Konstrukce haldy v lineárním čase

- výchozí rozložení dat představuje úplný binární strom hloubky *d* (bez uspořádání hodnot do haldy)
- nejprve postavíme „haldy“ z podstromů, jejichž kořeny mají hloubku *d*-1, potom pro *d*-2, … atd., až do kořene celé haldy
- stavění hald se provádí záměnami hodnot od kořene k listům (výměna s menším z obou synů)

Časová složitost

hloubka	počet uzlů	max. počet výměn pro každý z nich
0	2^0	d
1	2^1	d-1
…	…	…
j	2^j	d-j
…	…	…
d-1	2^{d-1}	1

Pozorování: Při tomto postupu konstrukce haldy má hodně uzlů malý maximální počet výměn a jen málo z nich může absolvovat výměn hodně $\rightarrow$ celková časová složitost **O(*N*)**.

Celkem se provede výměn (viz tabulka):

$$\sum_{j=0}^{d-1} 2^j \left(d-j\right) = d\sum_{j=0}^{d-1} 2^j - \sum_{j=0}^{d-1} j.2^j =$$

$$= d.\left(2^d - 1\right) - \left(\left(d-2\right).2^d + 2\right) = O\left(2^d\right) = O\left(N\right)$$

… viz dva důkazy matematickou indukcí dále

Důkaz matematickou indukcí č.1: $\sum_{j=0}^{d-1} 2^j = 2^d - 1$

1. pro $d = 1$ zjevně platí

2. nechť platí pro všechna $d < D$, dokazujeme platnost pro $D > 1$:

$$\sum_{j=0}^{D-1} 2^j = \sum_{j=0}^{D-2} 2^j + 2^{D-1} = \left(2^{D-1} - 1\right) + 2^{D-1} = 2.2^{D-1} - 1 = 2^D - 1$$

↑ qed

podle indukčního předpokladu

Důkaz matematickou indukcí č.2: $\sum_{j=0}^{d-1} j.2^j = (d-2).2^d + 2$

1. pro d=1 zjevně platí

2. nechť platí pro všechna $d < D$, dokazujeme platnost pro $D > 1$:

$$\sum_{j=0}^{D-1} j.2^j = \sum_{j=0}^{D-2} j.2^j + (D-1).2^{D-1} = \left((D-3).2^{D-1} + 2\right) + (D-1).2^{D-1} =$$

↑

podle indukčního předpokladu

$$= D.2^{D-1} - 3.2^{D-1} + D.2^{D-1} - 2^{D-1} + 2 = 2.D.2^{D-1} - 4.2^{D-1} + 2 =$$

$$= D.2^D - 2.2^D + 2 = (D-2).2^D + 2$$ qed

Prioritní fronta

- podobné jako fronta, prvky se „předbíhají“ podle svých priorit
- zachování vzájemného pořadí mezi prvky téže priority požadovat můžeme, ale nemusíme (záleží na konkrétní aplikaci)

Možnosti implementace:

- seznam (pole, LSS), do něhož zařazujeme podle priority
- seznam (pole, LSS), z něhož vybíráme podle priority
- samostatné seznamy pro každou hodnotu priority (pokud tyto hodnoty známe a není jich mnoho)
- halda řazená podle priorit – pokud nepožadujeme zachovat pořadí
- halda řazená podle dvojic (priorita, čas příchodu) – pokud požadujeme zachovat vzájemné pořadí mezi prvky téže priority

Slovník

- uchovává dvojice *klíč – hodnota* (klíč je jednoznačný)
- abstraktní datový typ s operacemi
  vyhledej(klíč), vlož(klíč, hodnota) a vymaž(klíč)
- „asociativní pole“
- uložené údaje se vyhledávají podle klíče (indexuje se klíčem)
- klíč může mít jakoukoliv neměnitelnou hodnotu,
  dokonce třeba každý záznam má klíč jiného typu
- některé programovací jazyky přímo podporují
  Python: dictionary
- efektivní implementace je složitější (pomocí hešování)
- časová složitost přístupu k prvku je v průměru konstantní,
  ale v nejhorším případě lineární vzhledem k počtu prvků

Rekurze

- objekt je definován pomocí sebe sama

V programování ve dvou hladinách:

- **rekurzivní algoritmus** – řešení úlohy je definováno pomocí řešení menších instancí téhož problému (tzn. podúloh stejného charakteru)

- **rekurzivní volání funkce** – funkce volá sama sebe
(přímo nebo případně nepřímo prostřednictvím jiných funkcí)

Většinou se rekurzivní algoritmy realizují pomocí rekurzivních volání, ale není to nezbytné:

- rekurzivní algoritmus lze realizovat bez rekurzivních volání (pomocí vlastního zásobníku na uložení rozpracovaných nedokončených podúloh)
$\rightarrow$ více práce pro programátora, program obvykle delší a méně přehledný, výpočet ale může být o něco efektivnější

- rekurzivní volání lze teoreticky použít i při realizaci nerekurzivních iteračních algoritmů (dokonce každý cyklus lze nahradit rekurzivní procedurou)
$\rightarrow$ většinou nevhodné, nečitelný a méně efektivní program

Ukončení rekurze

- rekurzivní volání funkce musí být vázáno na nějakou podmínku, která časem jistě přestane platit → jinak „zacyklení výpočtu“

- na rozdíl od nekonečného while-cyklu dojde v Pythonu k zastavení výpočtu po dosažení maximální povolené hloubky rekurze (1012)

```
def rekurze(p):
    print(p)
    rekurze(p+1)

rekurze(1)
```

Poznámka: programovací jazyky bez limitu na hloubku rekurze
→ běhová chyba přetečení zásobníku (stack overflow)

Průběh výpočtu při rekurzivním volání funkce

- najednou je rozpočítáno více exemplářů téže funkce

- všechny počítají podle téhož kódu

- každý exemplář má na volacím zásobníku svůj vlastní aktivační záznam s lokálními proměnnými, parametry a technickými údaji (kde je rozpočítán, návratová adresa)

- funkce nemá přístup k lokálním proměnným jiného rekurzivního exempláře

Příklad 3: výpis znaků ze vstupu pozpátku

```
def otoc():
    u = input("Znak: ")
    if u != " ":
        otoc()
    print(u)
```

Vstup:
A
B
C
<mezera>

Výstup:
<mezera>
C
B
A

Příklady jednoduchých rekurzivních funkcí

Palindrom – řetězec se čte stejně zleva i zprava (je symetrický)

```
def palindrom1(s):
    n = len(s)
    for i in range(n//2):
        if s[i] != s[n-i-1]:
            return False
    return True

def palindrom2(s):
    if len(s) <=1 :
        return True
    else:
        return s[0] == s[-1] and palindrom2(s[1:-1])
```

Eukleidův algoritmus (odčítací) realizovaný cyklem *(bylo dříve):*

```
def nsd(x, y):
    while x != y:
        if x > y:
            x -= y
        else:
            y -= x
    return x
```

Eukleidův algoritmus realizovaný rekurzivní funkcí
- přesně kopíruje rekurzivní vztah pro NSD,
  na němž je Eukleidův algoritmus založen:

když X > Y	NSD(X ,Y) = NSD(X-Y, Y)
když X < Y	NSD(X ,Y) = NSD(X, Y-X)
když X = Y	NSD(X ,Y) = X

```
def nsd(x, y):
    if x > y:
        return nsd(x-y, y)
    elif x < y:
        return nsd(x, y-x)
    else:
        return x
```

Faktoriál *n!* (součin čísel od 1 do *n*, pro *n* >= 0)

rekurzivní definice: $n! = 1$ pro $n = 0$

$n! = n \cdot (n-1)!$ pro $n > 0$

```
def faktorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f

def faktorial(n):
    if n == 0:
        return 1
    else:
        return n * faktorial(n-1)
```

v obou případech časová složitost O(*n*)

Fibonacciho čísla $F_0 = 0$

$F_1 = 1$

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

rekurzivní definice posloupnosti čísel
→ realizace rekurzivní funkcí přesně podle definice:

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

funkce teoreticky správná, ale časová složitost exponenciální
→ pro *n* > cca 40 prakticky nepoužitelná
důvod: mnohokrát se opakovaně počítají stejné věci

Možnosti řešení:

1. rekurzivní algoritmus + pomocné pole velikosti *n*
   pro uložení již spočítaných funkčních hodnot

   $\rightarrow$ každé F_i se počítá jen jednou $\rightarrow$ časová složitost O(*n*)

   „chytrá rekurze“
   kešování hodnot (cache = mezipaměť)
   memoizace (memory = paměť)
   dynamické programování

2. počítat hodnoty iteračně odspodu – v pořadí F_1, F_2, … F_n
   → časová složitost O(*n*), navíc stačí konstantní paměť

   dynamické programování

```
def fib(n):
    if n == 0:
        return 0
    a = 0; b = 1
    while n > 1:
        a, b = b, a+b
        n -= 1
    return b
```

3. z rekurzivní definice odvodit explicitní vzorec a počítat podle něj

$$F_n = \frac{\sqrt{5}}{5}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

4. využití rychlého umocňování matice

Platí rovnost $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} . \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} b \\ a+b \end{pmatrix}$

Tedy $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} . \begin{pmatrix} F0 \\ F1 \end{pmatrix} = \begin{pmatrix} F1 \\ F2 \end{pmatrix}$

$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} . \begin{pmatrix} F1 \\ F2 \end{pmatrix} = \begin{pmatrix} F2 \\ F3 \end{pmatrix}$

…

$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n . \begin{pmatrix} F0 \\ F1 \end{pmatrix} = \begin{pmatrix} Fn \\ Fn+1 \end{pmatrix}$

Matici $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$ spočítáme rychlým umocňováním

→ časová složitost O(log *N)*

Rekurzivní generování

úlohy typu „zkoušení všech možností“ nebo „generování všech možností“

Vypsat všechna *K*-ciferná čísla v poziční soustavě o základu *n*

Pokud *k* je předem pevně dáno, např pro *k* = 4:

```
for i1 in range(n):
    for i2 in range(n):
        for i3 in range(n):
            for i4 in range(n):
                print(f"{i1} {i2} {i3} {i4}")
```

Je-li *k* vstupním údajem, nemůžeme toto zapsat pomocí vnořených cyklů – nevíme předem, kolik jich máme v programu napsat.

Řešení: rekurzivní funkce obsahující jeden takový cyklus, rekurzivní zanoření jde vždy do hloubky *k*

```
k = 4               # počet cifer
n = 3               # číselná soustava

def cislo(c):
    """ vypíše všechna k-ciferná čísla
        v poziční soustavě o základu "n",
        "c" je vytvářené číslo
    """
    for i in range(n):
        c.append(i)
        if len(c) < k:
            cislo(c)
        else:
            print(c)
        c.pop()

cislo([])
```

Stejné řešení pomocí globálního pole
(v Pythonu implementováno seznamem)

```
k = 4              # počet cifer
n = 3              # číselná soustava
c = [0] * k        # vytvářené číslo

def cislo(p):
    """vypíše všechna k-ciferná čísla
       v poziční soustavě o základu "n",
       "p" je pořadové číslo vybírané cifry
    """
    for i in range(n):
        c[p] = i
        if p < k-1:
            cislo(p+1)
        else:
            print(c)

cislo(0)
```

Podobné řešení: rekurzivní zanoření do hloubky *k*+1

```
k = 4               # počet cifer
n = 3               # číselná soustava
c = [0] * k         # vytvářené číslo

def cislo(p):
    """vypíše všechna k-ciferná čísla
       v poziční soustavě o základu "n",
       "p" je pořadové číslo vybírané cifry
    """
    if p == k:
        print(c)
    else:
        for i in range(n):
            c[p] = i
            cislo(p+1)

cislo(0)
```

Variace s opakováním
k-prvkové z *n*-prvkové množiny {1, 2, …, *n*}

= všechny uspořádané *k*-tice tvořené prvky z {1, 2, …, *n*}
s možností opakování hodnot

Např. pro *k* = 2, *n* = 4: (1,1) (1,2) (1,3) (1,4) (2,1), (2,2) (2,3) (2,4)
(3,1) (3,2) (3,3) (3,4) (4,1), (4,2) (4,3) (4,4)

Řešení: na každou z *k* pozic vytvářené variace
postupně umístíme každé z *n* čísel

→ zcela totéž jako předchozí úloha
(jenom místo čísel 0, …, *n*-1 umisťujeme čísla 1, …, *n*)

Kombinace bez opakování
k-prvkové z *n*-prvkové množiny {1, 2, …, *n*}

= všechny *k*-prvkové podmnožiny vybrané z množiny {1, 2, …, *n*}
(bez možnosti opakování hodnot)

Např. pro *k* = 2, *n* = 4: (1,2) (1,3) (1,4) (2,3) (2,4) (3,4)

Řešení: generujeme pouze ostře rostoucí *k*-tice
hodnot z množiny {1, 2, …, *n*}

```
k = 3                # počet prvků v kombinaci
n = 5                # z kolika prvků vybíráme
c = [0] * (k+1)      # vytvářená kombinace
# technický trik: c[0]=0, kombinace začíná až v c[1]

def kombinace(p):
    """vypíše všechny k-prvkové kombinace
       z "n" prvků bez opakování,
       "p" je pořadové číslo vybíraného prvku
    """
    if p > k:
        print(c[1:])
    else:
        for i in range(c[p-1]+1, n-(k-p)+1):
            c[p] = i
            kombinace(p+1)

kombinace(1)
```