Algoritmi sortiranja

Katedra za informatiku, Fakultet tehničkih nauka, Univerzitet u Novom Sadu

2022.

Sortiranje

- sortiranje: izmena redosleda elemenata u kolekciji tako da budu poređani od najmanjeg ka najvećem

Implementacija u Pythonu

```
def bubble_sort(array):
  n = len(array)
  for i in range(n-1):
    for j in range(0, n-i-1):
      #ako je naredni element manji, zameni im mesta
      if array[j] > array[j + 1] :
        array[j], array[j + 1] = array[j + 1], array[j]
```

Red sa prioritetom i sortiranje

- možemo upotrebiti red sa prioritetom za sortiranje niza elemenata
  - dodamo elemente jedan po jedan putem add operacije
  - uklonimo elemente jedan po jedan putem remove_min operacije
- vreme izvršavanja zavisi od načina implementacije

PQ_sort(S, C)

Input: sekvenca S, komparator C
Output: rastuće sortirana S u skladu sa C

$P \leftarrow$ RSP sa komparatorom C
while $\neg S$.is_empty() **do**
  $e \leftarrow S$.remove_first()
  P.add$(e, \emptyset)$
while $\neg P$.is_empty() **do**
  $e \leftarrow P$.remove_min().key()
  S.add_last(e)

Selection sort

- selection sort je varijanta PQ-sorta gde je RSP implementiran pomoću **nesortirane** liste
- vreme izvršavanja selection sorta:
  - dodavanje n elemenata u RSP traje $O(n)$
  - uklanjanje n elemenata u sortiranom redosledu traje

$$1 + 2 + ... + n$$

- selection sort radi u $O(n^2)$ vremenu

Selection sort: primer

	sekvenca S	**red** P
ulaz:	$(7, 4, 8, 2, 5, 3, 9)$	$()$
faza 1		
(a)	$(4, 8, 2, 5, 3, 9)$	(7)
(b)	$(8, 2, 5, 3, 9)$	$(7, 4)$
...	...	...
(g)	$()$	$(7, 4, 8, 2, 5, 3, 9)$
faza 2		
(a)	(2)	$(7, 4, 8, 5, 3, 9)$
(b)	$(2, 3)$	$(7, 4, 8, 5, 9)$
(c)	$(2, 3, 4)$	$(7, 8, 5, 9)$
(d)	$(2, 3, 4, 5)$	$(7, 8, 9)$
(e)	$(2, 3, 4, 5, 7)$	$(8, 9)$
(f)	$(2, 3, 4, 5, 7, 8)$	(9)
(g)	$(2, 3, 4, 5, 7, 8, 9)$	$()$

Implementacija u Pythonu

```
from pqueue import UnsortedPriorityQueue

def pq_sort(A):
  """
  Funkcija sortira listu koristeći dodatni prioritetni red

  Argument:
  - `A`: lista koja se sortira
  """
  size = len(A)
  pq = UnsortedPriorityQueue()

  # svi elementi liste prebacuju se u prioritetni red
  for i in range(size):
    element = A.pop()
    pq.add(element, element)

  # vraćanje elementa iz prioritetnog reda u listu
  for i in range(size):
    (k, v) = pq.remove_min()
    A.append(v)
```

Insertion sort

- insertion sort je varijanta PQ-sorta gde je RSP implementiran pomoću **sortirane** liste
- vreme izvršavanja insertion sorta:
  - dodavanje n elemenata u RSP traje

