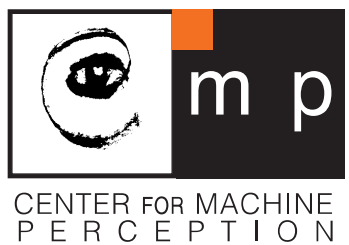
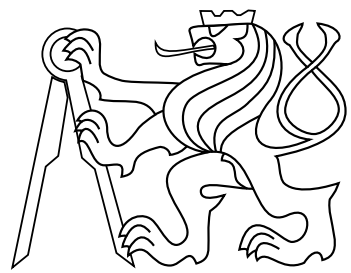
Rekurze a rychlé třídění

Jan Kybic

http://cmp.felk.cvut.cz/~kybic
kybic@fel.cvut.cz

2016–2017





Rekurze

Rychlé třídění

Rekurze
Recursion

- *Rekurze* = odkaz na sama sebe, definice za pomoci sebe sama
- *Rekurzivní funkce* = funkce volá sama sebe *(i nepřímo)*
- Řešení problému za pomoci řešení jednodušší varianty téhož problému
- Obvykle elegantní × je potřeba hlídat efektivitu

Příklad: Umocňování

Přímá definice

$$x^n = \Pi_{i=1}^{n} x = \underbrace{x \cdot x \cdot \cdots \cdot x}_{n\text{-krát}}$$

Příklad: Umocňování

Přímá definice

$$x^n = \Pi_{i=1}^{n} x = \underbrace{x \cdot x \cdot \cdots \cdot x}_{n\text{-krát}}$$

Rekurzivní definice

$$x^0 = 1$$
$$x^{n+1} = x\, x^n \qquad \text{for } n > 0$$

Příklad: Umocňování

Přímá definice

$$x^n = \Pi_{i=1}^{n} x = \underbrace{x \cdot x \cdot \cdots \cdot x}_{n\text{-krát}}$$

Rekurzivní definice

$$x^0 = 1$$
$$x^{n+1} = x\, x^n \qquad \text{for } n > 0$$

Anatomie rekurze

- Základní / bázový případ *(base case)*
- Převedení problému na jednodušší
- Odkaz na sebe / Rekurzivní volání

Příklad: Umocňování — implementace

```
def power_iterative(x,n):
    prod=1.0
    for i in range(n):
        prod*=x
    return prod

print(power_iterative(2.0,10))
```

```
1024.0
```

Příklad: Umocňování — implementace

```
def power_iterative(x,n):
    prod=1.0
    for i in range(n):
        prod*=x
    return prod

print(power_iterative(2.0,10))
```

```
1024.0
```

```
def power_recursive(x,n):
    if n<=0:
        return 1.0
    return x*power_recursive(x,n-1)

print(power_recursive(2.0,10))
```

```
1024.0
```

Příklad: Umocňování — implementace (2)

```
def power_recursive(x,n):
    if n<=0:
        return 1.0
    return x*power_recursive(x,n-1)
```

Stručnější verze

```
def power_recursive2(x,n):
  return x*power_recursive2(x,n-1) if n>0 else 1.0
```

Volte verzi, která je pro vás čitelnější.

Příklad: součet pole

```
def sum_array(a):
  s=0
  for x in a:
    s+=x
  return s

a=[68, 0, 61, 34, 2, 51, 29, 10, 5, 45]
print(sum_array(a))
```

```
305
```

Příklad: součet pole

```
def sum_array(a):
  s=0
  for x in a:
    s+=x
  return s

a=[68, 0, 61, 34, 2, 51, 29, 10, 5, 45]
print(sum_array(a))
```

```
305
```

Šlo by to napsat bez cyklu?

Příklad: součet pole — rekurzivně

```
def sum_array_recursive(a):
  if len(a)==0:
    return 0
  return a[0]+sum_array_recursive(a[1:])

print(sum_array_recursive(a))
```

```
305
```

Každý cyklus lze nahradit rekurzí

Iterativní verze

```
def count_to(n):
  for i in range(1,n+1):
    print(i)

count_to(5)
```

```
1
2
3
4
5
```

Každý cyklus lze nahradit rekurzí (2)

Rekurzivní verze:

```
def count_to_recursive(n):
  count_to_recursive_inner(n,1)

def count_to_recursive_inner(n,i):
  if i<=n:
    print(i)
    count_to_recursive_inner(n,i+1)

count_to_recursive(5)
```

```
1
2
3
4
5
```

Každý cyklus lze nahradit rekurzí (3)

Rekurzivně s vnitřní funkcí

```
def count_to_recursive2(n):
  def count_to_recursive_inner(i):
    if i<=n:
      print(i)
      count_to_recursive_inner(i+1)
  count_to_recursive_inner(1)

count_to_recursive2(5)
```

```
1
2
3
4
5
```

Vnitřní funkce

```
def count_to_recursive2(n):
  def count_to_recursive_inner(i):
    if i<=n:
      print(i)
      count_to_recursive_inner(i+1)
  count_to_recursive_inner(1)
```

+ Skrytí soukromých funkcí
+ Sdílení proměnných vnější funkce
– Nemožnost znovupoužití
– Nemožnost samostatného odladění

Příklad: Řetězec pozpátku

Iterativně

```
def reverse_iterative(s):
  r="" # result
  for i in range(len(s)-1,-1,-1):
    r+=s[i]
  return r

print(reverse_iterative("dobrý večer"))
```

```
rečev ýrbod
```

Příklad: Řetězec pozpátku

Rekurzivně

```
def reverse_recursive(s):
  if len(s)==0:
    return ""
  return reverse_recursive(s[1:])+s[0]

print(reverse_recursive("dobrý večer"))
```

```
rečev ýrbod
```

Příklad: Řetězec pozpátku

Rekurzivně

```
def reverse_recursive(s):
  if len(s)==0:
    return ""
  return reverse_recursive(s[1:])+s[0]

print(reverse_recursive("dobrý večer"))
```

```
rečev ýrbod
```

Kratší verze:

```
def reverse_recursive2(s):
  return "" if  s=="" else reverse_recursive2(s[1:])+s[0]
```

Příklad: Řetězec pozpátku

Rekurzivně

```
def reverse_recursive(s):
  if len(s)==0:
    return ""
  return reverse_recursive(s[1:])+s[0]

print(reverse_recursive("dobrý večer"))
```

```
rečev ýrbod
```

Kratší verze:

```
def reverse_recursive2(s):
  return "" if  s=="" else reverse_recursive2(s[1:])+s[0]
```

‘Pythonská’ verze:

```
print("dobrý večer"[::-1])
```

```
rečev ýrbod
```

Příklad: Číselné soustavy

Převeď číslo n v soustavě se základem b na řetězec.

