6. Динамічне програмування (Dynamic Programming)

Динамічне програмування – метод розв’язання задачі шляхом її розбиття на декілька однакових підзадач, рекурентно пов’язаних між собою.

Динамічне програмування - це як метод математичної оптимізації, так і метод комп'ютерного програмування.

Динамічне програмування знайшло застосування у численних галузях - від аерокосмічної інженерії до економіки.

Слово динамічне було обране Беллманом, тому що звучало більш переконливо і краще підходило для передачі того факту, що проблема оптимального управління, яку він розв'язував цим методом, має аспект залежності від часу.

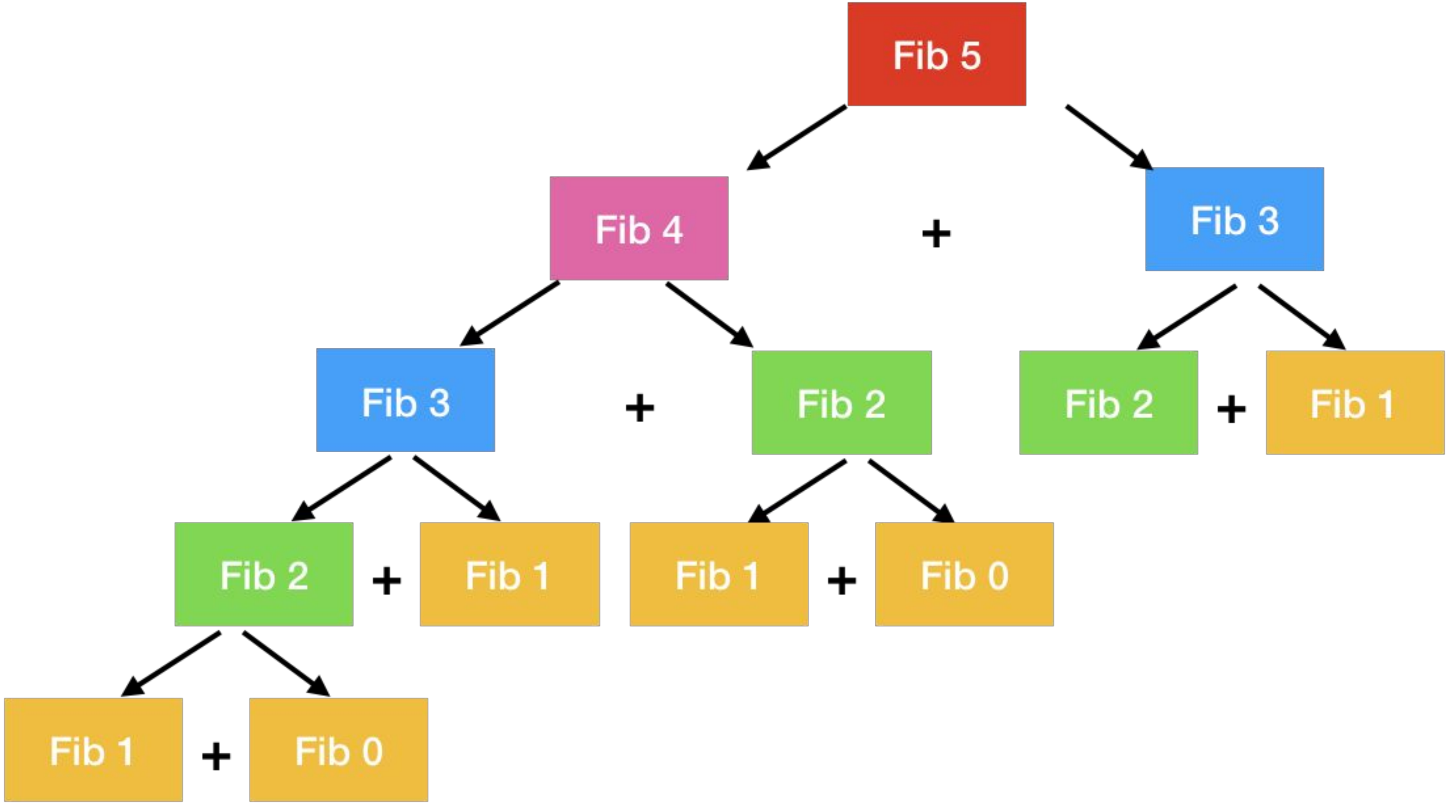
Слово програмування в цьому словосполученні в дійсності до «традиційного» програмування (написання тексту програм) майже ніякого відношення не має.

