Dynamické programování

KIV/ADT – 7. přednáška

Miloslav Konopík

18. dubna 2024

Obsah

Dynamické programování

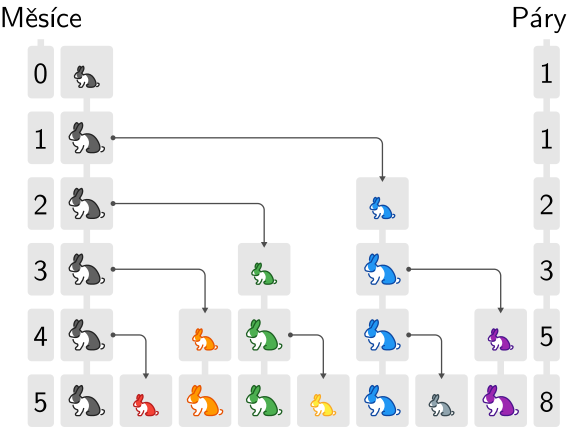
Dynamické programování

- Jedna z technika řešení složitých problémů pomocí rozkladu na menší, jednodušší podproblémy.
- Existence rekursivního řešení znamená, že podproblémy jsou rozložitelné.
- Rekurzivní algoritmus implikuje graf výpočtu.
- Dynamické programování, pokud se závislosti mezi podproblémy překrývají (tvoří DAG[1]).
- Dvě možnosti:
  - Shora dolů: "Rekurzivní volání s opakovaným využitím již vyřešených podproblémů" (zaznamenání a opětovné použití řešení podproblémů).
  - Zdola nahoru: "Opatrná hrubá síla" (řešení každého podproblému postupně).

[1] Directed Acyclic Graph – typ grafu, který nemá žádné smyčky ani cykly.

Příklad I – Úloha o zajících

- Máme pár zajíců a chceme spočítat, kolik zajíců bude po n měsících.
- Každý měsíc se páry rozmnožují a vytvářejí nové páry, kteří ale rozmnožují až za jeden měsíc.
- Kolik párů zajíců bude po n měsících?
  - První, se znovu narodí ve druhém měsíci; druhém měsíci jsou **dva** páry.
  - Pouze jeden je březí ve třetím měsíci, narodí opět jeden pár; tj. jsou **tři** páry.
  - Z nich dva březí ve stejném měsíci, takže je ve čtvrtém měsíci **pět** párů. Z nich tři páry porodí další tři páry; pokud je přidáme k pěti párům, máme v pátém měsíci **8** párů.



Příklad II

Úloha o schodech:

- Máme schodiště se n schody, přičemž můžeme buď přeskakovat jeden schod nebo dva schody.
- Kolik různých způsobů existuje, jak se dostat na vrchol schodiště?

Příklad II

Úloha o schodech:

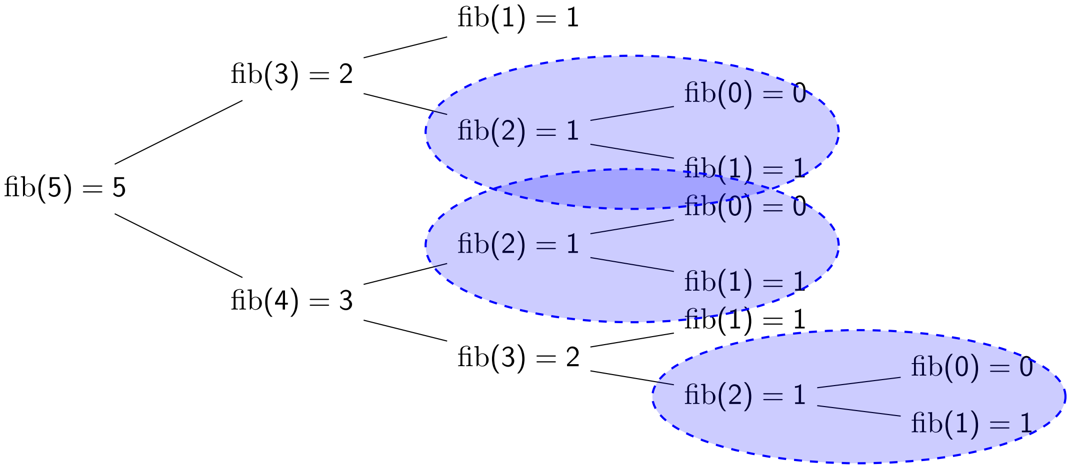
- Máme schodiště se n schody, přičemž můžeme buď přeskakovat jeden schod nebo dva schody.
- Kolik různých způsobů existuje, jak se dostat na vrchol schodiště?