- $5_{10} = 101_2$ protože $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$.
- Pokud $b > 10$ používáme $\mathrm{A} = 10$, $\mathrm{B} = 11, \ldots$
- Např. $2016_{10} = 7\mathrm{E}0_{16}$ protože
  $7 \cdot 16^2 + 14 \cdot 16^1 + 0 \cdot 16^0 = 2016$

Příklad: Číselné soustavy

Převeď číslo n v soustavě se základem b na řetězec.

- $5_{10} = 101_2$ protože $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$.
- Pokud $b > 10$ používáme $\mathrm{A} = 10$, $\mathrm{B} = 11, \ldots$
- Např. $2016_{10} = 7\mathrm{E}0_{16}$ protože
  $7 \cdot 16^2 + 14 \cdot 16^1 + 0 \cdot 16^0 = 2016$

Myšlenka řešení

- Pokud $n < b$ pak vrať číslici odpovídající n.
- Jinak
  - Najdi `n // b` v soustavě b.
  - Přidej nakonec číslici odpovídající `n % b`

Číselné soustavy — implementace

Vrať `n` jako řetězec v číselné soustavě se základem `base`.

```
def to_str(n, base):
  assert(n>=0)
  cislice = "0123456789ABCDEF"
  if n < base:
    return cislice[n]
  return to_str(n // base, base) + cislice[n % base]

print(to_str(5,2))
```

```
101
```

```
print(to_str(2016,16))
```

```
7E0
```

Číselné soustavy (3)

Nerekurzivní řešení

- Každé rekurzivní řešení je možné napsat bez rekurze — ale může to být těžké.
- Nutnost zapamatovat si lokální proměnné v jednotlivých voláních na *zásobníku (stack)*.

Číselné soustavy (3)

Nerekurzivní řešení

- Každé rekurzivní řešení je možné napsat bez rekurze — ale může to být těžké.
- Nutnost zapamatovat si lokální proměnné v jednotlivých voláních na *zásobníku (stack)*.

```
def to_str_nonrecursive(n,base):
  cislice = "0123456789ABCDEF"
  stack=[n] # hodnoty n
  n//=base
  while n>0:
    stack+=[n]
    n//=base
  result=""
  for m in stack[::-1]:
    result+=cislice[m % base]
  return result
```

Číselné soustavy (4)

Nerekurzivní řešení bez zásobníku

```
def to_str_nonrecursive2(n,base):
  assert(n>=0)
  cislice="0123456789ABCDEF"
  result=""
  while True:
    result=cislice[n % base]+result
    n//=base
    if n==0: break
  return result
```

- Přidávání na začátek řetězce je pomalé (lineární).
- Skrytě kvadratický algoritmus.

Permutace

Vytiskni všechny permutace prvků dané množiny M.

Permutace

Vytiskni všechny permutace prvků dané množiny M.

Myšlenka řešení

- Vezmi každý prvek m_i z M.
  - Najdi všechny permutace prvků $M \backslash \{m_i\}$
  - Ke každé na začátek přidej m_i

Permutace — implementace

```
def tisk_permutaci(m):
  """ Vytiskne všechny permutace prvků v 'm' """
  tisk_permutaci_acc(m,"")

# acc - řetězec přidávaný na začátek
def tisk_permutaci_acc(m,acc):
  if len(m)==0:
    print(acc,end=", ")
  else:
    for i in range(len(m)):
      tisk_permutaci_acc(m[:i]+m[i+1:],acc+m[i]+" ")
```

Permutace — příklad

```
tisk_permutaci(["a","b","c","d"])
```

```
a b c d , a b d c , a c b d , a c d b , a d b c , a d c b ,
b a c d , b a d c , b c a d , b c d a , b d a c , b d c a ,
c a b d , c a d b , c b a d , c b d a , c d a b , c d b a ,
d a b c , d a c b , d b a c , d b c a , d c a b , d c b a ,
```

Příklad: Mince

Vypiš všechny způsoby, jak zaplatit x Kč mincemi v hodnotách $h = \{50, 20, 10, 5, 2, 1\}$ Kč.

Příklad: Mince

Vypiš všechny způsoby, jak zaplatit x Kč mincemi v hodnotách $h = \{50, 20, 10, 5, 2, 1\}$ Kč.

Myšlenka řešení

- Vyber největší minci $h_i \leq x$. Pak jsou dvě možnosti:
  - Použij h_i — zaplať $x - h_i$ pomocí $h_i, h_{i+1}, \ldots, h_n$ a přidej jednu h_i.
  - Nepoužij h_i — zaplať x pomocí $h_{i+1}, \ldots, h_n$

Mince — implementace

Vytiskni všechny možné způsoby, jak zaplatit x Kč.

```
def zaplat(x):
  h=[50,20,10,5,2,1] # hodnoty mincí sestupně
  def doplat(x,m,i):
    """ m - kolik zaplaceno v počtech mincí
        i - kterou mincí začít """
    if x==0:
      vytiskni_platbu(m,h)
    else:
      if x>=h[i]: # zaplať minci h[i]
        doplat(x-h[i],m[:i]+[m[i]+1]+m[i+1:],i)
      if i<len(h)-1: # zaplať menšími, lze-li
        doplat(x,m,i+1)
  doplat(x,len(h)*[0],0) # tohle je začátek fce zaplat
```