Словосполучення “Динамічне програмування” схоже на “лінійне програмування” та “математичне програмування”, які фактично є синонімами для математичної оптимізації. Тут воно означає оптимальну послідовність дій, оптимальну програму для отримання розв'язку задачі. Наприклад, певний розклад подій на виставці чи в театрі теж називають програмою. Програма в даному випадку розуміється як запланована послідовність подій.

Динамічне програмування на практиці:

- “Розумний перебір”
- “Розбиття на підзадачі” + “Перевикористання їх рішень”
- ...

Послідовність Фібоначчі:
F(n)=F(n-1) + F(n-2); F(1)=1; F(0)=0;



Наївна реалізація:

```
def naive_fibonacci(n):
    if n == 0 or n == 1:
        result = n
    else:
        result = naive_fibonacci(n-1) + naive_fibonacci(n-2)
    return result
```

Складність наївної реалізації - Експоненційна

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

$T(n-1)+T(n-2)+C >= 2T(n-2) \sim 2^{(n/2)}$

Число	**Час виконання**
F(5)	2.4 нс
F(35)	5.04 с
F(40)	Over 9000

Мемоізація

```
memory = {}

def memoized_fibonacci(n):
    if n in memory:
        return memory[n]

    if n == 0 or n == 1:
        result = n
    else:
        result = memoized_fibonacci(n-1) + memoized_fibonacci(n-2)

    memory[n] = result
    return result
```

Складність реалізації з мемоізацією - Лінійна

memory[n] ~ O(1)

Кількість викликів fib(n) = n, кожен з яких ~ O(1)

Число	**Час виконання**
F(5)	2.15 нс
F(35)	17.8 нс
F(40)	19.3 нс

Динамічне програмування на практиці:

- “Розумний перебір”
- “Розбиття на підзадачі” + “Перевикористання їх рішень”
- “Рекурсія” + “Мемоізація”

Час вирішення задачі = кількість підзадач * час вирішення підзадачі

Підхід “знизу вгору” (“Bottom-up”)

```
def bottom_up_fibonacci(n):
    memory = {}
    for k in range(n+1):
        if k == 0 or k == 1:
            result = k
        else:
            result = memory[k-1] + memory[k-2]
        memory[k] = result
    return memory[n]
```

Підхід “знизу вгору” (“Bottom-up”)

- Виконує точно ті ж операції, що і рекурсивний підхід
- Рекурсивний підхід завжди можна переписати як Bottom-up, і навпаки
- Виконує топологічне сортування підзадач
- Легше оцінити складність

Підхід “знизу вгору” (“Bottom-up”) з оптимізацією пам’яті

```python
def bottom_up_fibonacci_constant_space(n):
    k_minus_1, k_minus_2 = 1, 0
    for k in range(2, n+1):
        result = k_minus_1 + k_minus_2
        k_minus_1, k_minus_2 = result, k_minus_1

    return result
```

