

Algoritmy

KIV/ADT – 6. přednáška

Miloslav Konopík

21. března 2024

- 1 Hladový algoritmus
- 2 Algoritmus se zpětným návratem
- 3 Rozděl a panuj
- 4 Dynamické programování

Hladový algoritmus

Definice

Hladové algoritmy jsou algoritmy, které na každém kroku dělají lokálně optimální volby k řešení problému. Mohou a nemusí najít globální optimum.

- **Rozměňování mincí:** Chceme sestavit danou sumu při využití minima mincí.
- **Problém výběru aktivit:** Pokud máme množinu aktivit s časy začátku a konce, problémem je vybrat maximální počet aktivit, které může vykonat jedna osoba nebo stroj za předpokladu, že osoba může pracovat pouze na jedné aktivitě najednou.

Hladové algoritmy představují jednoduché a často účinné řešení. Ovšem v řadě případů nedosáhnou optima nebo není garantováno, že zadání vyřeší za všech podmínek

Rozměňování mincí

Úloha:

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



Implementace rozměňování

```
1  # Definice mincí v sestupném pořadí
2  coins: list[int] = [50, 20, 10, 5, 2, 1]

3  def coin_changing(amount: int) -> list[int]:
4      # Inicializace seznamu počtu použitých mincí
5      coin_count: list[int] = [0] * len(coins)

6      # Iterace přes mince v sestupném pořadí
7      for i, coin in enumerate(coins):
8          coin_count[i] = amount // coin
9          amount -= coin_count[i] * coin

10 return coin_count
```

Implementace rozměňování – použití

```
1 counts = coin_changing(67)

2 # Výpis počtu použitých mincí
3 print("Počet použitých mincí:")
4 for i in range(len(coins)):
5     if counts[i] > 0:
6         print(f"{coins[i]} Kč: {counts[i]}x")
```

Výstup:

Počet použitých mincí:

50 Kč: 1x

10 Kč: 1x

5 Kč: 1x

1 Kč: 2x

Rozměňování mincí – omezený počet

Úloha:

- Máme zadanou sumu a úkolem je najít nejmenší množství mincí, které danou sumu vytvoří.
- Každé mince máme zadaný počet kusů.



Implementace rozměňování I

```
1  coins: list[int] = [50, 20, 10, 5, 2, 1]

2  def coin_changing(amount: int, available: list[int]) -> list[int]:
3      # Inicializace seznamu počtu použitých mincí
4      if len(coins) != len(available):
5          raise ValueError("Coins and counts must have the same length.")
6      coin_count: list[int] = [0] * len(coins)

7      # Iterace přes mince v sestupném pořadí
8      for i, coin in enumerate(coins):
9          max_count = min(available[i], amount // coin)
10         coin_count[i] = max_count
11         amount -= max_count * coin

12     return None if amount > 0 else coin_count
```

Implementace rozměňování – použití

```
1 available = [1, 3, 0, 1, 1, 0]
2 counts = coin_changing(67, available)

3 if counts is None:
4     print("Směna se nepodařila!")
5 else:
6     # Výpis počtu použitých mincí
7     print("Počet použitých mincí:")
8     for i in range(len(coins)):
9         if counts[i] > 0:
10            print(f"{coins[i]} Kč: {counts[i]}x")
```

Výstup:

Implementace rozměňování – použití

```
1 available = [1, 3, 0, 1, 1, 0]
2 counts = coin_changing(67, available)

3 if counts is None:
4     print("Směna se nepodařila!")
5 else:
6     # Výpis počtu použitých mincí
7     print("Počet použitých mincí:")
8     for i in range(len(coins)):
9         if counts[i] > 0:
10            print(f"{coins[i]} Kč: {counts[i]}x")
```

Výstup:

Směna se nepodařila!