Mince — implementace (2)

```
def vytiskni_platbu(m,h):
  """ m - počty mincí, h - hodnoty """
  for j in range(len(h)):
    if m[j]>0:
       print("%3d*%3dKč" % (m[j],h[j]), end="")
  print("")
```

Mince — zkouška

```
zaplat(12)
```

```
 1* 10Kč  1*  2Kč
 1* 10Kč  2*  1Kč
 2*  5Kč  1*  2Kč
 2*  5Kč  2*  1Kč
 1*  5Kč  3*  2Kč  1*  1Kč
 1*  5Kč  2*  2Kč  3*  1Kč
 1*  5Kč  1*  2Kč  5*  1Kč
 1*  5Kč  7*  1Kč
 6*  2Kč
 5*  2Kč  2*  1Kč
 4*  2Kč  4*  1Kč
 3*  2Kč  6*  1Kč
 2*  2Kč  8*  1Kč
 1*  2Kč 10*  1Kč
12*  1Kč
```

Mince — rychlejší varianta

```
def zaplat2(x):
  h=[50,20,10,5,2,1] # hodnoty mincí sestupně
  def doplat(x,m,i):
    """ m - kolik zaplaceno v počtech mincí
        i - kterou mincí začít """
    if x==0:
      vytiskni_platbu(m,h)
    else:
      if x>=h[i]: # zaplat minci h[i]
        m[i]+=1
        doplat(x-h[i],m,i)
        m[i]-=1  # úklid
      if i<len(h)-1: # zaplať menšími
        doplat(x,m,i+1)
  doplat(x,len(h)*[0],0)
```

Mince — zkouška (2)

```
zaplat2(12)

  1* 10Kč  1*  2Kč
  1* 10Kč  2*  1Kč
  2*  5Kč  1*  2Kč
  2*  5Kč  2*  1Kč
  1*  5Kč  3*  2Kč  1*  1Kč
  1*  5Kč  2*  2Kč  3*  1Kč
  1*  5Kč  1*  2Kč  5*  1Kč
  1*  5Kč  7*  1Kč
  6*  2Kč
  5*  2Kč  2*  1Kč
  4*  2Kč  4*  1Kč
  3*  2Kč  6*  1Kč
  2*  2Kč  8*  1Kč
  1*  2Kč 10*  1Kč
 12*  1Kč
```

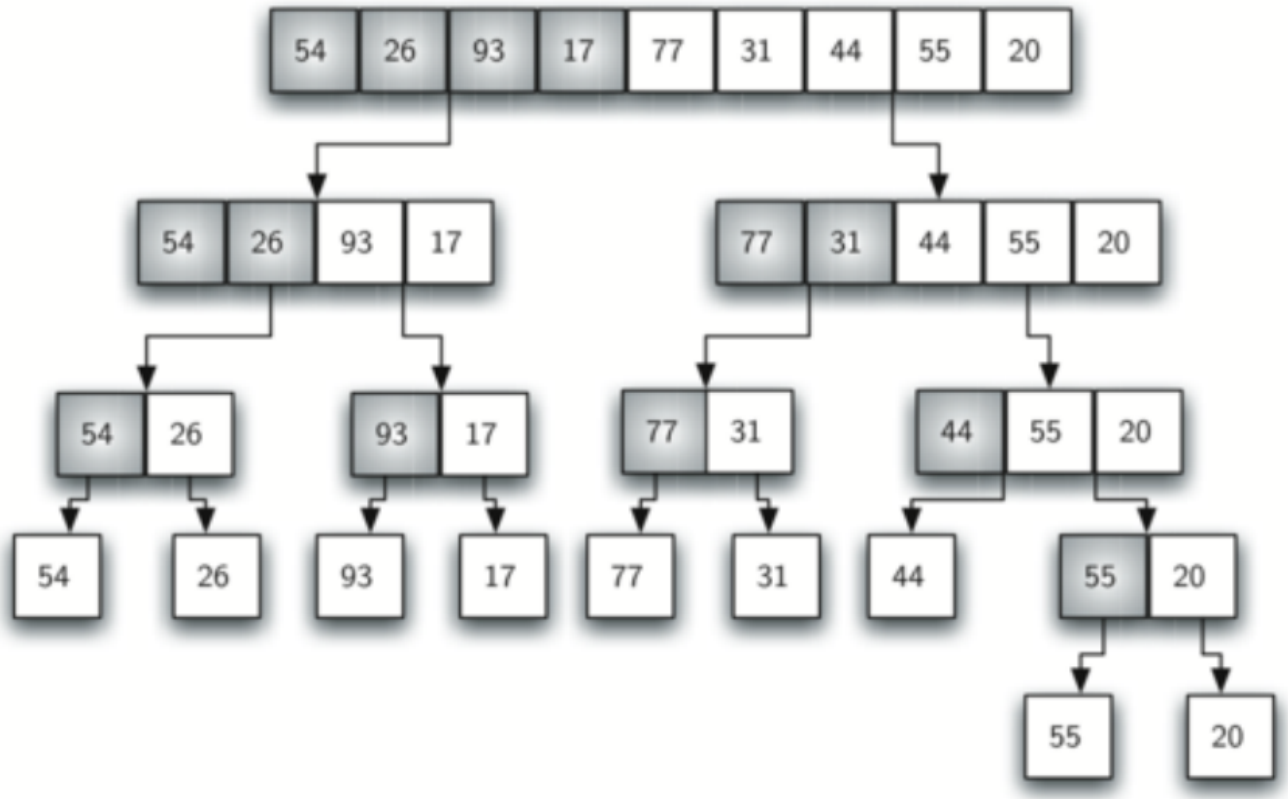
Rychlé třídění

Setřiď vzestupně pole a.

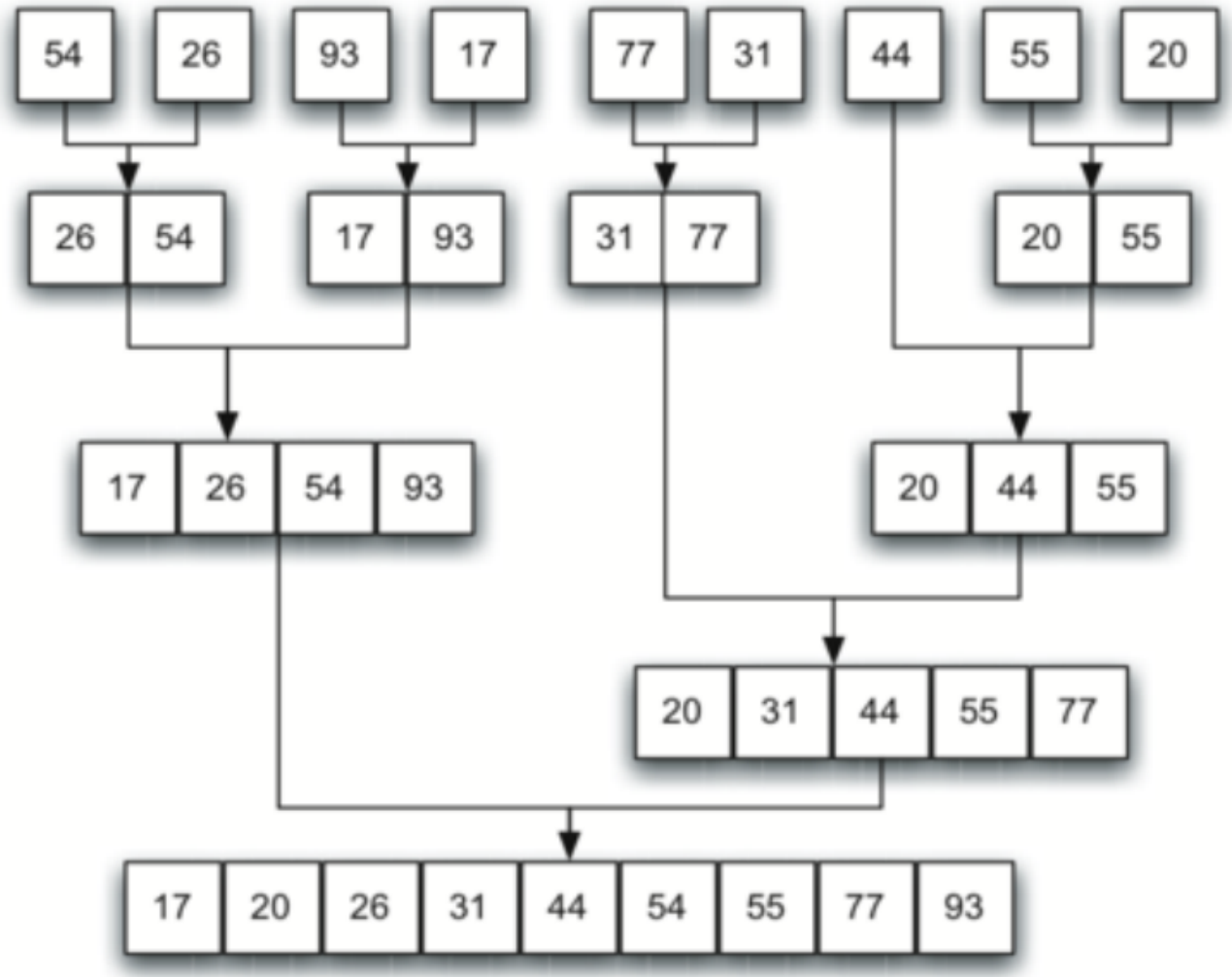
Základní myšlenka

- Rozděl pole a na dvě poloviny
- Každou polovinu setřiď (rekurzivně)
- Spoj obě setříděné poloviny

Merge sort (2) — rozdělování
(splitting)



Merge sort (3) — spojování
(joining)



Merge sort (4) — implementace

Rozdělování

```
def merge_sort(a):
  """ Setřídí pole 'a' pomocí merge sort """
  if len(a)<=1:    # triviální případ
    return a
  mid=len(a)//2    # rozdělení pole na dvě
  left=merge_sort(a[:mid])
  right=merge_sort(a[mid:])
  return join_arrays(left,right)
```

Merge sort (4) — implementace

Spojování

```
def join_arrays(left,right):
  """ Dvě vzestupně setříděná pole spojí do jednoho """
  result=[] # spojení dvou setříděných polí
  i=0       # index do 'left'
  j=0       # index do 'right'

  while i<len(left) and j<len(right):
    if left[i]<right[j]:
      result+=[left[i]]
      i+=1
    else:
      result+=[right[j]]
      j+=1

  result+=left[i:]    # doplnit zbytky
  result+=right[j:]
  return result
```

Merge sort (6) — příklad

```
a=[random.randrange(100) for i in range(10)]
print(a)
print(merge_sort(a))
```

```
[15, 43, 3, 60, 38, 38, 57, 30, 82, 0]
[0, 3, 15, 30, 38, 38, 43, 57, 60, 82]
```

Merge sort (5) — vlastnosti

- *Merge sort* potřebuje pomocná pole (není *in place*).
- *Merge sort* je stabilní.

Merge sort (5) — vlastnosti

- *Merge sort* potřebuje pomocná pole (není *in place*).
- *Merge sort* je stabilní.

```
def merge_sort_inplace(a):
  """ setřídí pole 'a' na místě """
  a[:]=merge_sort(a)
```

Merge sort (6) — složitost

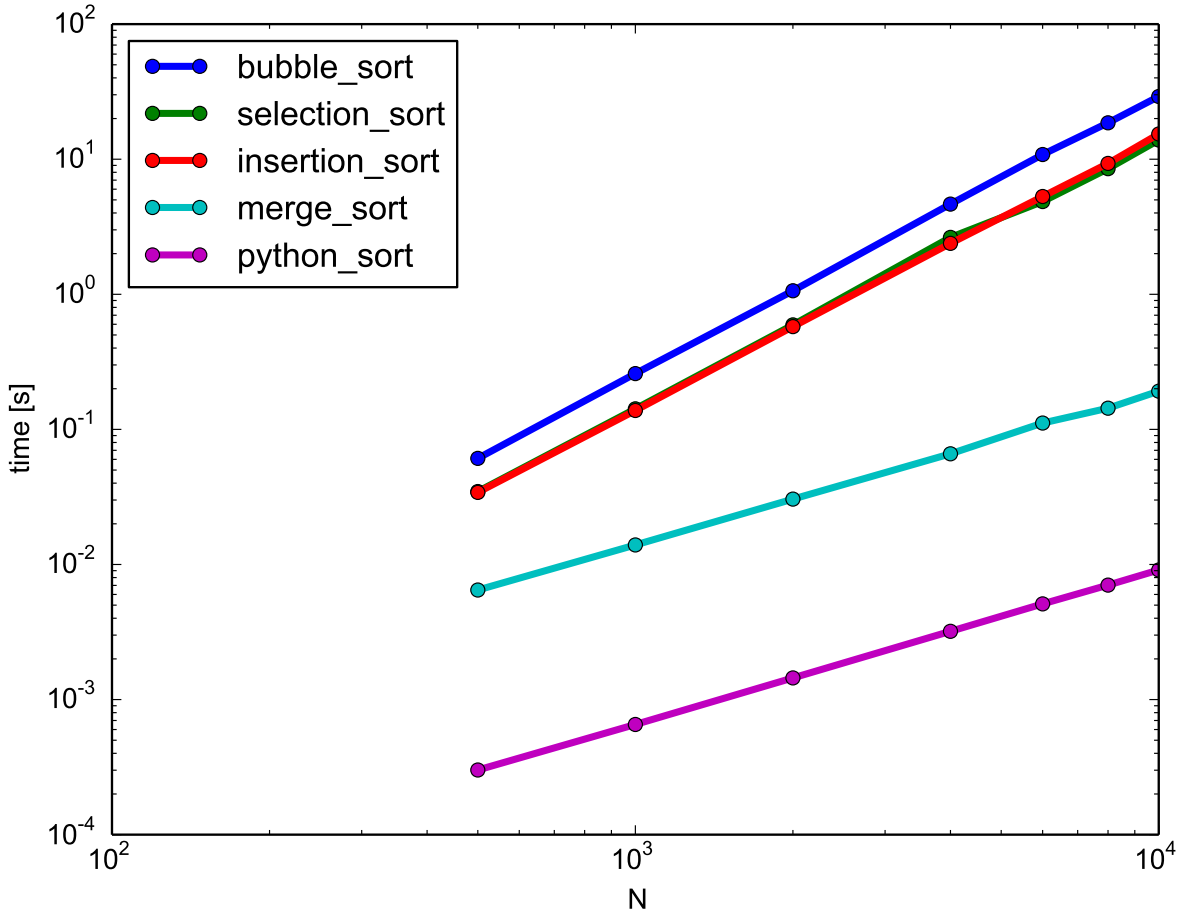
Složitost setřídění n prvků

- Rozdělení lze provést v čase $O(n)$ v každé úrovni rekurze.
- Spojování lze provést v čase $O(n)$ v každé úrovni rekurze.
- Úrovní rekurze je $\sim \log_2 n$.

Složitost algoritmu *merge sort* je $O(n \log n)$.

Merge sort (6) — porovnání rychlosti

Empirická složitost



Quick sort

Setřiď vzestupně pole a.

Základní myšlenky a vlastnosti

- Vybere jeden prvek p z pole a — *pivot* .
- Provedeme částečné setřídění *(partitioning)*:

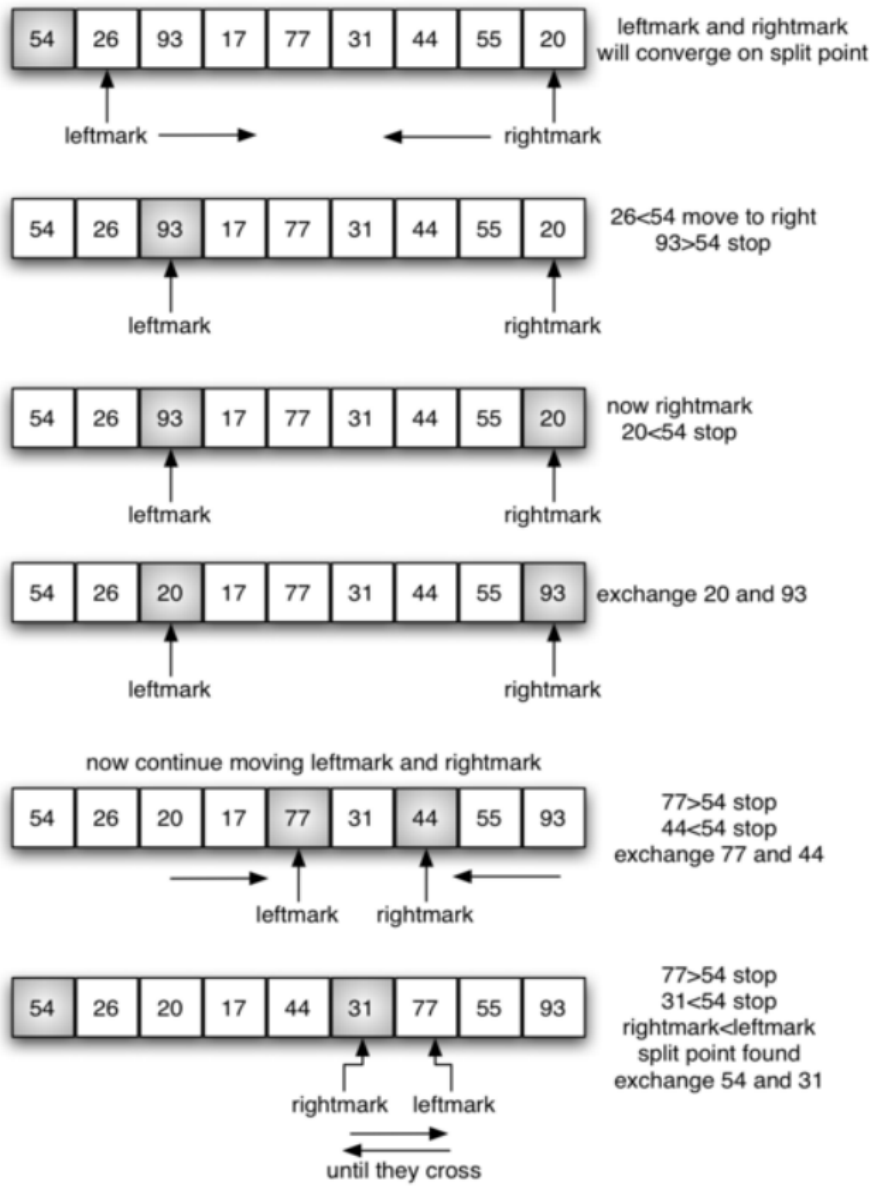
$$[\underbrace{\ldots a_i \ldots}_{a_i \leq p}\ p\ \underbrace{\ldots a_j \ldots}_{a_j \geq p}]$$

- Částeční setřídění lze udělat v lineárním čase, $O(n)$.
- Rekurzivně setřídíme $[\ldots a_i \ldots]$ a $[\ldots a_j \ldots]$.

- Třídění na místě, bez pomocného pole *(in place)*.
- Není stabilní.

Quick sort (2) — částečné setřídění

Partitioning



Quick sort (3) — implementace

```
def quick_sort(a):
    """ Setřídí pole a na místě """
    quick_sort_helper(a,0,len(a)-1)

def quick_sort_helper(a,first,last):
    """ setřídí podpole a[first]..a[last] """
    if first < last:
        m = partition(a,first,last) # vrátí index pivotu
        quick_sort_helper(a,first,m-1)
        quick_sort_helper(a,m+1,last)
```

Quick sort (4) — implementace

Partitioning

Změní a a vrátí index m takový, že všechny prvky před m jsou menší než a_m a všechny prvky po m jsou větší než a_m.

```
def partition(a,first,right):
  pivot=a[first] # pivot je první prvek
  left=first+1
  while True:
    while left<=right and a[left] <= pivot:
      left+=1      # najdi první zleva větší než pivot
    while left<=right and a[right] >= pivot:
      right-=1     # najdi první zprava menší než pivot
    if right<left:                              # končíme?
      a[first],a[right]=a[right],a[first]   # pivot na místo
      return right
    a[left],a[right]=a[right],a[left] # výměna
```

Quick sort (5) — příklad

```
a=[random.randrange(100) for i in range(10)]
print(a)
```

```
[63, 73, 8, 16, 44, 63, 9, 89, 64, 58]
```

```
quick_sort(a)
print(a)
```

```
[8, 9, 16, 44, 58, 63, 63, 64, 73, 89]
```

Quick sort (6) — složitost

- Částeční setřídění lze udělat v lineárním čase, $O(n)$. To platí pro všechna volání na dané úrovni rekurze.
- Úrovní rekurze je v průměru $\sim \log_2 n$.
- Průměrná složitost *quick sort*u je $O(n \log n)$.

Quick sort (6) — složitost

- Částeční setřídění lze udělat v lineárním čase, $O(n)$. To platí pro všechna volání na dané úrovni rekurze.
- Úrovní rekurze je v průměru $\sim \log_2 n$.
- Průměrná složitost *quick sort*u je $O(n \log n)$.

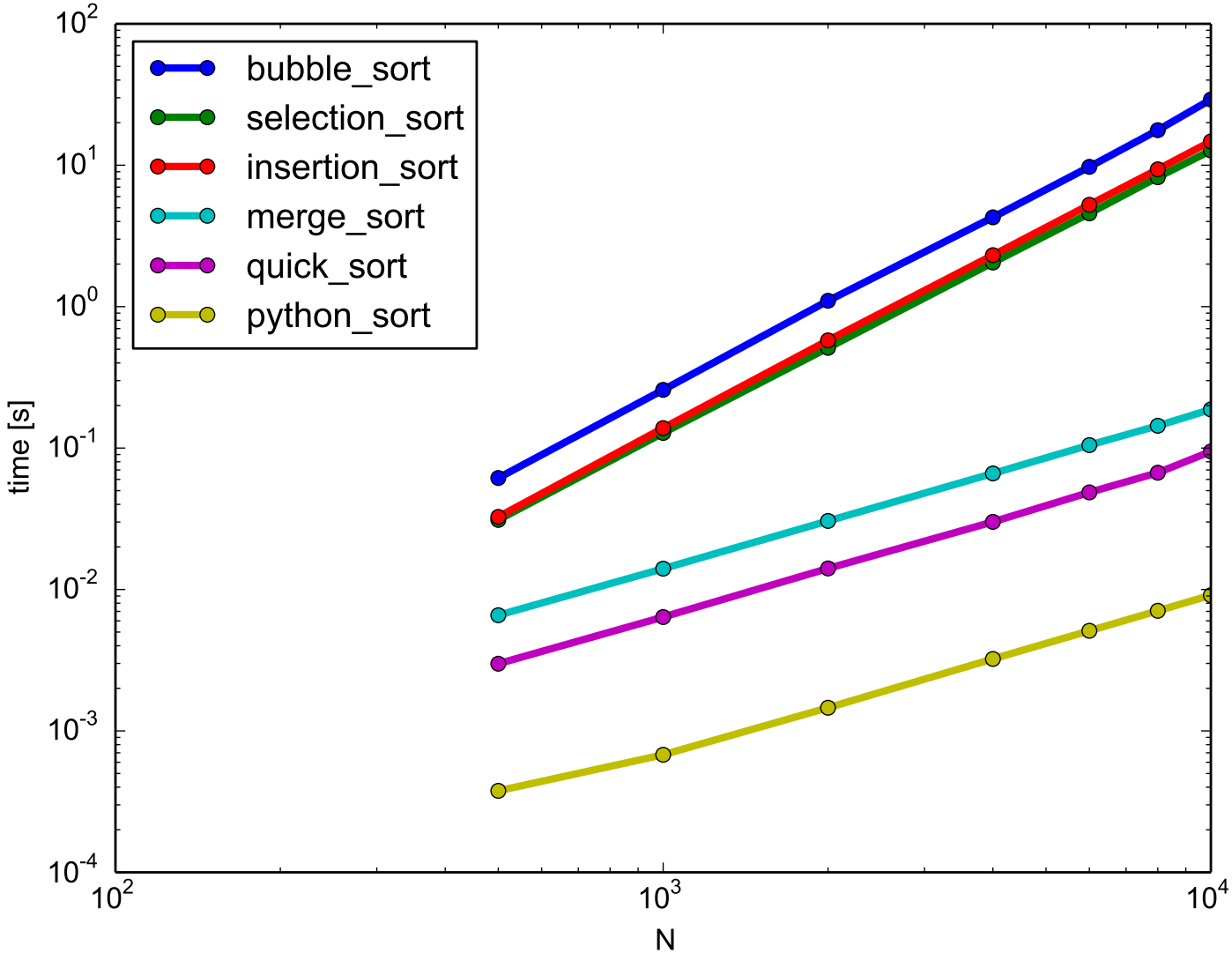
ale...

- Úrovní rekurze je v nejhorším případě $\sim n \quad \rightarrow \quad$ časová složitost v nejhorším případě je tedy $O(n^2)$.
- Nejhorším případem naší implementace je setříděná posloupnost.
- Sofistikovanější výběr pivotů pravděpodobnost degenerace podstatně sníží.

Quick sort je v průměru jeden z nejrychlejších obecných třídících algoritmů, rychlejší než např. *merge sort*. Časová složitost v nejhorším případě je ale horší.

Quick sort (6) — porovnání rychlosti

Empirická složitost



Složitost problému třídění

Otázky:

- Nešlo by najít ještě lepší třídící algoritmus?
- Jaká je složitost *problému třídění*?

Složitost problému třídění

Otázky:

- Nešlo by najít ještě lepší třídící algoritmus?
- Jaká je složitost *problému třídění*?

Výpočetní model: Porovnávací třídící algoritmus *(compare-based sorting algorithm)* získává informace pouze porovnáváním prvků.

- Úkolem je najít permutaci $k_1, \ldots, k_n$ indexů $1, \ldots, n$, tak aby $a_{k_1} \leq a_{k_2} \leq \cdots \leq a_{k_n}$.
- Kolik porovnání $a_i \leq a_j$ je potřeba?

Dolní odhad složitosti problému třídění

- Je 2^p možných výsledků p porovnání.
- Je $n!$ permutací n prvků.

Musí platit

$$2^p \geq n!$$
$$p \geq \log_2 n!$$

Dolní odhad složitosti problému třídění

- Je 2^p možných výsledků p porovnání.
- Je $n!$ permutací n prvků.

Musí platit

$$2^p \geq n!$$
$$p \geq \log_2 n!$$

☝ Neexistuje žádný porovnávací třídící algoritmus, který pro setřídění n prvků potřeboval vždy méně než $\log_2 n!$ porovnání

Optimální asymptotická složitost problému třídění

Platí, že

$$n \log \frac{n}{\mathrm{e}} + 1 \leq \log n! \leq (n+1) \log \frac{n+1}{\mathrm{e}} + 1$$

tedy asymptoticky $\log n! \sim n \log n$ a $\log_2 n! \sim n \log_2 n$.

Optimální asymptotická složitost problému třídění

Platí, že

$$n \log \frac{n}{\mathrm{e}} + 1 \leq \log n! \leq (n+1) \log \frac{n+1}{\mathrm{e}} + 1$$

tedy asymptoticky $\log n! \sim n \log n$ a $\log_2 n! \sim n \log_2 n$.

☝ Neexistuje žádný porovnávací třídící algoritmus, jehož asymptotická časová složitost v nejhorším případě by byla lepší než $O(n \log n)$.

Optimální asymptotická složitost problému třídění

Platí, že

$$n \log \frac{n}{e} + 1 \leq \log n! \leq (n+1) \log \frac{n+1}{e} + 1$$

tedy asymptoticky $\log n! \sim n \log n$ a $\log_2 n! \sim n \log_2 n$.

☝ Neexistuje žádný porovnávací třídící algoritmus, jehož asymptotická časová složitost v nejhorším případě by byla lepší než $O(n \log n)$.

Poznámky:

- Optimální časovou složitost má *merge sort* (třídění spojováním), *heap sort*,. . .
- Algoritmus *quick sort* je často v praxi rychlejší, ale má optimální časovou složitost pouze v průměru, složitost nejhoršího případu je $O(n^2)$.
- Tato analýza platí pouze pro porovnávací algoritmy.

Vlastnosti

- Třídění n přirozených čísel (nebo řetězců) pevné délky k.
- Časová složitost $O(nk)$.
- Může být implementován jako stabilní. Potřebuje pomocnou paměť.

Radix sort
Číslicové/přihrádkové třídění

Vlastnosti

- Třídění n přirozených čísel (nebo řetězců) pevné délky k.
- Časová složitost $O(nk)$.
- Může být implementován jako stabilní. Potřebuje pomocnou paměť.

Základní myšlenka

- Mějme přihrádku pro každou možnou číslici ($0 \ldots 9$).
- Postupně od nejméně významného řádu
  - Každý prvek přidáme do přihrádky dle číslice daného řádu.
  - Obsah přihrádek za sebe zřetězíme v pořadí dle hodnoty číslic.