Динамічне програмування у 5 кроків

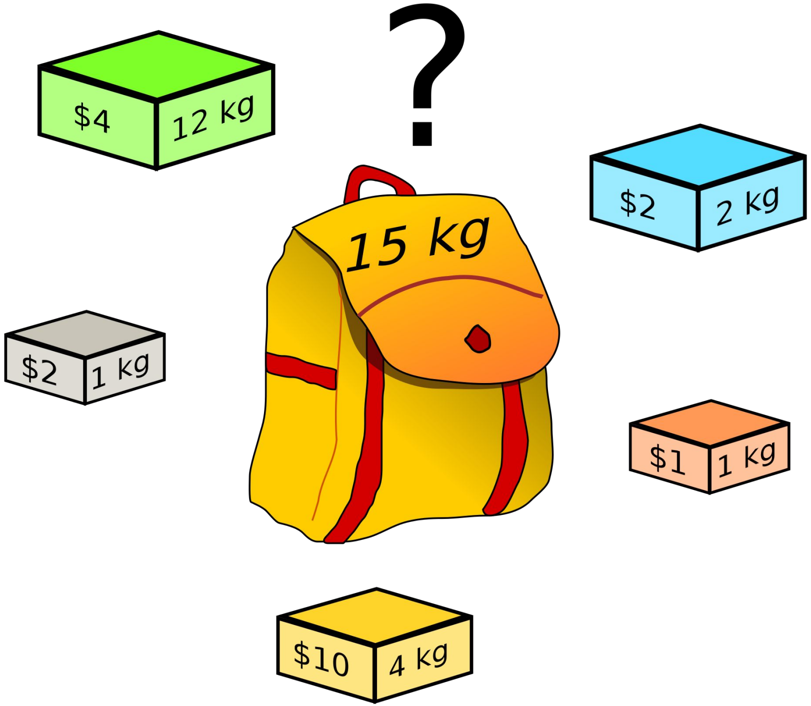
1. Визначити підзадачі та порахувати їх кількість
2. “Вгадати” частину рішення
3. Пов’язати розв’язки підзадач (наприклад, через рекурсію)
4. Побудувати алгоритм
   a. Рекурсія + мемоізація
   b. Bottom-up
5. Вирішити оригінальну задачу

Динамічне програмування у 5 кроків для послідовності Фібоначчі

Визначити підзадачі та порахувати їх кількість	F(k) for k=0,...,n-1
“Вгадати” частину рішення	-
Пов’язати розв’язки підзадач (наприклад, через рекурсію)	F(k) = F(k-1) + F(k-2)
Побудувати алгоритм a. Рекурсія + мемоізація b. Bottom-up	k=0,...,n
Вирішити оригінальну задачу	F(n)

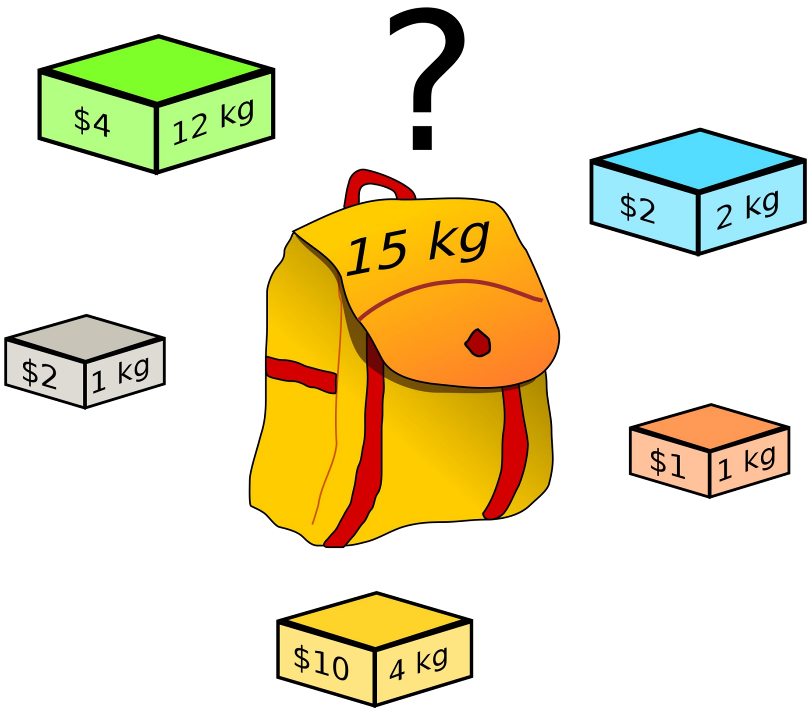
Задача пакування рюкзака (Knapsack problem)

Задача пакування рюкзака — задача комбінаторної оптимізації: для заданої множини предметів, кожен з яких має вагу і цінність, визначити яку кількість кожного з предметів слід взяти, так, щоб сумарна вага не перевищувала задану, а сумарна цінність була максимальною.