Implementace rozměňování – použití

```
1 available = [1, 3, 0, 1, 1, 0]
2 counts = coin_changing(67, available)

3 if counts is None:
4     print("Směna se nepodařila!")
5 else:
6     # Výpis počtu použitých mincí
7     print("Počet použitých mincí: ")
8     for i in range(len(coins)):
9         if counts[i] > 0:
10             print(f"{coins[i]} Kč: {counts[i]}x")
```

Chyba!

Výstup:

Směna se nepodařila!

Implementace rozměňování II

```
1  coins: list[int] = [50, 20, 10, 5, 2, 1]

2  def coin_changing(amount: int, available: list[int]) -> list[int]:
3      # Inicializace seznamu počtu použitých mincí
4      counts = [0] * len(coins)
5      available = available.copy()

6      # Iterace přes mince v sestupném pořadí
7      for i in range(len(coins)):
8          while amount >= coins[i] and available[i] > 0:
9              amount -= coins[i]
10             available[i] -= 1
11             counts[i] += 1

12     return None if amount > 0 else counts
```

Analýza chyby

```
1  coins: list[int] = [50, 20, 10, 5, 2, 1]

2  def coin_changing(amount: int, available: list[int]) -> list[int]:
3      # Inicializace seznamu počtu použitých mincí
4      counts = [0] * len(coins)
5      available = available.copy()

6      # Iterace přes mince v sestupném pořadí
7      for i in range(len(coins)):
8          while amount >= coins[i] and available[i] > 0:
9              amount -= coins[i]
10             available[i] -= 1
11             counts[i] += 1

12     return None if amount > 0 else counts
```

Algoritmus se zpětným návratem

- Algoritmus se zpětným návratem – backtracking.
- Rekurzivní backtracking: backtracking s rekurzivní funkcí.
- Backtracking: hrubá síla technika pro hledání řešení. Tato technika je charakterizována schopností vrátit se zpět (“backtrack”), když je potenciální řešení shledáno neplatným.
- Hrubá síla: ne moc chytrý, ale velmi silný algoritmus.
 - Konkrétněji: ne moc efektivní, ale najde platné řešení (pokud platné řešení existuje)
 - Pro řadu úloh velmi efektivní.

Rekurzivní backtracking VS rekurzivní algoritmus

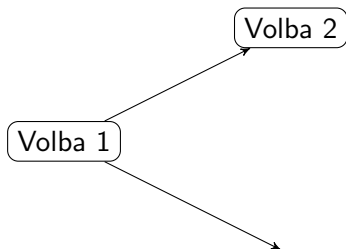
- Běžná otázka: jaký je rozdíl mezi “rekurzí” a rekurzivním backtrackingem”?
- Rekurze: jakákoli metoda, která volá sama sebe k řešení problému.
- Rekurzivní backtracking: konkrétní technika (backtracking), která je vyjádřena pomocí rekurze (backtracking algoritmy mají často rekurzivní podstatu).

Koncept backtracking algoritmu

```
1: function BACKTRACK(solution, candidates)
2:   if ISCOMPLETE(solution) then
3:     OUTPUT(solution)
4:   else
5:     for each c in candidates do
6:       if ISVALID(solution, c) then
7:         ADD(solution, c)
8:         BACKTRACK(solution, candidates)
9:         REMOVE(solution, c)
10:      end if
11:    end for
12:  end if
13: end function
```

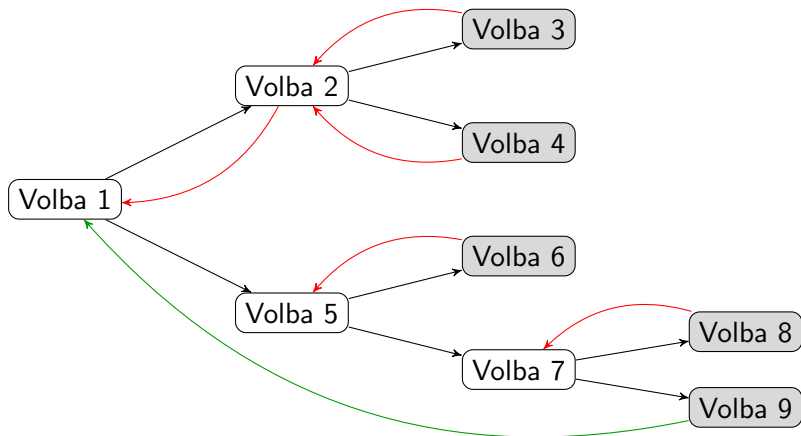
Základní myšlenka I

- Chceme vyzkoušet každou možnost, abychom zjistili, zda je to řešení.
- Pokračujeme dokud nenajdeme správné řešení.
- Můžeme to považovat za sekvenci rozhodnutí. První volba by mohla vypadat nějak takto:
- Co se stane, když vybereme jednu z možností:



Základní myšlenka II

Vyzkoušení všech možností:



Řešení úlohy rozměňování mincí backtrackingem – kód I.

```
1 def coin_changing(amount: int, available: list[int]) -> list[int]:
2     # Inicializace seznamu počtu použitých mincí
3     counts = [0] * len(available)

4     # Rekurzivní funkce pro výpočet počtu použitých mincí
5     def backtrack(amount_left: int, coin_index: int) -> bool:
6         ...

7     # Spuštění rekurzivní funkce
8     success = backtrack(amount, 0)

9     return counts if success else None
```

Řešení úlohy rozměňování mincí backtrackingem – kód II.

```
1  # Rekurzivní funkce pro výpočet počtu použitých mincí
2  def backtrack(amount_left: int, coin_index: int) -> bool:
3      nonlocal counts

4      if amount_left == 0:
5          return True          # Pokud rozměnili celou částku, True

6      # Pokud jsme prošli všechny mince nebo překročili hodnotu, False
7      if coin_index >= len(coins) or amount_left < 0:
8          return False

9      # Iterace přes možné počty aktuální mince
10     for count in reversed(range(available[coin_index] + 1)):
11         if backtrack(amount_left - count * coins[coin_index], coin_index + 1):
12             counts[coin_index] = count
13             return True      # Pokud jsme našli platné řešení, vrátíme True

14     return False
```

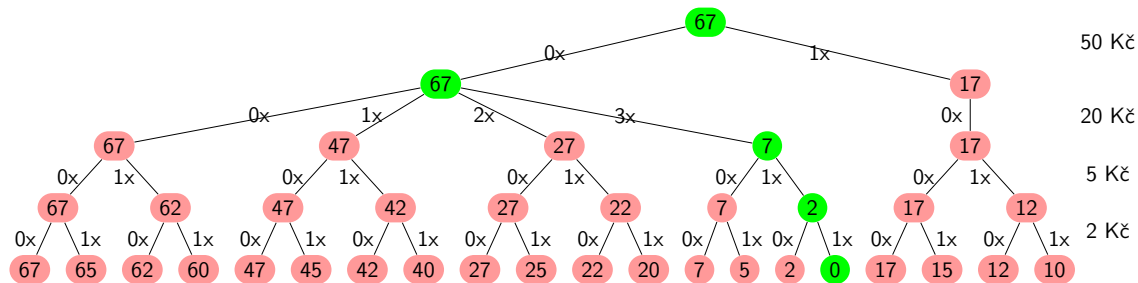
Řešení úlohy rozměňování mincí backtrackingem – kód III.

```
1  # available = [1, 10, 10, 10, 10, 50]
2  available = [1, 3, 0, 1, 1, 100]
3  # available = [1, 3, 0, 1, 1, 0]
4  counts = coin_changing(67, available)

5  if counts is None:
6      print("Směna se nepodařila!")
7  else:
8      # Výpis počtu použitých mincí
9      print("Počet použitých mincí:")
10     for i in range(len(coins)):
11         if counts[i] > 0:
12             print(f"{coins[i]} Kč: {counts[i]}x")
```

Řešení úlohy rozměňování mincí backtrackingem – vizualizace.