$$1 + 2 + ... + n$$

  - uklanjanje n elemenata traje $O(n)$
- insertion sort radi u $O(n^2)$ vremenu

Insertion sort: primer

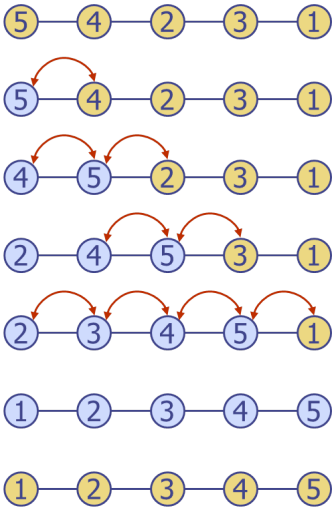
	sekvenca S	**red** P
ulaz:	$(7, 4, 8, 2, 5, 3, 9)$	$()$
faza 1		
(a)	$(4, 8, 2, 5, 3, 9)$	(7)
(b)	$(8, 2, 5, 3, 9)$	$(4, 7)$
(c)	$(2, 5, 3, 9)$	$(4, 7, 8)$
(d)	$(5, 3, 9)$	$(2, 4, 7, 8)$
(e)	$(3, 9)$	$(2, 4, 5, 7, 8)$
(f)	(9)	$(2, 3, 4, 5, 7, 8)$
(g)	$()$	$(2, 3, 4, 5, 7, 8, 9)$
faza 2		
(a)	(2)	$(3, 4, 5, 7, 8, 9)$
(b)	$(2, 3)$	$(4, 5, 7, 8, 9)$
...	...	...
(g)	$(2, 3, 4, 5, 7, 8, 9)$	$()$

Implementacija u Pythonu

```
from pqueue import SortedPriorityQueue

def pq_sort(A):
  """
  Funkcija sortira listu koristeći dodatni prioritetni red

  Argument:
  - `A`: lista koja se sortira
  """
  size = len(A)
  pq = SortedPriorityQueue()

  # svi elementi liste prebacuju se u prioritetni red
  for i in range(size):
    element = A.pop()
    pq.add(element, element)

  # vraćanje elementa iz prioritetnog reda u listu
  for i in range(size):
    (k, v) = pq.remove_min()
    A.append(v)
```

Sortiranje unutar iste strukture podataka (in-place)

- umesto korišćenja 2 strukture možemo implementirati selection i insertion sort u okviru jedne strukture
- deo ulaznog niza će poslužiti kao RSP
- za insertion sort
  - držimo sortiran početak niza
  - elemente menjamo pomoću swap operacije



Implementacija u Pythonu

```
def selection_sort(array):

  for i in range(len(array)):
    # Nađi najmanji element u preostalom
    # delu nesortiranog niza
    min_i = i
    for j in range(i+1, len(array)):
      if array[min_i] > array[j]:
        min_i = j

    # Zameni pronađeni minimalni element sa
    # prvim elementom u trenutnoj iteraciji
    array[i], array[min_i] = array[min_i], array[i]
```

Implementacija u Pythonu

```
def insertion_sort(array):
  n = len(array)
  for i in range(1, n):
    curr = array[i]
    # Pomeraj elemente array[0..i-1], koji su veći
    # od trenutnog na poziciju za jedan veću
    # od njihove trenutne
    j = i-1
    while j >=0 and curr < array[j] :
      array[j+1] = array[j]
      j -= 1
    array[j+1] = curr
```

Merge sort

- merge sort je primer **divide-and-conquer** šablona
  - **divide**: podeli S na dva disjunktna podskupa S_1 i S_2
  - **recur**: reši potproblem za S_1 i S_2
  - **conquer**: kombinuj rešenja za S_1 i S_2 u rešenje za S
- bazni slučaj za rekurziju je skup veličine 0 ili 1
- merge sort je $O(n \log n)$

Merge sort algoritam

- sortira sekvencu S dužine n u tri koraka

1 divide: podeli S na S_1 i S_2, svaki dužine $n/2$

2 recur: rekurzivno sortiraj S_1 i S_2

3 conquer: spoj sortirane S_1 i S_2 u sortiranu sekvencu

mergeSort(S)

Input: sekvenca S sa n elemenata
Output: sortirana sekvenca S
  if len(S) > 1 **then**
    $(S_1, S_2) \leftarrow$ partition($S, n/2$)
    mergeSort(S_1)
    mergeSort(S_2)
    $S \leftarrow$ merge(S_1, S_2)

Spajanje sortiranih sekvenci

merge(A, B)

Input: sekvenca A i B sa $n/2$ elemenata svaka
Output: sortirana sekvenca $A \cup B$

$S \leftarrow$ prazna sekvenca
while $\neg A.isEmpty() \wedge \neg B.isEmpty()$ **do**
  if $A.first().element() < B.first().element()$ **then**
    $S.addLast(A.remove(A.first()))$
  else
    $S.addLast(B.remove(B.first()))$
while $\neg A.isEmpty()$ **do**
  $S.addLast(A.remove(A.first()))$
while $\neg B.isEmpty()$ **do**
  $S.addLast(B.remove(B.first()))$

Merge sort pomoću reda u Pythonu

```
def merge(S1, S2, S):
  while not S1.is_empty() and not S2.is_empty():
    if S1.first() < S2.first():
      S.enqueue(S1.dequeue())
    else:
      S.enqueue(S2.dequeue())
  while not S1.is_empty():     # dodaj preostale elemente S1 u S
    S.enqueue(S1.dequeue())
  while not S2.is_empty():     # dodaj preostale elemente S2 u S
    S.enqueue(S2.dequeue())
```

Merge sort pomoću reda u Pythonu

```
def merge_sort(S):
  n = len(S)
  if n < 2:
    return  # lista je već sortirana
  # podeli
  S1 = Queue()
  S2 = Queue()
  while len(S1) < n // 2: # dodaj prvu polovinu elemenata u S1
    S1.enqueue(S.dequeue())
  while not S.is_empty(): # dodaj ostatak elemenata u S2
    S2.enqueue(S.dequeue())
  # vladaj
  merge_sort(S1)        # sortiraj prvu polovinu
  merge_sort(S2)        # sortiraj drugu polovinu
  merge(S1, S2, S)      # spoji sortirane polovine u S
```

Spajanje sortiranih sekvenci

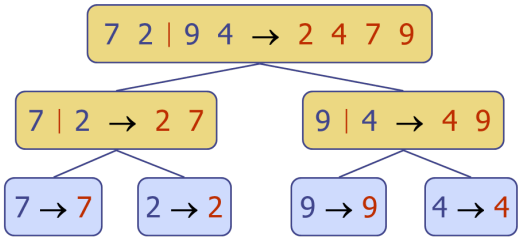
```
def merge(L, R):
  n = len(L)
  m = len(R)
  # indeksi listi L i R, respektivno
  i = 0
  j = 0
  # izlazna lista
  sorted = []
  # dokle god u obe liste ima neispitanih elemenata, proveravaj tekuće
  while i < n and j < m:
    if L[i] < R[j]:
      # element `leve` liste je manji, dodaj u sortiranu i pomeri indeks
      sorted.append(L[i])
      i += 1
    else:
      # element `desne` liste je manji, dodaj u sortiranu i pomeri indeks
      sorted.append(R[j])
      j += 1
  # u jednoj od listi je ostalo elemenata, proveri koja je lista i kopiraj
  # preostale elemente u rezultujuću listu
  if i < n:
    sorted.extend(L[i:])
  else:
    sorted.extend(R[j:])
  return sorted
```

Merge sort u Pythonu

```
def merge_sort(array):
  """
  Merge sort algoritam

  Argument:
  - `array`: lista za sortiranje
  """
  # bazni slučaj (lista od jednog elementa)
  n = len(array)
  if n == 1:
    return array

  # prepolovi listu i sortiraj polovine
  mid = n//2
  L = merge_sort(array[:mid])
  R = merge_sort(array[mid:])

  # spoji liste i vrati rezultat spajanja
  return merge(L, R)
```

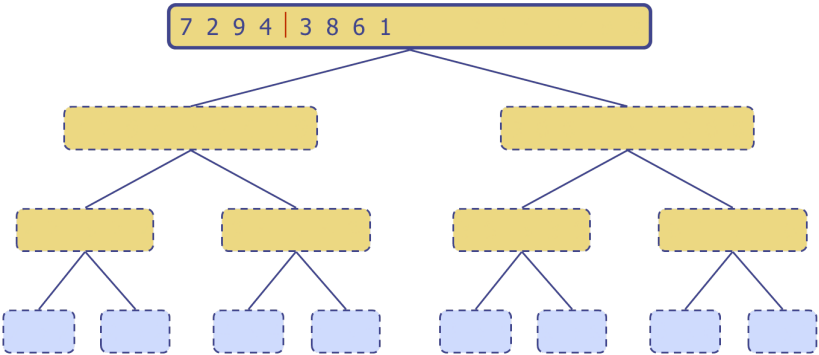
Stablo sortiranja

- izvršavanje merge sorta može se prikazati binarnim stablom
- čvor stabla predstavlja jedan rekurzivni poziv i čuva
  - nesortiranu sekvencu pre podele
  - sortiranu sekvencu nakon završetka
- koren je početni poziv funkcije
- listovi su pozivi sa podsekvence dužine 0 ili 1



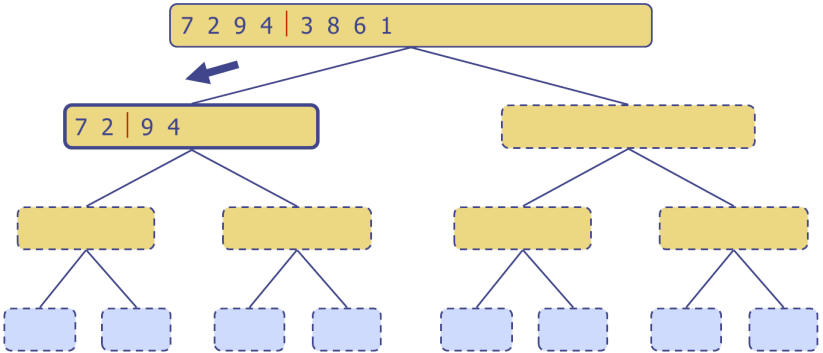
Primer sortiranja

- podela



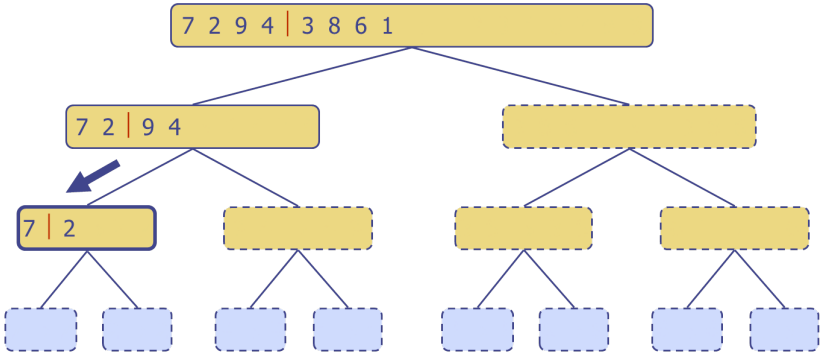
Primer sortiranja

- rekurzija, podela



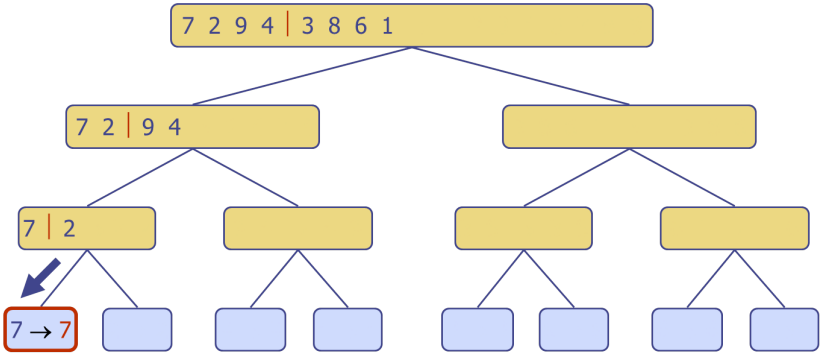
Primer sortiranja

- rekurzija, podela



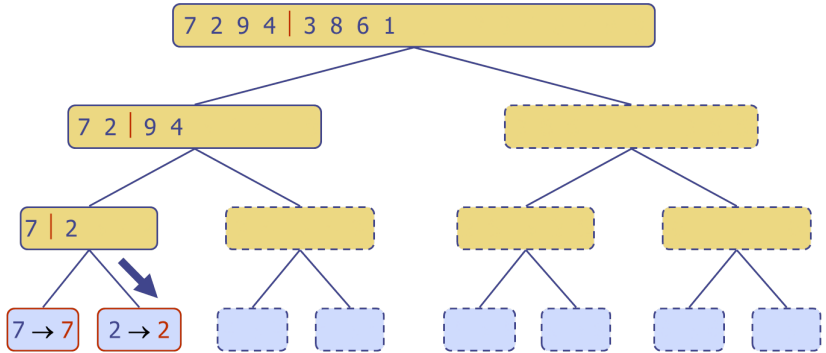
Primer sortiranja

- rekurzija, bazni slučaj



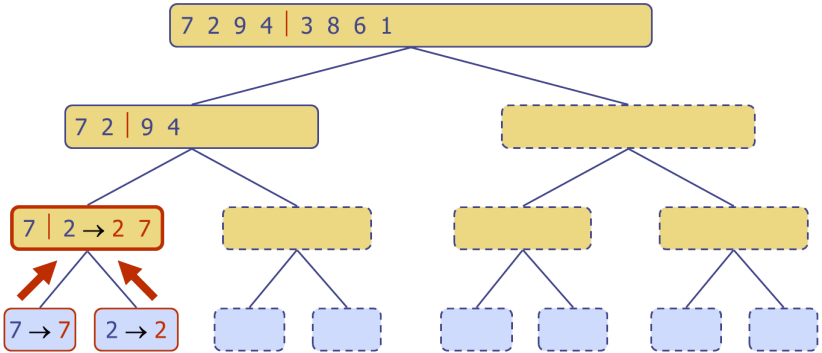
Primer sortiranja

- rekurzija, bazni slučaj



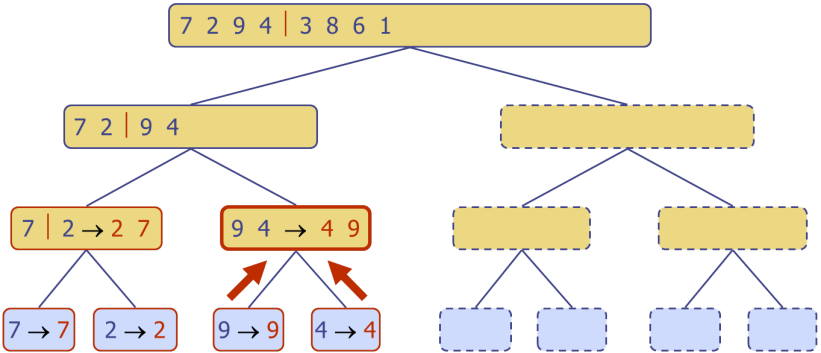
Primer sortiranja

- spajanje



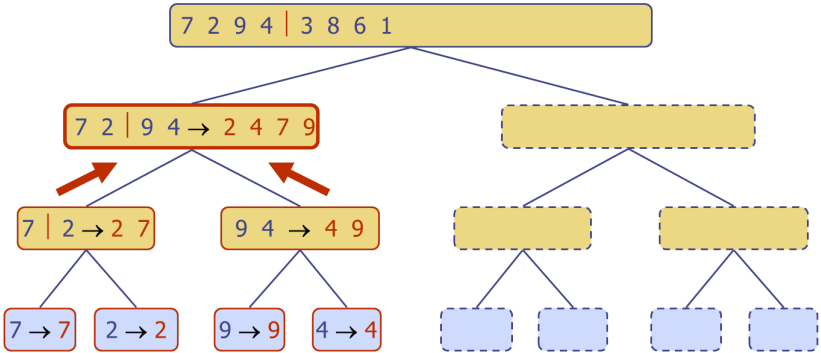
Primer sortiranja

- rekurzija, ..., bazni slučaj, spajanje



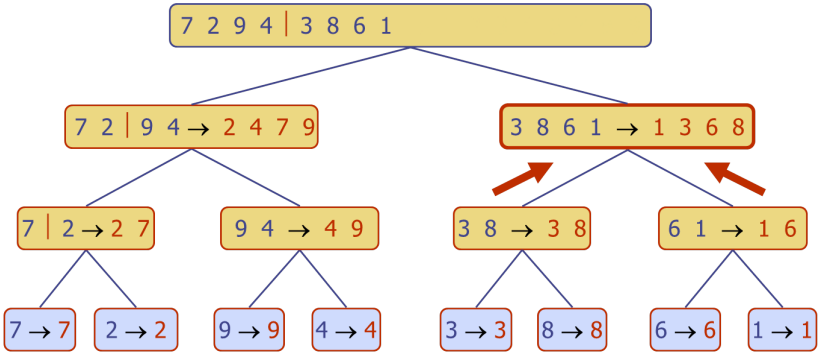
Primer sortiranja

- spajanje



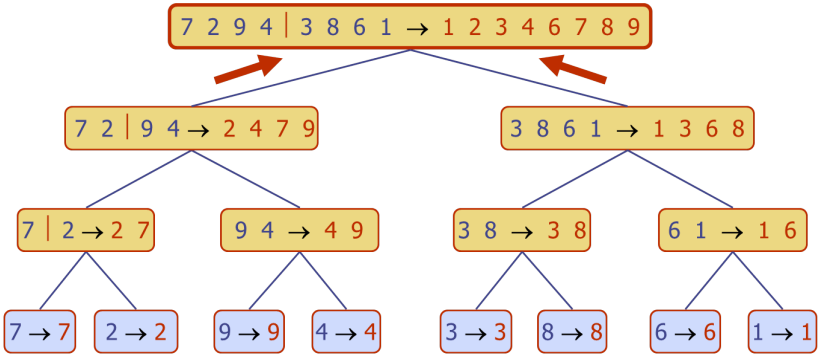
Primer sortiranja

- rekurzija, ..., spajanje, spajanje



Primer sortiranja

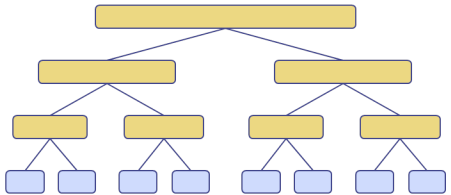
- spajanje



Merge sort: performanse

- visina h stabla za merge sort je $O(\log n)$
  - delimo sekvencu na pola za svaku rekurziju
- ukupan broj operacija na nivou i je $O(n)$
  - delimo i spajamo 2^i sekvenci dužine $n/2^i$
  - pravimo 2^{i+1} rekurzivnih poziva
- ukupno vreme izvršavanja je $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
...	...	...



Algoritmi za sortiranje (za sada)

algoritam	**vreme**	**napomene**
selection	$O(n^2)$	▷ spor ▷ in-place ▷ za male sekvence ($<$ 1K)
insertion	$O(n^2)$	▷ spor ▷ in-place ▷ za male sekvence ($<$ 1K)
merge	$O(n \log n)$	▷ brz ▷ sekvencijalan ▷ za ogromne sekvence ($>$ 1M)

Merge sort: nerekurzivna varijanta 1

```
def merge_sort(S):
  """Sort the elements of Python list S using the merge-sort algorithm."""
  n = len(S)
  logn = math.ceil(math.log(n,2))
  src, dest = S, [None] * n              # make temporary storage for dest
  for i in (2**k for k in range(logn)): # pass i creates all runs of length 2i
    for j in range(0, n, 2*i):           # each pass merges two length i runs
      merge(src, dest, j, i)
    src, dest = dest, src                # reverse roles of lists
  if S is not src:
    S[0:n] = src[0:n]                    # additional copy to get results to S
```