Radix sort — implementace (1)

Pomocné funkce:

```
def num_digits(a):
  """ počet číslic v desítkovém zápisu čísla a """
  num=1
  while a>10:
    a//=10
    num+=1
  return num

def digit(a,n):
  """ n-tá číslice zprava čísla 'a', počítáno od nuly"""
  return a//(10**n) % 10
```

Soubor `radix_sort_experiments.py`

Radix sort — implementace (2)

Hlavní smyčka: `radix_sort_integers` setřídí pole přirozených čísel s omezeným počtem číslic

```
def radix_sort_integers(a):
  max_digits=num_digits(max(a)) # max. počet číslic
  for i in range(max_digits):
    a=sort_by_digit(a,i)
  return a
```

Radix sort — implementace (2)

Vnitřní smyčka:

```
def sort_by_digit(a,i):
  """ setřídí pole 'a' dle i-té číslice """
  prihradka=[ [] for j in range(10) ]
  for x in a:     # rozhoď do přihrádek
    c=digit(x,i) # číslice pro třídění
    prihradka[c]+=[x]
  r=[]            # spoj přihrádky
  for p in prihradka:
    r+=p
  return r
```

Časová složitost $O(nk)$.

Radix sort — příklad

```
a=[random.randrange(100) for i in range(10)]
print(a)
```

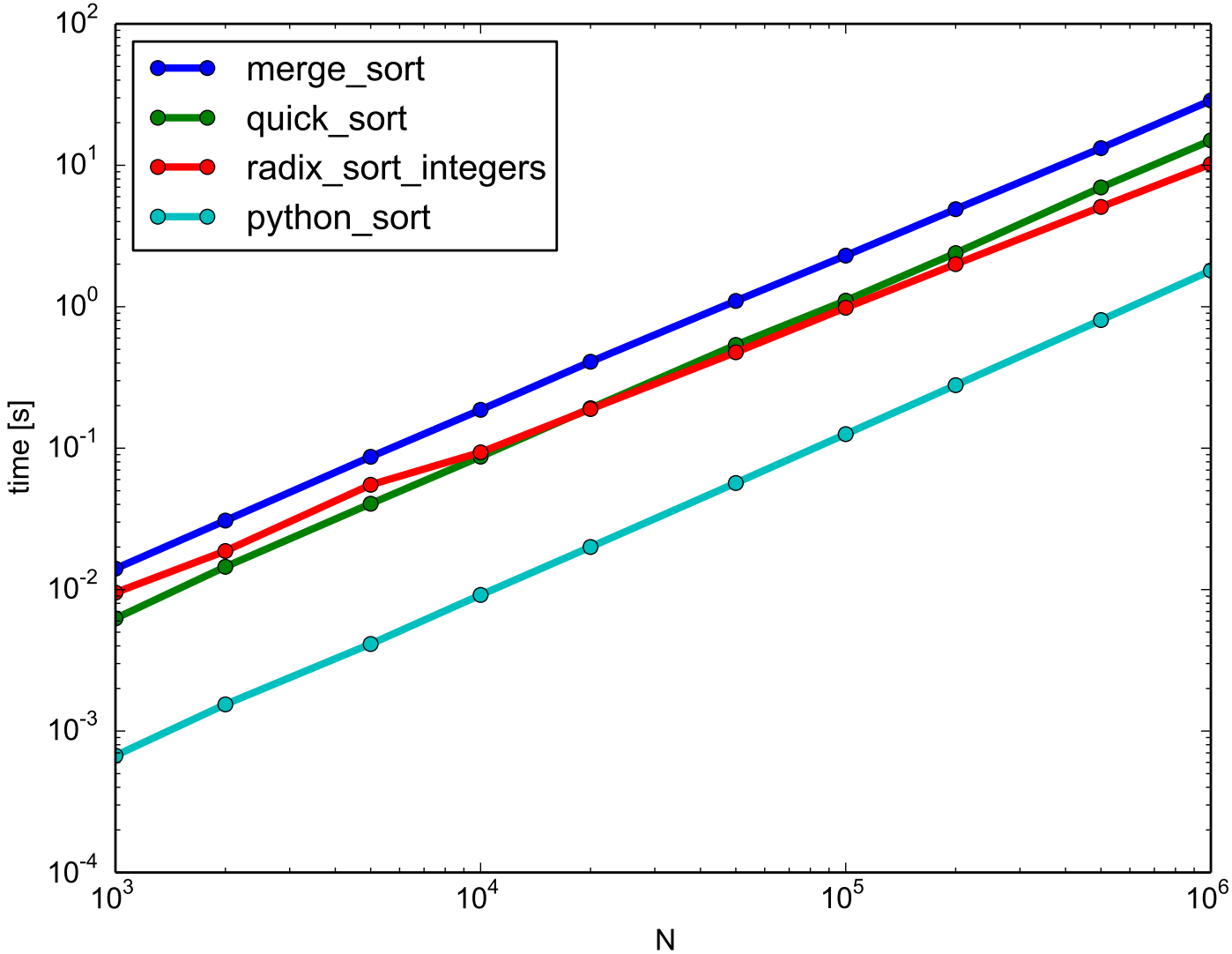
```
[47, 71, 96, 75, 62, 23, 9, 68, 86, 10]
```

```
print(radix_sort_integers(a))
```

```
[9, 10, 23, 47, 62, 68, 71, 75, 86, 96]
```

Radix sort — porovnání rychlosti

Empirická složitost



Shrnutí

Rekurze

- Rekurzivní a iterativní formulace jsou ekvivalentní.
- Rekurzivní formulace bývá elegantní a stručná.
- Rekurzivní implementace může být neefektivní.

Třídění

- Jednoduché třídící algoritmy mají složitost $O(n^2)$.
- Optimální složitost porovnávacích třídících algoritmů je $O(n \log n)$, např. *merge sort*.
- Typicky bývá nejrychlejší *quick sort*, až na degenerované případy.
- V případech omezené vstupní množiny lze použít *radix sort* se složitostí $O(n)$.