Algoritmy

KIV/ADT – 6. přednáška

Miloslav Konopík

21. března 2024

Obsah

Hladový algoritmus

Hladové algoritmy

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

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

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

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

return coin_count
```

Implementace rozměňování – použití

```python
counts = coin_changing(67)

# Výpis počtu použitých mincí
print("Počet použitých mincí:")
for i in range(len(coins)):
    if counts[i] > 0:
        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

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

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

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

    return None if amount > 0 else coin_count
```

Implementace rozměňování – použití

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

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

Výstup:

Implementace rozměňování – použití

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

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

Výstup:

```
Směna se nepodařila!
```

Implementace rozměňování – použití

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

if counts is None:
    print("Směna se nepodařila!")
else:
    # Výpis počtu použitých m
    print("Počet použitých minci: ")
    for i in range(len(coins)):
        if counts[i] > 0:
            print(f"{coins[i]} Kč: {counts[i]}x")
```

Chyba!

Výstup:

```
Směna se nepodařila!
```

Implementace rozměňování II

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

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

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

    return None if amount > 0 else counts
```

Analýza chyby

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

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

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

    return None if amount > 0 else counts
```

Algoritmus se zpětným návratem

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

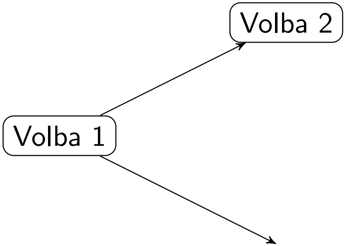
- Běžná otázka: jaký je rozdíl mezi "rekurzí" a rekurzivním backtrackingem"?
- Rekurze: jakákoli metoda, která volá sama sebe k řešení problému.
- Rekurzivní backtracking: konkrétní technika (backtracking), která je vyjádřena pomocí rekurze (backtracking algoritmy mají často rekurzivní podstatu).

Koncept backtracking algoritmů

```
function Backtrack(solution, candidates)
    if IsComplete(solution) then
        Output(solution)
    else
        for each c in candidates do
            if IsValid(solution, c) then
                Add(solution, c)
                Backtrack(solution, candidates)
                Remove(solution, c)
            end if
        end for
    end if
end function
```

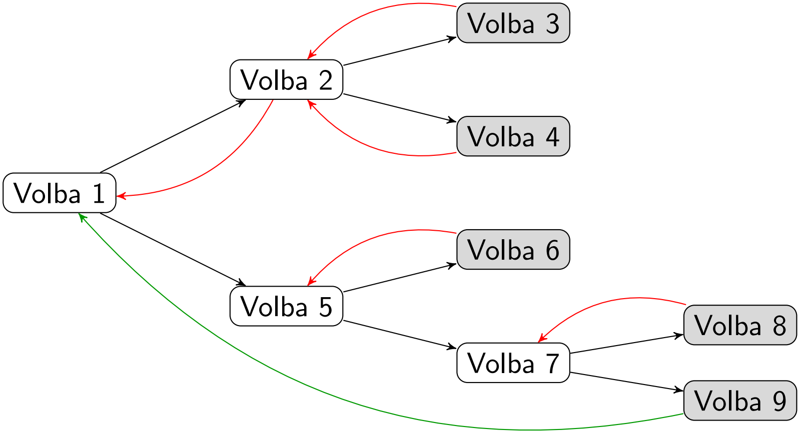
Základní myšlenka I

- 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.

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

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

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

    return counts if success else None
```

Řešení úlohy rozměňování mincí backtrackingem – kód II.

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

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

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

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

    return False
```

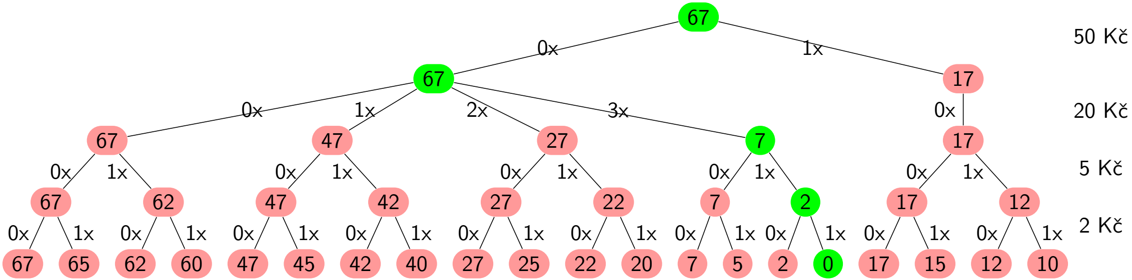
Řešení úlohy rozměňování mincí backtrackingem – kód III.

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

if counts is None:
    print("Směna se nepodařila!")
else:
    # Výpis počtu použitých mincí
    print("Počet použitých mincí:")
    for i in range(len(coins)):
        if counts[i] > 0:
            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 rekurzivních volání:** závisí na hloubce rekurze a větvícím faktoru.
  - Hloubka rekurze: nejvýše celkový počet různých nominálních hodnot mincí
  - Větvící 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 větvícím faktoru.
  - Hloubka rekurze: nejvýše celkový počet různých nominálních hodnot mincí
  - Větvící 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 rekurzivních volání:** závisí na hloubce rekurze a větvícím faktoru.
  - Hloubka rekurze: nejvýše celkový počet různých nominálních hodnot mincí
  - Větvící 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 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

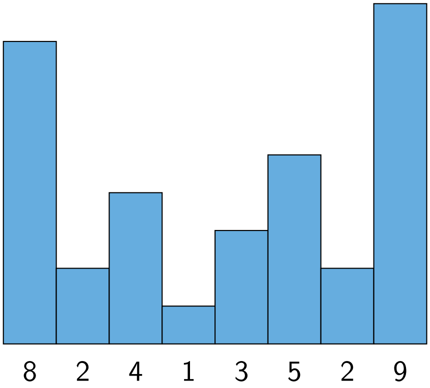
- procházíme obě posloupnosti současně
  - držíme aktuální index pro obě
- do výsledné posloupnosti zapíšeme vždy menší z dvou aktuálních prvků
- v příslušné posloupnosti se posuneme na další prvek

Algoritmus

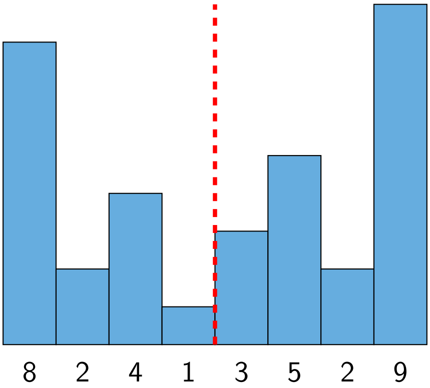
Rekurzivní postup

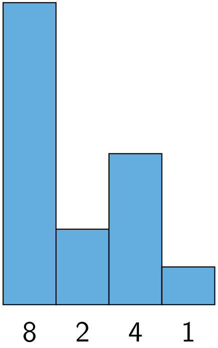
- je-li vstupní pole jednoprvkové, pak se rovnou vrátí - je seřazené
- v opačném případě se rozdělí na dvě menší pole `A`, `B`
- `A` i `B` se rekurzivním voláním seřadí
- metodou `merge` se seřazená pole spojí a výsledek se vrátí

Merge sort – vizualizace

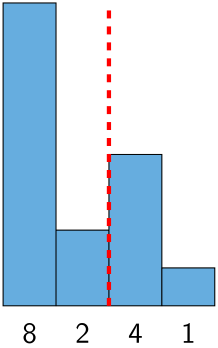


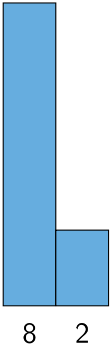
Merge sort – vizualizace



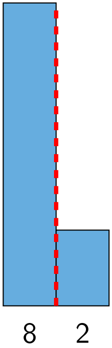


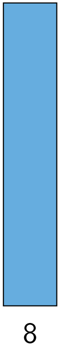
Merge sort – vizualizace





Merge sort – vizualizace



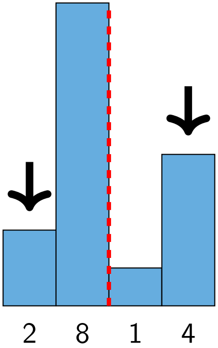


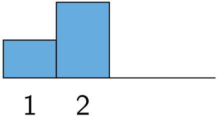
Merge sort – vizualizace

Merge sort – vizualizace

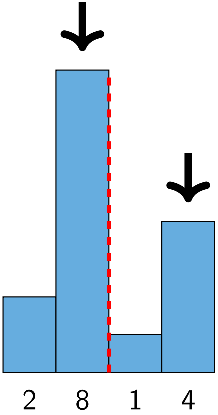
Merge sort – vizualizace

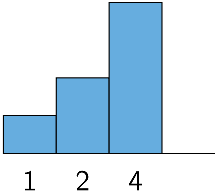
Merge sort – vizualizace



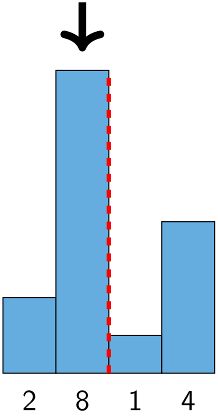


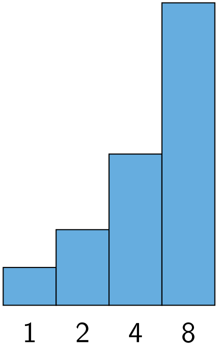
Merge sort – vizualizace



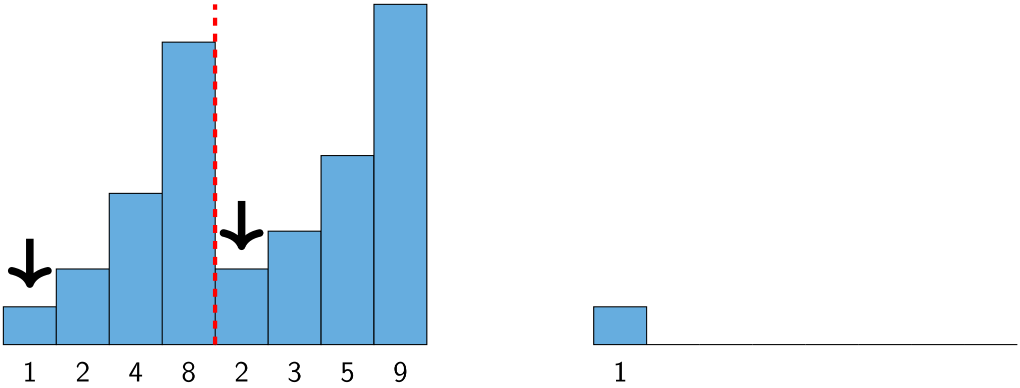


Merge sort – vizualizace

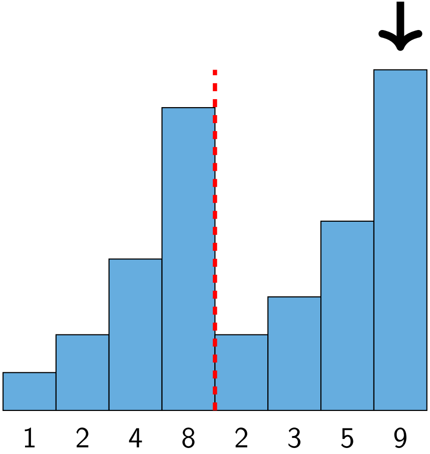


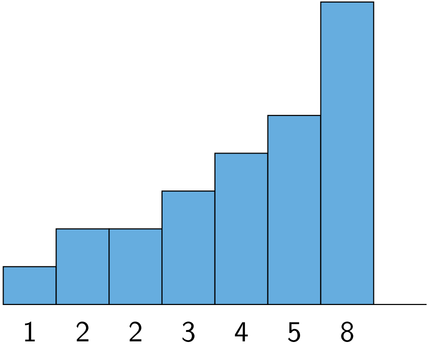


Merge sort – vizualizace

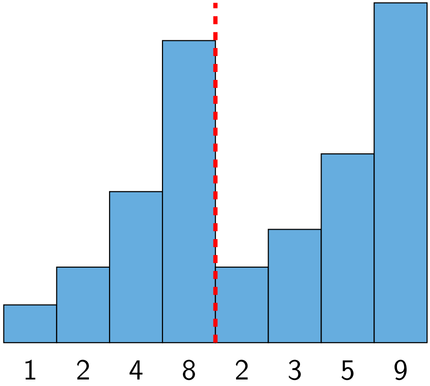


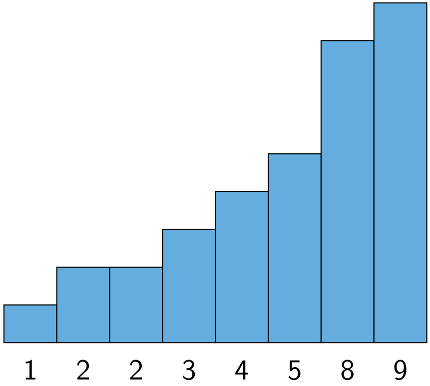
Merge sort – vizualizace





Merge sort – vizualizace





Implementace

```python
def merge_sort(array: list) -> list:
# Pokud je seznam prázdný nebo má jeden prvek, vrátí ho
if len(array) <= 1:
    return array
# Najde střední bod seznamu a rozdělí ho na dva podseznamy
mid = len(array) // 2  # Střední bod seznamu
left = array[:mid]     # Levý podseznam
right = array[mid:]    # Pravý podseznam
# Rekurzivně seřadí levý a pravý podseznam
left = merge_sort(left)
right = merge_sort(right)
# Sloučí seřazené levé a pravé podseznamy
array = merge(left, right)
# Vrátí seřazený seznam
return array
```

Implementace sloučení (Merge)

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

Implementace sloučení (Merge) – pokračování

```python
def merge(left: list, right: list) -> list:
    ...
    # Přidejte zbývající prvky z levého nebo pravého seznamu do výsledku
    while i < len(left):
        result[k] = left[i]
        i += 1
        k += 1
    while j < len(right):
        result[k] = right[j]
        j += 1
        k += 1
    # Vraťte výsledný seřazený seznam
    return result
```

Algoritmická složitost Merge sort

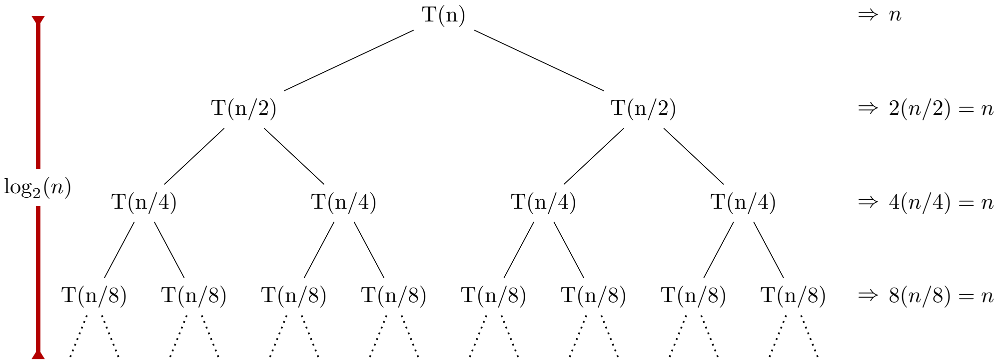
Složitost:

- 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

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



Poznámky

- 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