Vizualizace pro vstup: 67, [1, 3, 0, 1, 1, 0].



Časová složitost problému rozměňování mincí

Časová funkce $T(n)$ nejhoršího případu algoritmu backtracking pro rozměňování mincí:

- **Počet rekursivních volání:** závisí na hloubce rekurze a větvicím faktoru.
 - Hloubka rekurze: nejvýše celkový počet různých nominálních hodnot mincí
 - Větvicí faktor na každé úrovni: maximální počet dostupných mincí (+1 pro žádnou možnost).
- **Práce provedená na každé úrovni rekurze:** zahrnuje iteraci přes dostupné počty pro každou nominální hodnotu mince.

Časová složitost problému rozměňování mincí

Časová funkce $T(n)$ nejhoršího případu algoritmu backtracking pro rozměňování mincí:

- **Počet rekurzivních volání:** závisí na hloubce rekurze a větvicím faktoru.
 - Hloubka rekurze: nejvýše celkový počet různých nominálních hodnot mincí
 - Větvicí faktor na každé úrovni: maximální počet dostupných mincí (+1 pro žádnou možnost).
- **Práce provedená na každé úrovni rekurze:** zahrnuje iteraci přes dostupné počty pro každou nominální hodnotu mince.

Výpočet $T(n)$:

- Nechť n je počet různých nominálních hodnot mincí.
- Nechť m je maximální počet dostupných mincí pro jakoukoliv nominální hodnotu.
- Časová složitost v nejhorším případě: $T(n, m) = m^n + nm$

Časová složitost problému rozměňování mincí

Časová funkce $T(n)$ nejhoršího případu algoritmu backtracking pro rozměňování mincí:

- **Počet rekursivních volání:** závisí na hloubce rekurze a větvicím faktoru.
 - Hloubka rekurze: nejvýše celkový počet různých nominálních hodnot mincí
 - Větvicí faktor na každé úrovni: maximální počet dostupných mincí (+1 pro žádnou možnost).
- **Práce provedená na každé úrovni rekurze:** zahrnuje iteraci přes dostupné počty pro každou nominální hodnotu mince.

Výpočet $T(n)$:

- Nechť n je počet různých nominálních hodnot mincí.
- Nechť m je maximální počet dostupných mincí pro jakoukoliv nominální hodnotu.
- Časová složitost v nejhorším případě: $T(n, m) = m^n + nm$

Třída složitosti:

$$\Theta(m^n)$$



Rozděl a panuj

Rozděl a panuj.

- Rozděl problém na několik podproblémů (stejného druhu).
- Vyřeš (panuj) každý podproblém rekurzivně.
- Kombinuj řešení podproblémů do celkového řešení.

Nejčastější použití.

- Rozděl problém velikosti n na dva podproblémy velikosti $n/2 : \log_2(n)$ řešení.
- Vyřeš (opanuj) oba podproblémy rekurzivně.
- Kombinuj dva výsledky do celkového řešení. Časová složitost $\Theta(n)$.

Složitost pro náš případ:

- $\Theta(n \log(n))$
- Obecně složitější: viz KIV / ZEP (Master teorém).

Řazení slučováním (Mergesort)

Základní myšlenka

Dvě seřazené posloupnosti lze sloučit do jedné v **lineárním čase**

Řazení slučováním (Mergesort)

Základní myšlenka

Dvě seřazené posloupnosti lze sloučit do jedné v **lineárním čase**

- Vnější řazení: potřebujeme paměťový prostor navíc!

Řazení slučováním (Mergesort)

Základní myšlenka

Dvě seřazené posloupnosti lze sloučit do jedné v **lineárním čase**

- Vnější řazení: potřebujeme paměťový prostor navíc!

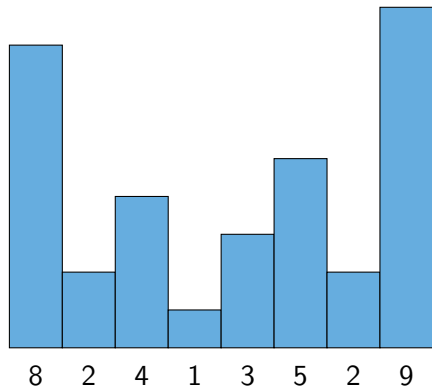
Postup

- procházíme obě posloupnosti současně
 - držíme aktuální index pro obě
- do výsledné posloupnosti zapíšeme vždy menší z dvou aktuálních prvků
- v příslušné posloupnosti se posuneme na další prvek

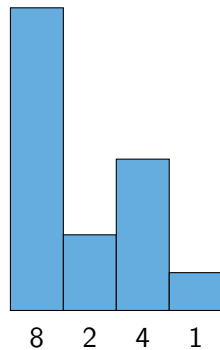
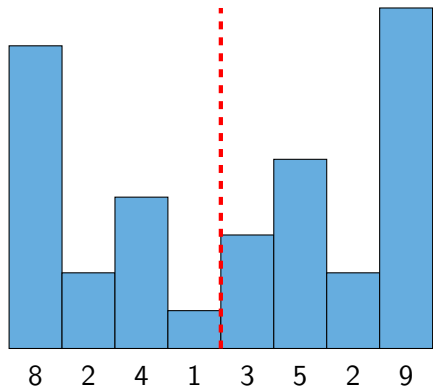
Rekurzivní postup

- je-li vstupní pole jednoprvkové, pak se rovnou vrátí - je seřazené
- v opačném případě se rozdělí na dvě menší pole A, B
- A i B se rekurzivním voláním seřadí
- metodou merge se seřazená pole spojí a výsledek se vrátí

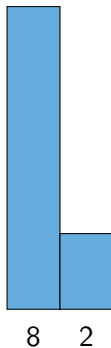
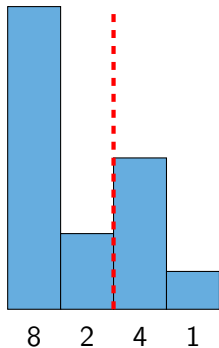
Merge sort – vizualizace



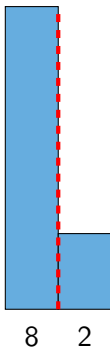
Merge sort – vizualizace



Merge sort – vizualizace



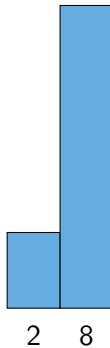
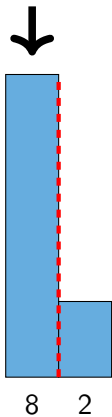
Merge sort – vizualizace



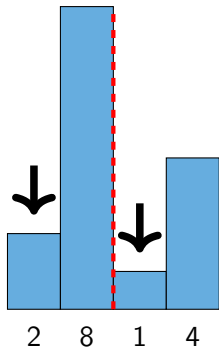
Merge sort – vizualizace



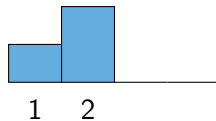
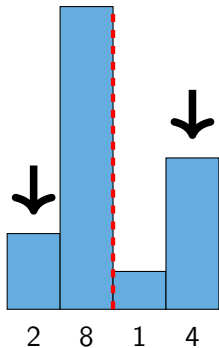
Merge sort – vizualizace



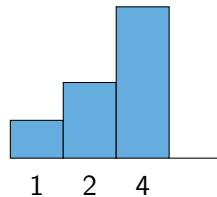
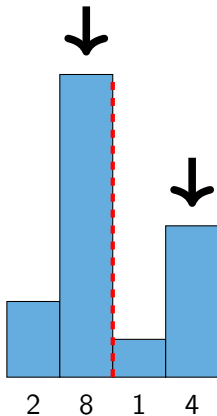
Merge sort – vizualizace



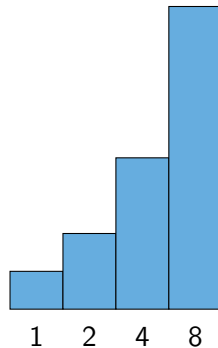
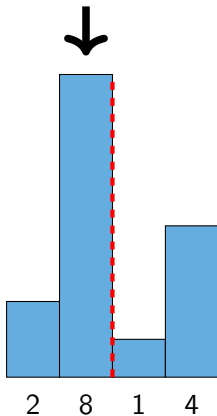
Merge sort – vizualizace



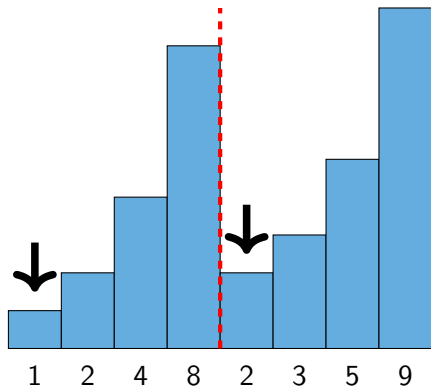
Merge sort – vizualizace



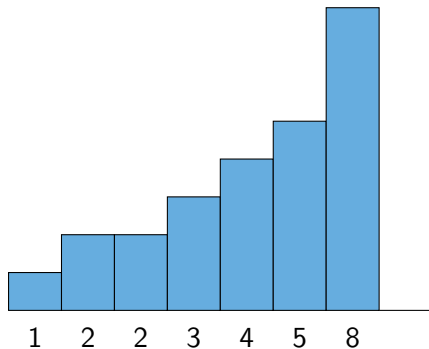
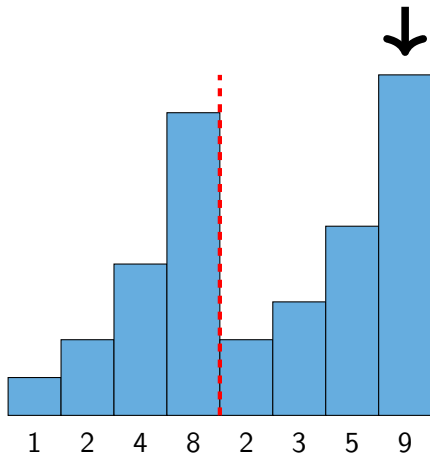
Merge sort – vizualizace



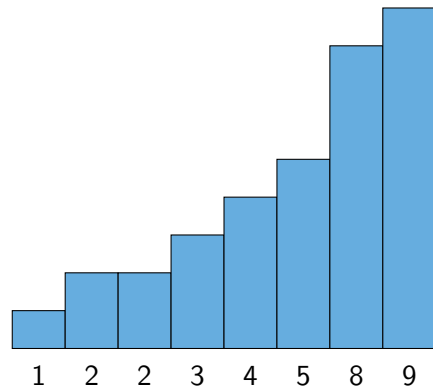
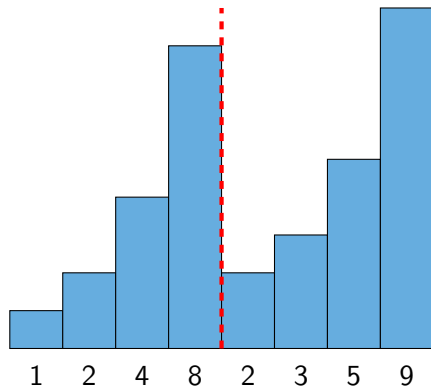
Merge sort – vizualizace



Merge sort – vizualizace



Merge sort – vizualizace



Implementace

```
1 def merge_sort(array: list) -> list:
2     # Pokud je seznam prázdný nebo má jeden prvek, vrátí ho
3     if len(array) <= 1:
4         return array
5     # Najde střední bod seznamu a rozdělí ho na dva podseznamy
6     mid = len(array) // 2 # Střední bod seznamu
7     left = array[:mid] # Levý podseznam
8     right = array[mid:] # Pravý podseznam
9     # Rekurzivně seřadí levý a pravý podseznam
10    left = merge_sort(left)
11    right = merge_sort(right)
12    # Sloučí seřazené levé a pravé podseznamy
13    array = merge(left, right)
14    # Vráť seřazený seznam
15    return array
```


Implementace sloučení (Merge)

```
1  # Definujte funkci pro sloučení dvou seřazených seznamů
2  def merge(left: list, right: list) -> list:
3      # Vytvoř seznam správné velikosti pro uložení výsledku
4      result = [None] * (len(left) + len(right))
5      i, j, k = (0,0,0) # Ukazatel na levý, pravý seznam a na výsledek
6      # Opakuj, dokud nebudou oba seznamy vyčerpány
7      while i < len(left) and j < len(right):
8          # Porovnej aktuální prvky obou seznamů a přidej menší do výsledku
9          if left[i] <= right[j]:
10             result[k] = left[i]
11             i += 1 # Posuň ukazatel na levý seznam o jednu pozici doprava
12         else:
13             result[k] = right[j]
14             j += 1 # Posuň ukazatel na pravý seznam o jednu pozici doprava
15         k += 1 # Posuň ukazatel na výsledek o jednu pozici doprava
16     # Přidejte zbývající prvky z levého nebo pravého seznamu do výsledku
17     ...
```

Implementace sloučení (Merge) – pokračování

```
1 def merge(left: list, right: list) -> list:
2     ...
3     # Přidejte zbývající prvky z levého nebo pravého seznamu do výsledku
4     while i < len(left):
5         result[k] = left[i]
6         i += 1
7         k += 1
8     while j < len(right):
9         result[k] = right[j]
10        j += 1
11        k += 1
12    # Vrat'te výsledný seřazený seznam
13    return result
```

Algoritmická složitost Merge sort

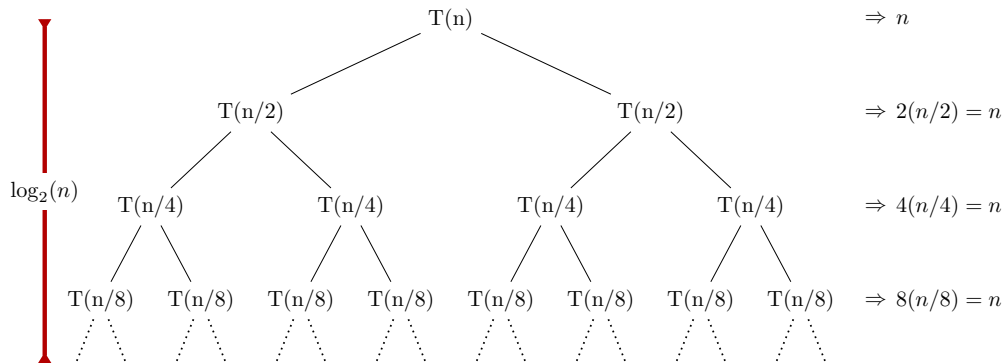
Složitost:

- pole se dělí na poloviny, hloubka zanoření je $h = \log_2(n)$
- na každé úrovni zanoření se celkem vykoná $\mathcal{O}(n)$ operací

=> celková složitost $\mathcal{O}(n \log(n))$.

Algoritmická složitost Merge sort – vizualizace

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$



- Složitost: $\Theta(n \log(n))$.
- Paměťová složitost:
 - $\Theta(n)$,
 - nutná pro uložení seřazené posloupnosti.
- Ve většině implementací se chová stabilně.
- Princip slučování lze použít i pro data, která se nevejdou do paměti nebo pro seřazená data, která přicházejí z více zdrojů.



Dynamické programování

References I