Бінарна задача пакування рюкзака (0-1 Knapsack problem)

Бінарна задача пакування рюкзака — для заданої множини предметів, кожен з яких має вагу і цінність, визначити які з предметів слід взяти (0 - не взяти, 1 - взяти), так, щоб сумарна вага не перевищувала задану, а сумарна цінність була максимальною.



Формулювання

$w(i)$ - вага (weight) i-го предмета.

$v(i)$ - цінність (value) i-го предмета.

W - максимальна вага, яку вміщає рюкзак.

Потрібно максимізувати суму $v(i)$ для предметів, які беруться, за умови, що сума їх $w(i) \le W$.

Динамічне програмування у 5 кроків для задачі пакування рюкзака

Визначити підзадачі та порахувати їх кількість	Knapsack(i, X) for i=1,...,n; X=0,...,W - залишок ваги; кількість підзадач: O(n*W)
“Вгадати” частину рішення	Брати предмет i, чи ні
Пов’язати розв’язки підзадач (наприклад, через рекурсію)	Knapsack(i, X) = max(Knapsack(i-1, X), Knapsack(i-1, X-w(i)) + v(i))
Побудувати алгоритм a. Рекурсія + мемоізація b. Bottom-up	i=0,...,n; X <= W
Вирішити оригінальну задачу	Knapsack(n, W)

Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Knapsack(i, X) = max(Knapsack(i-1, X), Knapsack(i-1, X-w(i)) + v(i))

Підхід “знизу вгору” (“Bottom-up”)

```
def knapsack(weights, values, total_weight):
    memory = {(0, w): 0 for w in range(total_weight + 1)}

    for i in range(1, len(weights)):
        for x in range(0, total_weight+1):
            if weights[i] > x:
                value = memory[(i-1, x)]
            else:
                value = max(memory[(i-1, x)], memory[(i-1, x-weights[i])] + values[i])

            memory[(i, x)] = value

    return memory[len(weights) - 1, total_weight]
```

Шлях до оптимального рішення. Parent pointers

```python
def knapsack_with_parent_pointers(weights, values, total_weight):
    parent_pointers = {}
    memory = {(0, w): 0 for w in range(total_weight + 1)}

    for i in range(1, len(weights)):
        for x in range(0, total_weight+1):
            if (weights[i] > x) or (memory[(i-1, x)] > memory[(i-1, x-weights[i])] + values[i]):
                value = memory[(i-1, x)]
                parent_pointers[(i, x)] = False
            else:
                value = memory[(i-1, x-weights[i])] + values[i]
                parent_pointers[(i, x)] = True

            memory[(i, x)] = value

    x = total_weight
    included_items = []
    for i in reversed(range(1, len(weights))):
        if parent_pointers[(i, x)]:
            included_items.append(i)
            x -= weights[i]

    return memory[len(weights) - 1, total_weight], included_items
```

Algorithms that use dynamic programming [edit]

This section **does not cite any sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed.
Find sources: "Dynamic programming" – news · newspapers · books · scholar · JSTOR *(May 2013) (Learn how and when to remove this template message)*

- Recurrent solutions to lattice models for protein-DNA binding
- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization problems
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-horizon, discrete-time, discounted, time-invariant dynamic optimization problems
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, Levenshtein distance (edit distance)
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth or bounded clique-width by using dynamic programming on a tree decomposition of the graph.
- The Cocke–Younger–Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar
- Knuth's word wrapping algorithm that minimizes raggedness when word wrapping text
- The use of transposition tables and refutation tables in computer chess
- The Viterbi algorithm (used for hidden Markov models, and particularly in part of speech tagging)
- The Earley algorithm (a type of chart parser)
- The Needleman–Wunsch algorithm and other algorithms used in bioinformatics, including sequence alignment, structural alignment, RNA structure prediction
- Floyd's all-pairs shortest path algorithm
- Optimizing the order for chain matrix multiplication
- Pseudo-polynomial time algorithms for the subset sum, knapsack and partition problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth–Lewis method for resolving the problem when games of cricket are interrupted
- The value iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Photoshop

Корисні посилання

1) https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-19-dynamic-programming-i-fibonacci-shortest-paths/ 4 лекції починаючи з цієї
2) https://www.youtube.com/watch?v=P8Xa2BitN3I
3) http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf