IAL – 11. přednáška

Techniky řešení problémů

26. a 27. listopadu 2024

Obsah přednášky:

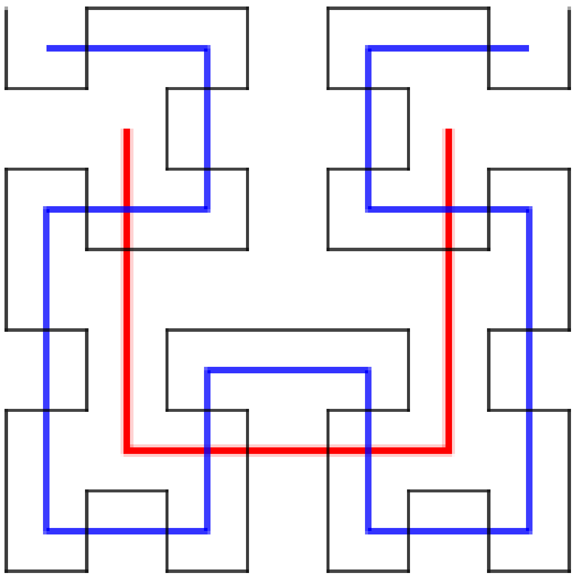
- Rozděl a panuj
  - Rekurze
  - Hanojské věže
- Dynamické programování
  - Fibonacciho čísla
  - Princip dynamického programování
  - Editační vzdálenost
  - Optimalizační problém batohu
  - Optimální BVS
  - Další případy využití dynamického programování

Rozděl a panuj

- Způsob řešení problému rozkladem na podproblémy
- Problém opakovaně rozkládáme na menší podproblémy, z jejichž výsledků potom složíme řešení celého problému
- Problém rozkládáme na podproblémy tak dlouho, až se dostaneme k tak jednoduchým vstupům, pro které už umíme problém vyřešit přímo
- Obvykle vede na rekurzivní algoritmus
- Příklady využití:
  - Quick sort
  - Merge sort
- Zmenši a panuj

Rekurze

- Metoda definování určitého objektu pomocí sebe sama
- Umožňuje definovat nekonečnou množinu objektů konečným popisem
- Rekurzivní struktura dat (lineární seznam)
- Rekurzivní struktura algoritmu
- Rekurze v matematických definicích
  - Definice přirozených čísel
  - Definice faktoriálu
- Definice geometrických objektů
  - Fraktály
    - Hilbertova křivka



Rekurze

- Konečnost rekurze – musí obsahovat ukončovací podmínku.
- Vztah rekurze a iterace:
  - Každou rekurzi lze nahradit iterací a naopak.
  - Rekurze – často průzračnější algoritmus.
  - Iterace – mnohdy efektivnější, využití zásobníku nebo jiné pomocné datové struktury.
- Příčiny neefektivity rekurze:
  - Opakované vyhodnocování funkce pro některé hodnoty argumentů.
  - Režie volání funkce.

Rekurze

- Přímá – volá se přímo
- Nepřímá – volá se zprostředkovaně přes jinou funkci

- Lineární – funkce volá sama sebou jen jedenkrát
- Násobná/stromová /kaskádová – funkce volá sebe sama vícekrát

- Koncová rekurze
  - Rekurzivní volání je posledním příkazem funkce.
  - Např. výpočet největšího společného dělitele.
  - Některé překladače nahradí obsah současného rámce nový obsahem a nemusí vkládat nový rámec na zásobník.

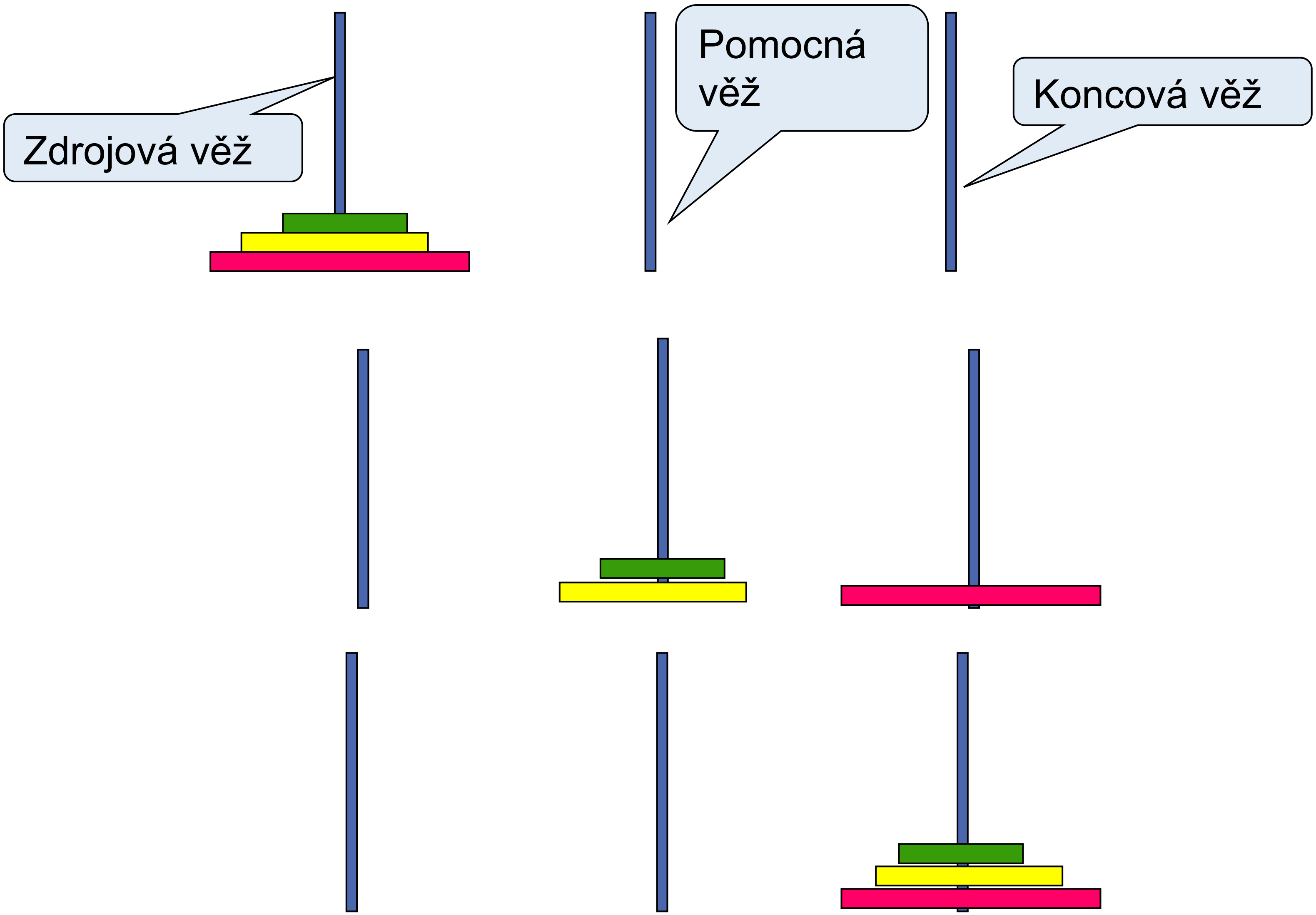
Rekurze a iterace

```
unsigned rFact (unsigned n)
  if n = 0:
    return 1
  else:
    return n*rFact(n-1)
```

```
usingned iFact (unsigned n)
  i ← 0
  f ← 1          // akumulátor
  while i < n:
    i ← i + 1
    f ← i * f
  return f
```

- ***Pozn.:*** U všech algoritmů v této prezentaci předpokládáme použití nezáporných čísel, i když u ostatních to již nezdůrazňujeme typem `unsigned`.

Hanojské věže



Hanojské věže – rekurzivně

```
procedure rHanoi (int h, int from, int to, int aux)
  if h > 0:
    rHanoi (h-1,from,aux,to)
    MoveDisk (from,to)
    rHanoi (h-1,aux,to,from)
```

Hanojské věže – nerekurzivně v.1

```
procedure PushInfo (int h, int from, int to, int aux)
  while h ≠ 0:
    Push(S,h,from,to,aux)
    to ↔ aux
    h ← h - 1

procedure iHanoi (int h, int from, int to, int aux)
  InitStack(S)
  PushInfo(h,from,to,aux)
  while not IsEmpty(S):
      TopPop(S,h,from,to,aux)
      MoveDisk(from,to)
      PushInfo(h-1,aux,to,from)
```

Hanojské věže – nerekurzivně v.2

- Problém Hanojských věží lze také řešit iterativně bez použití zásobníku dle tohoto schématu:
  - Pro sudý počet disků:
    - proveď legální přesun disku mezi věžemi A a B (v libovolném směru),
    - proveď legální přesun disku mezi věžemi A a C (v libovolném směru),
    - proveď legální přesun disku mezi věžemi B a C (v libovolném směru),
    - opakuj tyto tahy, dokud není věž přesunuta.
  - Pro lichý počet disků:
    - proveď legální přesun disku mezi věžemi A a C (v libovolném směru),
    - proveď legální přesun disku mezi věžemi A a B (v libovolném směru),
    - proveď legální přesun disku mezi věžemi B a C (v libovolném směru),
    - opakuj tyto tahy, dokud není věž přesunuta.

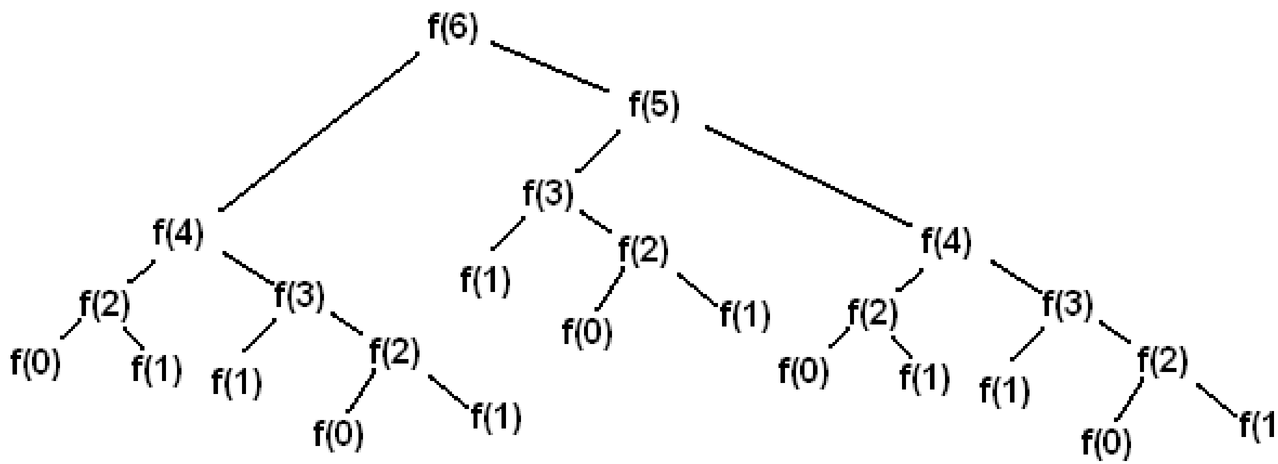
Dynamické programování

- Opět využívá rekurzivní rozklad problému na podproblémy.
- Pokud se ale podproblémy během rekurze opakují, řeší se pouze 1x → rychlejší algoritmy
- Využívá cyklus a pomocnou datovou strukturu.

- *Pozn.:* Název vymyslel Richard Bellman, který zkoumal vícekrokové plánování, v němž optimální volba každého kroku závisí na předchozích krocích – tzn. dynamické plánování (programování).

Fibonacciho čísla – rekurzivně v.1

$$F_n = F_{n-1} + F_{n-2}$$



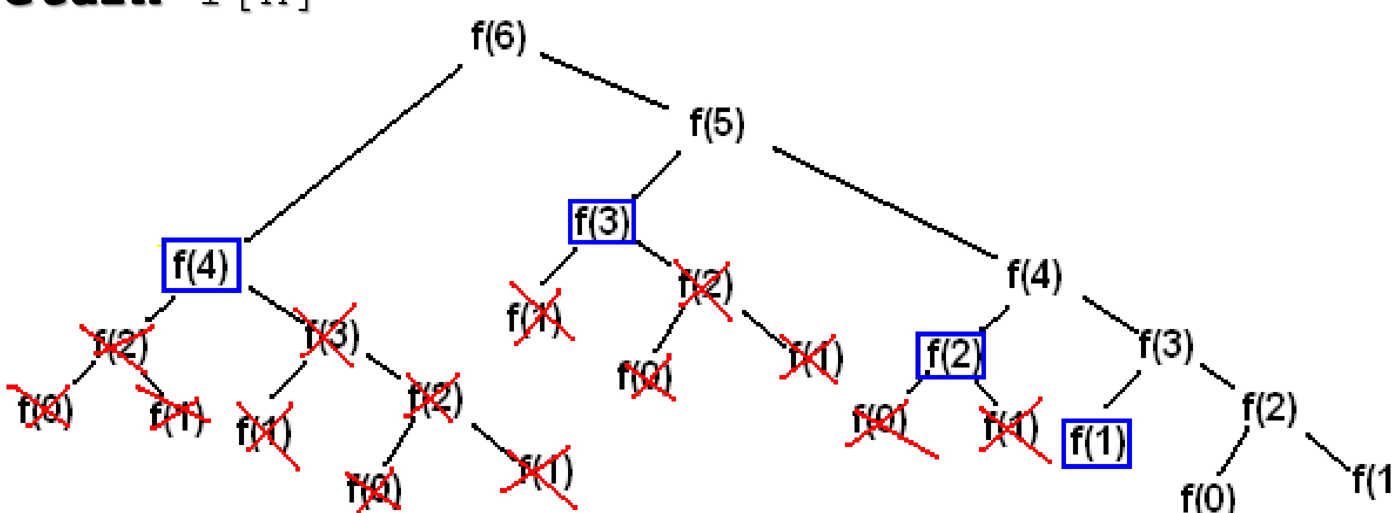

```
unsigned rFib (unsigned n)
// rekurzivní funkce pro výpočet n-tého Fibonacciho čísla
  if n = 0:
    return 0
  else:
    if n = 1:
      return 1
    else:
      return rFib(n-1) + rFib(n-2)
```

Fibonacciho čísla

- Lze ukázat, že složitost předchozího algoritmu je exponenciální (strom rekurze má přinejmenším exponenciálně mnoho listů), přestože funkci voláme pouze pro argumenty z rozsahu *0* až *n.*
- *Proč?* – protože mnohokrát počítáme totéž
- ***Vylepšení*** – co jsme již spočítali, si zapamatujeme v tabulce T a nebudeme to počítat znovu.
  - Tabulku můžeme implementovat např. globálním polem s prvky typu `int` o kapacitě *n+1* prvků, které před samotným výpočtem inicializujeme hodnotami -1.

Fibonacciho čísla – rekurzivně v.2

```
int rFib2 (int n)
  if T[n] ≠ -1:    // hodnota v tabulce pro n už je def.
    return T[n]
  else
    if n ≤ 1:
      T[n] ← n
    else:
      T[n] ← rFib2(n-1) + rFib2(n-2)
    return T[n]
```



Fibonacciho čísla – iterativně

- Získali jsme lineární časovou složitost.
- Dokonce ani nepotřebujeme rekurzi:

```
int iFib (int n)
  T[0] ← 0
  T[1] ← 1
  for i ← (2, n):
    T[i] ← T[i-1] + T[i-2]
  return T[n]
```

- *Pozn.:* Nepotřebujeme ani tabulku pro *n* prvků, stačí nám uchovávat poslední dvě Fibonacciho čísla.

Princip dynamického programování

1. Začneme s rekurzivním algoritmem, který je exponenciálně pomalý.
2. Odhalíme opakované výpočty stejných podproblémů.
3. Použijeme tabulku, ve které si budeme pamatovat výsledky podproblémů, které jsme již vyřešili (memoizace, kešování). Prořežeme tak strom rekurze a dostaneme rychlejší algoritmus.
4. Zvolíme vhodné pořadí řešení podproblémů, abychom se mohli vyhnout rekurzi (jednodušší algoritmus).

- Vhodné pro úlohy:
  - které lze rozdělit na podúlohy, které jsou si podobné a mohou se opakovat.
  - pro které platí, že optimální řešení problému obsahuje optimální řešení podproblémů.

Editační vzdálenost

- Editační vzdálenost (Levenshteinova vzdálenost) mezi dvěma řetězci je definována jako minimální počet operací, které musí být provedeny, aby řetězce byly totožné.
- Operace: substituce, vkládání, mazání
- Příklady:

  koule – boule: vzdálenost 1

  koule – kdoule: vzdálenost 1

  Kamil – omyl: vzdálenost 3

  potemník – poutník: vzdálenost ?

Editační vzdálenost

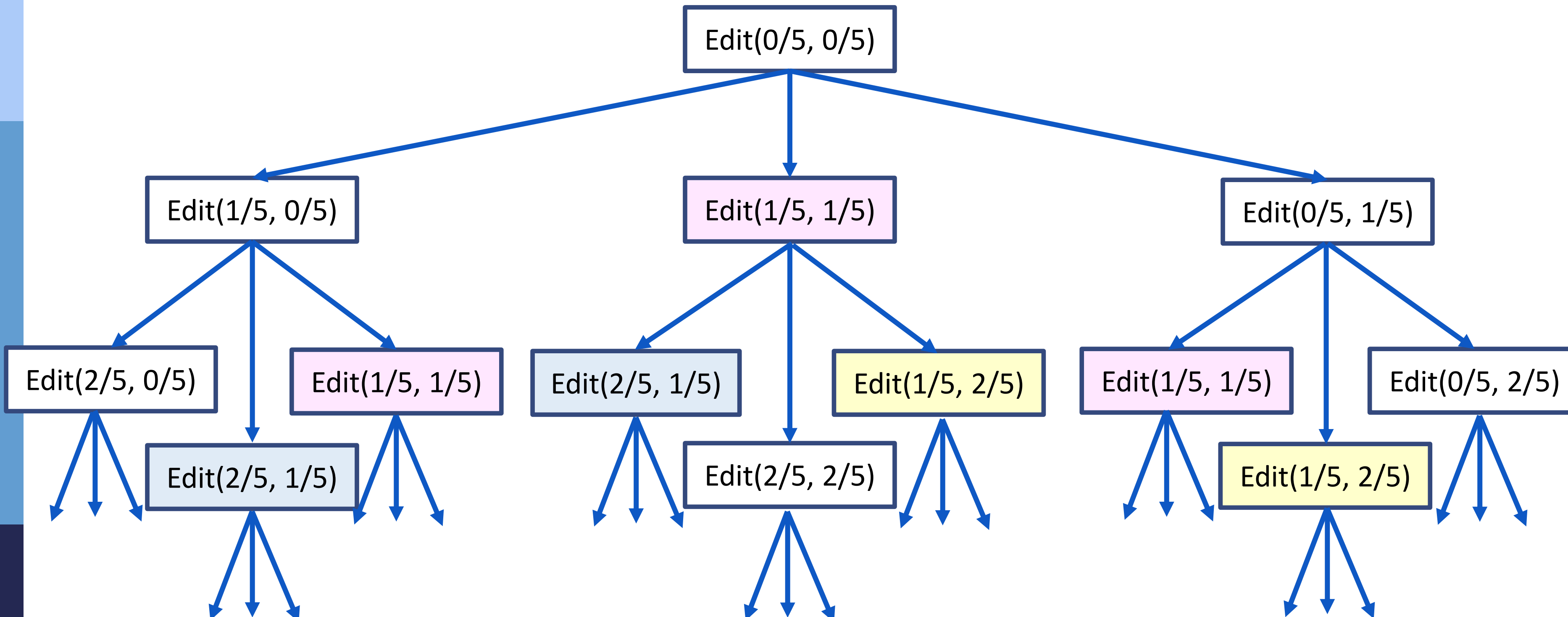
- Jaký nejmenší počet operací musíme provést, abychom z řetězce *x* vytvořili řetězec *y*?
- Každého znaku se týká nejvýše jedna editační operace, tzn. můžeme operace uspořádat *zleva doprava*.
- Můžeme předpokládat procházení řetězce x zleva doprava a jeho přetváření na řetězec y.
- Co může nastat pro první symbol?
  - Pokud $x_1 = y_1$, znak ponecháme beze změny a $L(x,y) = L(x_2 \ldots x_n, y_2 \ldots y_m)$
  - Znak x_1 změníme na y_1, pak $L(x,y) = 1 + L(x_2 \ldots x_n, y_2 \ldots y_m)$
  - Znak x_1 smažeme, pak $L(x,y) = 1 + L(x_2 \ldots x_n, y_1 \ldots y_m)$
  - Na začátek vložíme y_1, pak $L(x,y) = 1 + L(x_1 \ldots x_n, y_2 \ldots y_m)$
- Tzn. $L(x,y)$ závisí na vzdálenosti sufixů, kterou lze určit rekurzivně.
- Vzdálenost $L(\varepsilon,y) = |y|$

Editační vzdálenost - rekurzivně

```
int rEdit (int i, int j)
  if i > n:                          // řetězec x skončil
    return m-j+1
  else:
    if j > m:                        // řetězec y skončil
      return n-i+1
    else:
      le ← rEdit(i+1,j+1)   // ponechání/změna znaku
      if x[i] <> y[j]:
         le ← le + 1
      ld ← rEdit(i+1,j) + 1    // smazání znaku
      li ← rEdit(i,j+1) + 1    // vložení znaku
      return min(le,ld,li)
```

Pozn.: funkce hledá editační vzdálenost řetězců *x* a *y*, při každém volání hledá editační vzdálenost podřetězců $x_i \dots x_n$ a $y_j \dots y_m$

Editační vzdálenost – rekurze



Editační vzdálenost – lépe

- Časová složitost uvedeného algoritmu je exponenciální.
- Pro kolik různých vstupů můžeme funkci volat? *(n+1)*(m+1)*
- Exponenciální složitost je tedy opět způsobena opakovaným voláním funkce pro stejné hodnoty parametrů.
- Opět použijeme tabulku (matici) pro uchování již známých hodnot.
- Otočíme směr výpočtu, abychom se vyhnuli rekurzi, budeme postupovat od nejkratších sufixů směrem k delším sufixům.
- Výsledek – algoritmus běžící v čase $\Theta(nm)$

Editační vzdálenost - tabulka

y_j →, x_i ↓

	p	o	u	t	n	í	k	
p								8
o								7
t			3	2	3	4	5	6
e	4	3	2	2	2	3	4	5
m	4	3	2	1	1	2	3	4
n	4	3	2	1	0	1	2	3
í	5	4	3	2	1	0	1	2
k	6	5	4	3	2	1	0	1
	7	6	5	4	3	2	1	0

- Obsahuje editační vzdálenost pro všechny dvojice sufixů řetězců *x* a *y*
- Vyplňujeme od nejkratších sufixů směrem k nejdelším
- Inicializace: poslední řádek a sloupec – editační vzdálenost prázdného řetězce a všech sufixů příslušného řetězce
- Každý sufix můžeme získat 3 způsoby:
  - Přidáme pouze symbol z řetězce y ←
  - Přidáme symboly z obou řetězců ↖
  - Přidáme pouze symbol z řetězce x ↑
  - Použijeme způsob vedoucí na nejmenší editační vzdálenost (její hodnotu uložíme)

Editační vzdálenost - iterativně

```
int dpEdit (char *x, char *y)
  n ← length(x)
  m ← length(y)
  for i ← (0,n):
    T[i,m] ← n - i
  for j ← (0,m-1):
    T[n,j] ← m - j
  for i ← (n-1, 0)^-1:
    for j ← (m-1, 0)^-1:
      if x[i] = y[j]:
        d ← 0
      else:
        d ← 1
      T[i,j] ← min(d+T[i+1,j+1], 1+T[i+1,j], 1+T[i,j+1])
  return T[0,0]
```

y_j →, x_i ↓

	p	o	u	t	n	í	k	
p	3	4	4	4	5	6	7	8
o	4	3	3	3	4	5	6	7
t	4	3	3	2	3	4	5	6
e	4	3	2	2	2	3	4	5
m	4	3	2	1	1	2	3	4
n	4	3	2	1	0	1	2	3
í	5	4	3	2	1	0	1	2
k	6	5	4	3	2	1	0	1
	7	6	5	4	3	2	1	0

Editační vzdálenost - operace

- Kterých symbolů se dotknou editační operace? – 4 varianty:

	p	o	u	t	n	í	k	
p	**3**	4	4	4	5	6	7	8
o	4	**3**	3	3	4	5	6	7
t	4	3	**3**	**2**	3	4	5	6
e	4	3	2	2	**2**	3	4	5
m	4	3	2	1	**1**	2	3	4
n	4	3	2	1	**0**	1	2	3
í	5	4	3	2	1	**0**	1	2
k	6	5	4	3	2	1	**0**	1
	7	6	5	4	3	2	1	**0**

pout--ník
po-temník

	p	o	u	t	n	í	k	
p	**3**	4	4	4	5	6	7	8
o	4	**3**	3	3	4	5	6	7
t	4	3	**3**	2	3	4	5	6
e	4	3	**2**	2	2	3	4	5
m	4	3	2	**1**	1	2	3	4
n	4	3	2	1	**0**	1	2	3
í	5	4	3	2	1	**0**	1	2
k	6	5	4	3	2	1	**0**	1
	7	6	5	4	3	2	1	**0**

po-utník
potemník

Editační vzdálenost - operace

	p	o	u	t	n	í	k	
p	**3**	4	4	4	5	6	7	8
o	4	**3**	3	3	4	5	6	7
t	4	3	**3**	2	3	4	5	6
e	4	3	2	**2**	2	3	4	5
m	4	3	2	**1**	1	2	3	4
n	4	3	2	1	**0**	1	2	3
í	5	4	3	2	1	**0**	1	2
k	6	5	4	3	2	1	**0**	1
	7	6	5	4	3	2	1	**0**

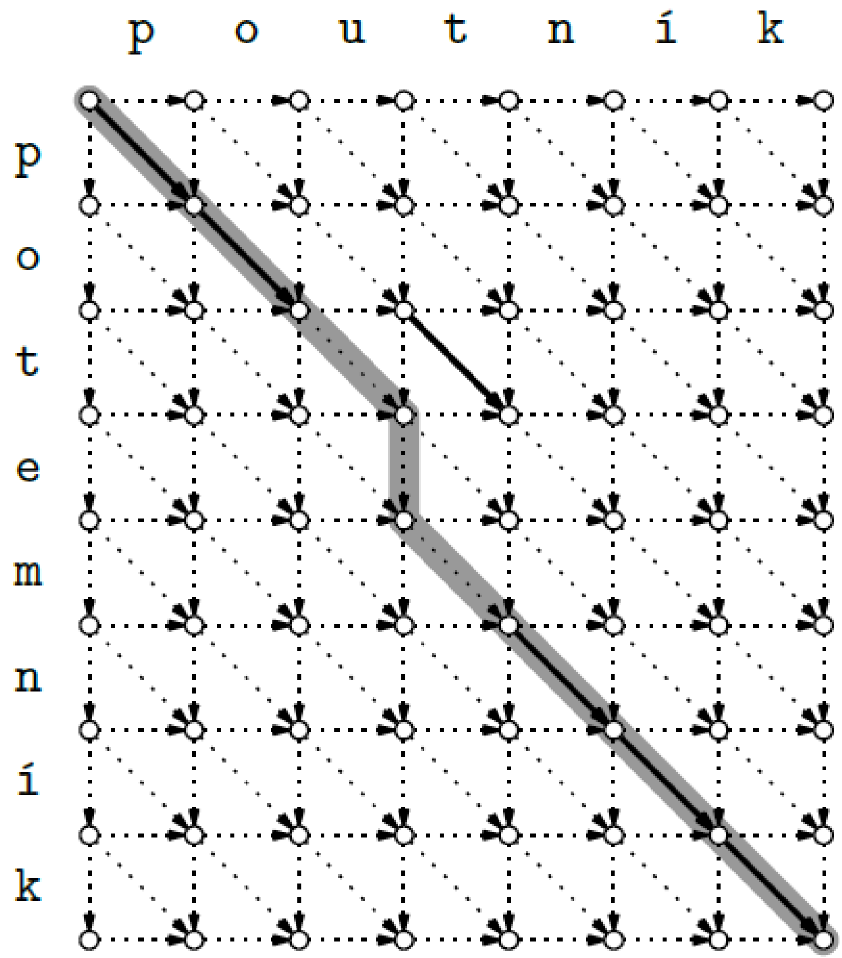
pou-tník
potemník

	p	o	u	t	n	í	k	
p	**3**	4	4	4	5	6	7	8
o	4	**3**	3	3	4	5	6	7
t	4	3	**3**	2	3	4	5	6
e	4	3	2	**2**	2	3	4	5
m	4	3	2	1	**1**	2	3	4
n	4	3	2	1	**0**	1	2	3
í	5	4	3	2	1	**0**	1	2
k	6	5	4	3	2	1	**0**	1
	7	6	5	4	3	2	1	**0**

pout-ník
potemník

Editační vzdálenost – graf

- Lze také popsat pomocí orientovaného grafu (symetrický problém):
  - Vrcholy – jednotlivé pozice v obou řetězcích
  - Hrany – možné operace (ponechání, změna, smazání, vložení znaku)
  - Délka hran 0 pro ponechání nezměněného písmene, v ostatních případech 1
  - Hledáme nejkratší cestu z vrcholu (0,0) do (n,m)



Optimalizační problém batohu

- *0-1 Knapsack problem*
- Máme batoh, který má danou nosnost.
- Dále máme množinu věcí, každá věc má určitou hmotnost a svoji cenu.
- **Úkol:** vybrat do batohu věci tak, aby nebyla překročena jeho nosnost a zároveň součet cen vybraných věcí byl maximální.

- *Pozn.:* Problém batohu je NP-úplný problém.

Optimalizační problém batohu

- **Formálněji:**
  - Nosnost batohu – celé číslo $W > 0$
  - $(v_1, \dots, v_n)$ je n-tice celých kladných čísel, kde v_i je hodnota (cena) položky i
  - $(w_1, \dots, w_n)$ je n-tice celých kladných čísel, kde w_i je váha položky i
- Hledáme množinu T, která splňuje následující podmínky:
  - $T \subseteq \{1 \dots n\}$ – řešením je podmnožina zadaných věcí
  - $\sum_{i \in T} v_i$ je maximální ze všech možných (hledá se maximální hodnota podmnožiny věcí)
  - $\sum_{i \in T} w_i \leq W$ – celková váha věcí musí být menší nebo rovna nosnosti batohu

Optimalizační problém batohu

0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

Problém batohu – řešení

- Rozložíme problém na podproblémy, které potom budeme skládat:
- Pokud nemáme žádnou věc – hodnota batohu je nulová
- Pokud už máme v batohu nějaké věci (batoh už má nějakou cenu a zbývá nám menší váha, kterou můžeme využít), pro každou další věc máme 2 možnosti co udělat:
  - Přidat věc do batohu – je-li váha věci menší, než zbývající váha, kterou můžeme využít, můžeme věc přidat a zvýšit tak cenu batohu
  - Nepřidat věc do batohu – hodnota batohu zůstane stejná, ale ušetříme váhu pro jiné věci
- Kterou možnost vybrat? – vybereme možnost, která vede na maximalizaci hodnoty batohu

Problém batohu – řešení

- **Formálněji:**
  $OPT(i, w)$ – podmnožina věcí s maximální hodnotou a váhovým limitem w

$$OPT(i,w) = \begin{cases} 0 & pro\ i = 0 \\ OPT(i-1,w) & pro\ w_i > w \\ max\{OPT(i-1,w), v_i + OPT(i-1,w-w_i)\} & jinak \end{cases}$$

Problém batohu – rekurzivně

```
int rKnapsack (int n, int W, int wt[], int val[])
// Vrací maximální hodnotu věcí, které nepřekročí nosnost W
// n je aktuální počet dostupných věcí
  if n = 0 or W = 0:
    return 0
  else:
    if (wt[n-1] > W):    // váha pol. překračuje nosnost
      return rKnapsack(n-1, W, wt, val)
    else:                // položka může být přidána
      return max(
        val[n-1]+rKnapsack(n-1, W-wt[n-1], wt, val),
        rKnapsack(n-1, W, wt, val))
```

Pozn.: váha a cena n-té věci je na indexu n-1.

Problém batohu – rekurze

Příklad rekurze pro kapacitu batohu 2 jednotky a 3 položky (každá s váhou 1 jednotka).

```
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}
                       K(3, 2)     ---------> K(n, W)
                   /              \
                /                    \
            K(2,2)                  K(2,1)
          /       \                 /    \
        /           \             /        \
     K(1,2)      K(1,1)        K(1,1)     K(1,0)
     /  \         /   \         /   \
   /      \     /       \     /       \
K(0,2)  K(0,1)  K(0,1)  K(0,0)  K(0,1)  K(0,0)
```

Problém batohu – lepší řešení

- Rekurzivní řešení je neefektivní, protože opakovaně řeší stejné podproblémy.
- Budeme raději ukládat výsledky již vyřešených problémů do vhodné „tabulky“.
  - Řešíme optimalizační problém s dvěma parametry (počet sebraných věcí a maximální váha), proto vytvoříme dvojrozměrné pole *M* o velikosti *n x W.*
  - Pozice *M[i,w]* bude obsahovat maximální hodnotu (cenu), kterou jsme schopni získat při využití věcí *0..i* a maximální váhy *w.*
  - Optimální řešení bude na pozici *M[n,W].*
- Výsledná časová složitost: $\Theta(nW)$

Problém batohu – iterativně

```
int dpKnapsack (int n, int W, int wt[], int val[])
// Vrací maximální hodnotu věcí, které nepřekročí nosnost W
  for i ← (0, n):
    for w ← (0, W):
      if i = 0 or w = 0:    // nemáme žádnou věc nebo váhu
        M[i,w] ← 0
      else:
        if wt[i-1] ≤ w:            // věc lze přidat
            M[i,w] ← max(
               val[i-1] + M[i-1,w-wt[i-1]], M[i-1,w])
        else:                 // věc nelze přidat kvůli váze
          M[i,w] ←  M[i-1,w]
  return M[n,W]
```

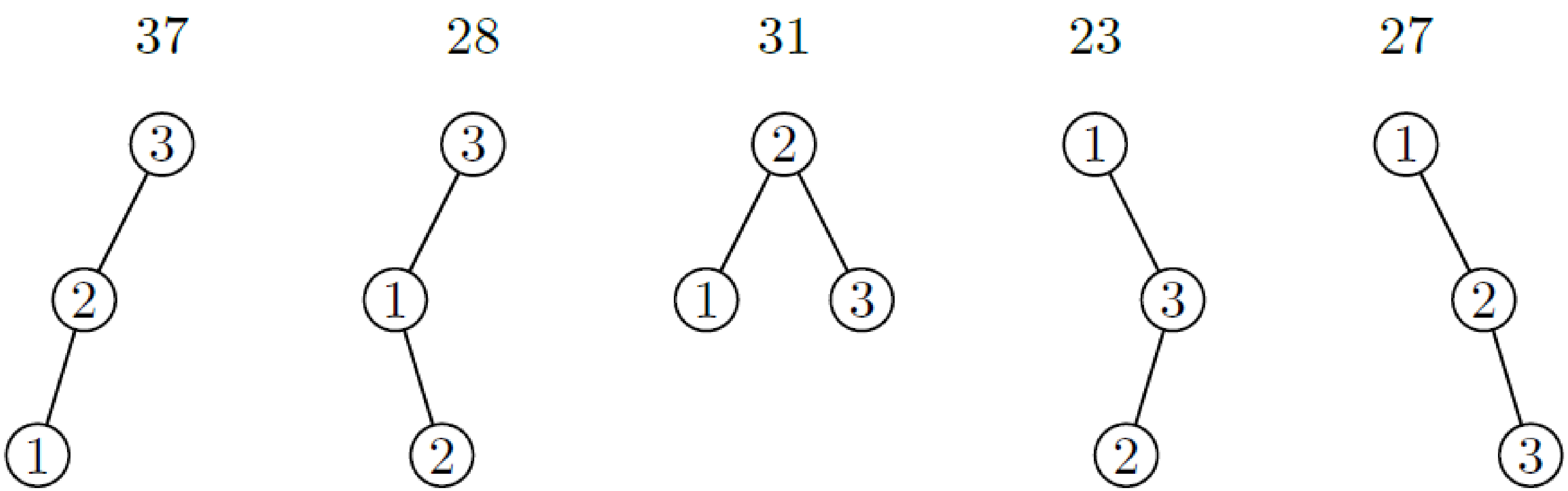
i	v_i	w_i
0	1	1
1	6	2
2	18	5
3	22	6
4	28	7

	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{0}	0	1	1	1	1	1	1	1	1	1	1	1
{0,1}	0	1	6	7	7	7	7	7	7	7	7	7
{0,1,2}	0	1	6	7	7	18	19	24	25	25	25	25
{0,1,2,3}	0	1	6	7	7	18	22	24	28	29	29	40
{0,1,2,3,4}	0	1	6	7	7	18	22	28	29	34	35	40

Optimální BVS

- Optimalizace BVS podle četnosti vyhledávání jednotlivých položek při zachování pravidel BVS
- ***Cíl:*** častěji vyhledávané položky umístit blízko ke kořeni
- ***Cena:*** počet navštívených vrcholů

klíč	Počet vyhledávání
1	10
2	1
3	5



Optimální BVS – rekurzivně

- Máme n položek s klíči $x_1 < \ldots < x_n$ a kladnými váhami $w_1, \ldots, w_n$
- Každému BVS lze přiřadit cenu C:

$$C = \sum_i w_i h_i$$

- kde h_i je hloubka klíče x_i (kořen je v hloubce 1)
- Optimální BVS – BVS s nejnižší cenou (při daném počtu vyhledávání jednotlivých klíčů)
- Do kořene postupně vkládáme všechny možné prvky a rekurzivním voláním téže funkce konstruujeme optimální levý a pravý podstrom.
- Algoritmus s exponenciální časovou složitostí

Optimální BVS – rekurzivně

```
int rOptBVS (int i, int j)
  if i > j:                // prázdný úsek, BVS s cenou 0
    return 0
  else:
    W ← wi + … + wj       // celková váha prvků
    C ← MaxInt                  // inicializace ceny
    for k ← (i,j):              // všechny volby kořene
      cl ← rOptBvs(i, k-1)   // cena levého podstromu
      cr ← rOptBvs(k+1, j)   // cena pravého podstromu
      C ← min(C, cl+cr+W)    // celková nejlepší cena
    return C
```

Optimální BVS – iterativně

- Cena optimálního BVS pro danou n-tici (úsek) klíčů závisí pouze na ceně optimálního BVS pro menší n-tice (úseky).
- Tabulku mezivýsledků můžeme vyplňovat postupně od nejkratších n-tic (úseků) množiny klíčů k nejdelším.
- Pro snadnější rekonstrukci stromu budeme v další tabulce také uchovávat nejlepší možný kořen pro danou n-tici klíčů.
- Algoritmus poběží v čase **$\Theta(n^3)$**

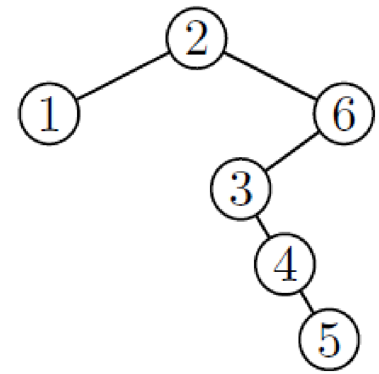
Optimální BVS – iterativně

```
int dpOptBVS (int n, int key[], int w[], int K[])
// počet prvků n, seřazená posloupnost klíčů key,
// odpovídající posloupnost vah w, tabulka kořenů BVS K
  for i ← (1, n+1):                    // prázdné stromy – cena 0
    T[i, i-1] ← 0
  for d ← (1, n):                      // všechny délky úseků
    for i ← (1, n-d+1):                // všechny začátky úseků
      j ← i+d-1                        // konec aktuálního úseku
      W ← w[i] + … + w[j]              // celková váha úseku
      T[i,j] ← MaxInt                  // inicializace ceny
      for k ← (i,j):                   // všechny volby kořene
        C ← T[i,k-1] + T[k+1,j] + W    // cena stromu
        if C < T[i,j]:                 // průběžné minimum
          T[i,j] ← C
          K[i,j] ← k
  return T[1,n]      // výslená cena optimálního BVS
```

$w_1 = 1$
$w_2 = 10$
$w_3 = 3$
$w_4 = 2$
$w_5 = 1$
$w_6 = 9$

T	0	1	2	3	4	5	6
1	0	1	12	18	24	28	52
2	–	0	10	16	22	26	50
3	–	–	0	3	7	10	25
4	–	–	–	0	2	4	16
5	–	–	–	–	0	1	11
6	–	–	–	–	–	0	9
7	–	–	–	–	–	–	0

K	1	2	3	4	5	6
1	1	2	2	2	2	2
2	–	2	2	2	2	2
3	–	–	3	3	3	6
4	–	–	–	4	4	6
5	–	–	–	–	5	6
6	–	–	–	–	–	6



- Konstrukce optimálního BVS
  - Kořenem je prvek s indexem *r = K[1,n]*
  - Jeho levým synem bude kořen optimálního BVS pro úsek *1,…,r-1*, což je prvek s indexem *K[1,r-1]*
  - Jeho pravým synem bude kořen optimálního BVS pro úsek *r+1,…,n*, což je prvek s indexem *K[r+1,n]*
  - Lze snadno implementovat rekurzivním algoritmem s lineární časovou složitostí.

Využití dynamického programování

- Nalezení nejdelší neklesající posloupnosti
- Nalezení nejdelší společné podsekvence dvou řetězců (symboly na sebe nemusí přímo navazovat)
- Viterbiho algoritmus pro skryté Markovovy modely
- Needleman-Wunsch a Smith-Waterman algoritmus pro zarovnání dvou sekvencí
- Bellman-Ford algoritmus pro nalezení nejkratší cesty v ohodnoceném grafu (pracuje i se záporně ohodnocenými hranami)
- Cocke-Kasami-Younger algoritmus pro syntaktickou analýzu
- Held-Karpův algoritmus pro TSP