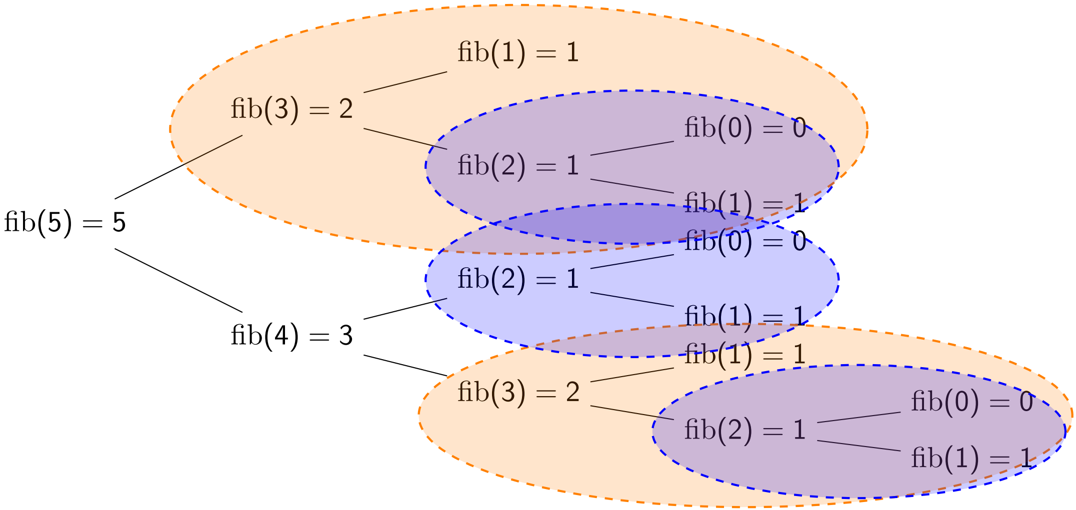
- Řešení:
  - Pokud nemáme žádný schod nebo jeden schod, existuje pouze jediný způsob, jak se na vrchol schodiště dostat.
  - Pro $n \geq 2$ schodů platí: $\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2)$, kde $\mathrm{fib}(n)$ je počet různých způsobů, jak se dostat na vrchol schodiště s n schody.
- Fibonacciho posloupnost.

Fibonacciho posloupnost – vizualizace

- $\mathrm{fib}(5) = 5$
  - $\mathrm{fib}(3) = 2$
    - $\mathrm{fib}(1) = 1$
    - $\mathrm{fib}(2) = 1$
      - $\mathrm{fib}(0) = 0$
      - $\mathrm{fib}(1) = 1$
  - $\mathrm{fib}(4) = 3$
    - $\mathrm{fib}(2) = 1$
      - $\mathrm{fib}(0) = 0$
      - $\mathrm{fib}(1) = 1$
    - $\mathrm{fib}(3) = 2$
      - $\mathrm{fib}(1) = 1$
      - $\mathrm{fib}(2) = 1$
        - $\mathrm{fib}(0) = 0$
        - $\mathrm{fib}(1) = 1$

Fibonacciho posloupnost – vizualizace




fib(5) = 5
fib(3) = 2
fib(1) = 1
fib(2) = 1
fib(0) = 0
fib(1) = 1
fib(4) = 3
fib(2) = 1
fib(0) = 0
fib(1) = 1
fib(3) = 2
fib(1) = 1
fib(2) = 1
fib(0) = 0
fib(1) = 1

Implementace Fibonacciho posloupnosti – rekurze shora dolů

```python
def fib(n: int) -> int:
    """Vrací n-tý Fibonacciho číslo.
    Args:
        n: Nezáporné celé číslo.
    Returns:
        N-té Fibonacciho číslo.
    Raises:
        ValueError: Pokud je n záporné.
    """
    if n < 0:   # Funkce je definována pouze pro nezáporné hodnoty
        raise ValueError("n must be non-negative")
    if n <= 1:  # Pokud je n menší nebo rovno 1, vrátíme n jako výsledek
        return n
    # Jinak rekurzivně zavoláme funkci fib pro dva předchozí členy
    return fib(n-1) + fib(n-2)
```

Implementace Fibonacciho posloupnosti – rekurze shora dolů

```python
def fib(n: int) -> int:
    """Vrací n-tý Fibonacciho číslo.
    Args:
        n: Nezáporné celé číslo.
    Returns:
        N-té Fibonacciho číslo.
    Raises:
        ValueError: Pokud je n záporné.
    """
    if n < 0:   # Funkce je definována pouze pro nezáporné hodnoty
        raise ValueError("n must be non-negative")
    if n <= 1:  # Pokud je n menší nebo rovno 1, vrátíme n jako výsledek
        return n
    # Jinak rekurzivně zavoláme funkci fib pro dva předchozí členy
    return fib(n-1) + fib(n-2)
```

Složitost?

Implementace Fibonacciho posloupnosti – rekurze shora dolů

```python
def fib(n: int) -> int:
    """Vrací n-tý Fibonacciho číslo.
    Args:
        n: Nezáporné celé číslo.
    Returns:
        N-té Fibonacciho číslo.
    Raises:
        ValueError: Pokud je n záporné.
    """
    if n < 0:   # Funkce je definována pouze pro nezáporné hodnoty
        raise ValueError("n must be non-negative")
    if n <= 1:  # Pokud je n menší nebo rovno 1, vrátíme n jako výsledek
        return n
    # Jinak rekurzivně zavoláme funkci fib pro dva předchozí členy
    return fib(n-1) + fib(n-2)
```

2^n

Implementace Fibonacciho posloupnosti – rekurze shora dolů s pamětí

```python
def fib_mem(n: int) -> int:
    # Vytvoříme seznam pro ukládání vypočítaných hodnot
    lookup: list[int] = [-1] * (n + 1)

    # Definujeme unitřní funkci pro rekurzivní výpočet s pamětí
    def fib_mem_int(n: int) -> int:
        if n <= 1:              # Pokud n <= 1, uložíme n jako výsledek.
            lookup[n] = n
        elif lookup[n] == -1: # Hodnota není v paměti, vypočteme.
            lookup[n] = fib_mem_int(n - 1) + fib_mem_int(n - 2)

        return lookup[n]      # Vrátíme hodnotu odpovídající danému n

    return fib_mem_int(n)     # Volání unitřní pomocné funkce
```

Implementace Fibonacciho posloupnosti – rekurze shora dolů s pamětí

```python
def fib_mem(n: int) -> int:
    # Vytvoříme seznam pro ukládání vypočítaných hodnot
    lookup: list[int] = [-1] * (n + 1)

    # Definujeme unitřní funkci pro rekurzivní výpočet s pamětí
    def fib_mem_int(n: int) -> int:
        if n <= 1:             # Pokud n <= 1, uložíme n jako výsledek.
            lookup[n] = n
        elif lookup[n] == -1: # Hodnota není v paměti, vypočteme.
            lookup[n] = fib_mem_int(n - 1) + fib_mem_int(n - 2)

        return lookup[n]      # Vrátíme hodnotu odpovídající danému n

    return fib_mem_int(n)     # Volání vnitřní pomocné funkce
```

Složitost?

Implementace Fibonacciho posloupnosti – rekurze shora dolů s pamětí

```python
def fib_mem(n: int) -> int:
    # Vytvoříme seznam pro ukládání vypočítaných hodnot
    lookup: list[int] = [-1] * (n + 1)

    # Definujeme unitřní funkci pro rekurzivní výpočet s pamětí
    def fib_mem_int(n: int) -> int:
        if n <= 1:              # Pokud n <= 1, uložíme n jako výsledek.
            lookup[n] = n
        elif lookup[n] == -1: # Hodnota není v paměti, vypočteme.
            lookup[n] = fib_mem_int(n - 1) + fib_mem_int(n - 2)

        return lookup[n]      # Vrátíme hodnotu odpovídající danému n

    return fib_mem_int(n)     # Volání unitřní pomocné funkce
```

Implementace Fibonacciho posloupnosti – výpočet zdola nahoru

```python
def fib_tab(n: int) -> int
    # Vytvoříme seznam pro ukládání vypočítaných hodnot
    f: list[int] = [0] * (n + 1)

    # Inicializujeme druhý člen posloupnosti
    f[1] = 1

    # Iterujeme od třetího členu až po n-tý a vypočítáme je pomocí
    ↪ předchozích dvou
    for i in range(2, n + 1):
        f[i] = f[i - 1] + f[i - 2]

    # Vrátíme n-tý člen posloupnosti
    return f[n]
```

Implementace Fibonacciho posloupnosti – výpočet zdola nahoru

```python
def fib_tab(n: int) -> int
    # Vytvoříme seznam pro ukládání vypočítaných hodnot
    f: list[int] = [0] * (n + 1)

    # Inicializujeme druhý člen posloupnosti
    f[1] = 1

    # Iterujeme od třetího členu až po n-tý a vypočítáme je pomocí předchozích dvou
    for i in range(2, n + 1):
        f[i] = f[i - 1] + f[i - 2]

    # Vrátíme n-tý člen posloupnosti
    return f[n]
```

Složitost?

Implementace Fibonacciho posloupnosti – výpočet zdola nahoru

```python
def fib_tab(n: int) -> int
    # Vytvoříme seznam pro ukládání vypočítaných hodnot
    f: list[int] = [0] * (n + 1)

    # Inicializujeme druhý člen posloupnosti
    f[1] = 1

    # Iterujeme od třetího členu až po n-tý a vypočítáme je pomocí předchozích dvou
    for i in range(2, n + 1):
        f[i] = f[i - 1] + f[i - 2]

    # Vrátíme n-tý člen posloupnosti
    return f[n]
```

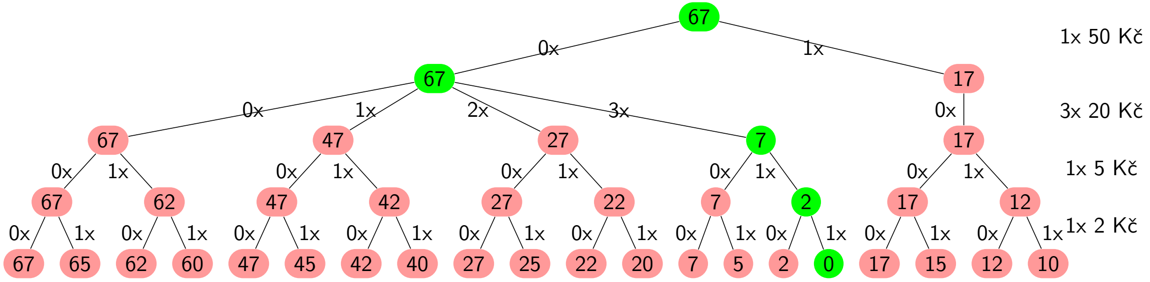
Úloha rozměňování mincí

Rozměňování mincí

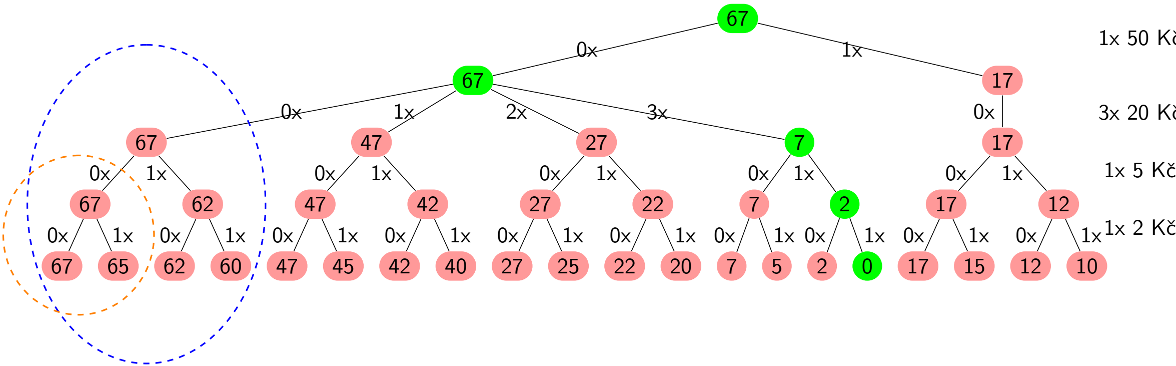
Úloha:

- Máme zadanou sumu a úkolem je najít nejmenší množství mincí, které danou sumu vytvoří.
- Každé mince máme neomezené / omezené množství.
- Řešení dynamickým programováním.

Výpočetní graf rozměňování – vizualizace



Výpočetní graf rozměňování – vizualizace



Rozměňování mincí – dekompozice na podúlohy

Rekurentní definice:

- Nechť $c[p] = \min \#$ mincí pro p korun
- Nechť $s[p] =$ poslední mince pro získání p korun
- Rekurentní definice:

$$c[p] = \begin{cases} 0, \text{ pokud } p = 0 \\ \min_{d_1 \leq d_i \leq p} \left(c\left[p - d_i\right] + 1 \right), \text{ pokud } p > 0 \end{cases}$$

Kvíz I – Dynamické programování: další hodnota

Kvíz II – Dynamické programování: rekurze

References I