=*true* //частичное решение
if ((v_0, v_1, ..., v_{j-1},*w*)^T- полное решение)
Вывод((v_0, v_1, ..., v_{j-1},*w*)^T)
else {v_j ← *w;* if j<n-1 j ← j+1}
}
else j ← j – 1 //возврат к более короткому частичному
//решению. Все элементы *Mj* становятся непросмотренными
}

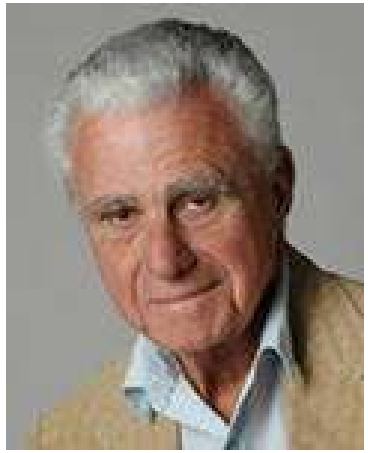
Схема 2. Рекурсивный вариант схемы перебора с возвратом для нахождения всех решений

```
backtraking(j)  // ищет все решения расширением
                // глобального массива (v0, v1, ..., vj-1)T
for (w ∈ Mj )
    { if G(v0, v1, ..., vj-1,w)=true //частичное решение
        { vj ← w
          if ((v0, v1, ..., vj-1,w)T- полное решение)
              Вывод((v0, v1, ..., vj-1,w)T)
           else
              if (j<n-1) backtraking(j+1)
        }
     }     // вызов:  backtraking(0)
```

Схема 3. Рекурсивный вариант схемы перебора с возвратом для нахождения одного решения

```
backtraking(j,pri)  // ищет одно решение расширением
                    // глобального массива (v0, v1, ..., vj-1)^T
while ((просмотрены не все элементы Mj) && (pri = false))
    { if G(v0, v1, ..., vj-1,w)=true //частичное решение
          { vj ← w
            if ((v0, v1, ..., vj-1,w)^T- решение задачи)
                { Вывод((v0, v1, ..., vj-1,w)^T); pri=True}
             else
                  if (j < n-1) backtraking(j+1,pri)
          }
     }     // вызов:   backtraking(0,false)
```

Динамическое программирование



Ричард Беллман ввел понятие динамическое программирование.

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

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 будет минимальным или максимальным.

Задача о рюкзаке в случае, когда вес каждого предмета представляет собой целое число, может быть решена с помощью динамического программирования

Постановка задачи

Имеется

- набор объектов $o_1, o_2, \ldots, o_N$
- весов (объемов) $w_1, w_2, \ldots, w_N$ и
- стоимостей $s_1, s_2, \ldots s_N$.

Необходимо упаковать рюкзак объемом W так, чтобы стоимость его S была максимальной.

Разновидности

Каждый предмет из множества можно выбирать произвольное количество раз.

Каждый предмет можно использовать только один раз.

Рекуррентное соотношение (однократный выбор)

$K_{w,i}$ - максимальная стоимость, которую можно набрать при весе не более w среди первых i предметов

Рекуррентное соотношение:

$$K_{0,i} = 0, \quad 0 \leq j \leq n$$

$$K_{w,0} = 0, \quad 0 \leq w \leq W$$

$$K_{w,i} = \begin{cases} \max\left(K_{w,i-1}, K_{w-w_i,i-1} + v_i\right), & w_i \leq w \\ K_{w,i-1}, & w_i > w \end{cases}, 0 \leq w \leq W$$

Пример

Двумерный массив: строка – предмет, столбец – вес

Дополнительные строка и столбец с нулями

Нижняя ячейка столбца дает максимальную стоимость для данного размера рюкзака.

Каждый столбец, кроме первого, заполняется на основе размера предмета в соответствующей строке и текущего суммарного размера предметов, положенных в рюкзак.

Обозначим предметы парой чисел $[w_i, s_i]$. Рассмотрим пример заполнения таблицы для 6 предметов, W=10

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	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

	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

Использование 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 (запоминания) получаем псевдополиномиальную сложность O(n * W)

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	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 клетки – не нужно вычислять, красные – вычисляются повторно

Использование рекурсии

Сначала заполним таблицу символами (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 (без вызова рекурсии)}.

```
#include <vector>
#include <limits>
int knapsack2(const std::vector<int>& wts,
                    const std::vector<int>& cost, int W)
{   size_t n = wts.size();
    std::vector<std::vector<int> > dp(W + 1,
                        std::vector<int>(n+1, 0));
  for (size_t j = 1; j <= n; j++)
   { for (int w = 1; w <= W; w++)
     { if (wts[j-1] <= w)
       { dp[w][j] = std::max(dp[w][j - 1],
                    dp[w - wts[j -1]][j - 1] + cost[j-1]); }
         else { dp[w][j] = dp[w][j - 1];
} } }
return dp[W][n]; }
```

Приближения к задаче об упаковке рюкзака

Простой жадный алгоритм, выбирающий наилучшее отношение стоимости к размеру.

Каждый объект представляется в виде пары [размер, стоимость].

Удельные стоимости = стоимость/размер.

Сортируем объекты по удельной стоимости по убыванию

Заполняем рюкзак последовательно элементами сортированного списка

Если объект не входит, отбрасываем его и переходим к следующему

Это – не оптимальный алгоритм

Домашняя работа

Реализовать алгоритмы упаковки рюкзака (используем предметы один раз)

Рекурсивный с возвратом

Рекурсивный (динам.программирование)

Упрощенный

Входы:

максимальный вес рюкзака от 50 до 100 с шагом 10.

Использовать 10 предметов

Веса предметов от 10 до 30

Стоимость от 200 до 1000

Домашняя работа

Проанализировать полученные результаты

(оценить временную сложность, сравнить алгоритмы и полученные результаты.)