Merge sort: nerekurzivna varijanta $_2$

```
def merge(src, result, start, inc):
  """Merge src[start:start+inc] and src[start+inc:start+2*inc]."""
  end1 = start+inc                    # boundary for run 1
  end2 = min(start+2*inc, len(src))   # boundary for run 2
  x, y, z = start, start+inc, start   # index into run 1, run 2, result
  while x < end1 and y < end2:
    if src[x] < src[y]:
      result[z] = src[x]
      x += 1
    else:
      result[z] = src[y]
      y += 1
    z += 1                            # increment z to reflect new result
  if x < end1:
    result[z:end2] = src[x:end1]      # copy remainder of run 1 to output
  elif y < end2:
    result[z:end2] = src[y:end2]      # copy remainder of run 2 to output
```

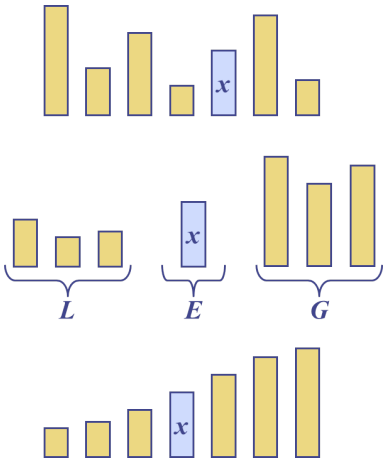
Quick sort

- quick sort je *randomized* podeli-pa-vladaj algoritam

1 divide: izaberi slučajan element x (pivot) i podeli S na
   - L: elementi manji od x
   - E: elementi jednaki x
   - G: elementi veći od x

2 recur: sortiraj L i G

3 conquer: spoj sortirane L, E i G



Particija ulaza

partition(S, p)

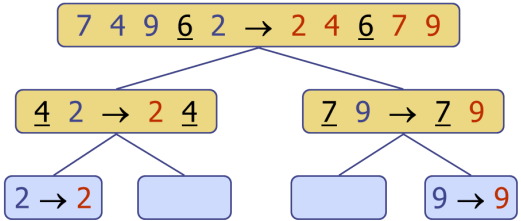
Input: sekvenca S i pozicija pivota p
Output: sekvence L, E, G

```
L, E, G ← prazne sekvence
x ← S.remove(p)
while ¬S.isEmpty() do
    y ← S.remove(S.first())
    if y < x then
        L.addLast(y)
    else if y = x then
        E.addLast(y)
    else
        G.addLast(y)
return L, E, G
```

particija je $O(n)$

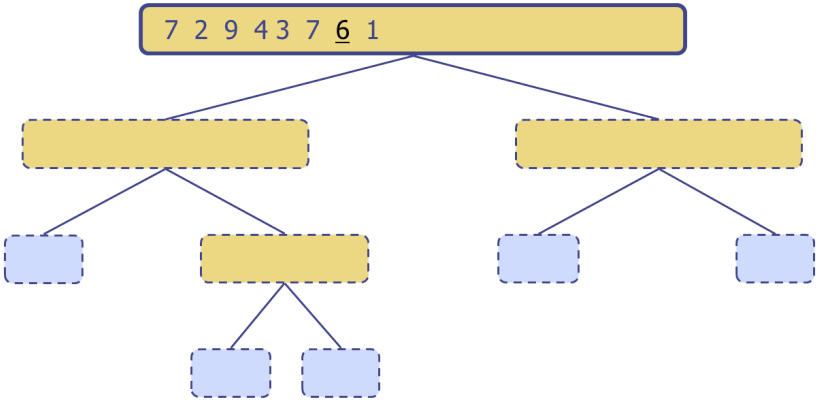
Stablo sortiranja

- izvršavanje quick sorta može se prikazati binarnim stablom
- čvor stabla predstavlja jedan rekurzivni poziv i čuva
  - nesortiranu sekvencu pre podele oko pivota
  - sortiranu sekvencu nakon završetka
- koren je početni poziv funkcije
- listovi su pozivi sa podsekvence dužine 0 ili 1



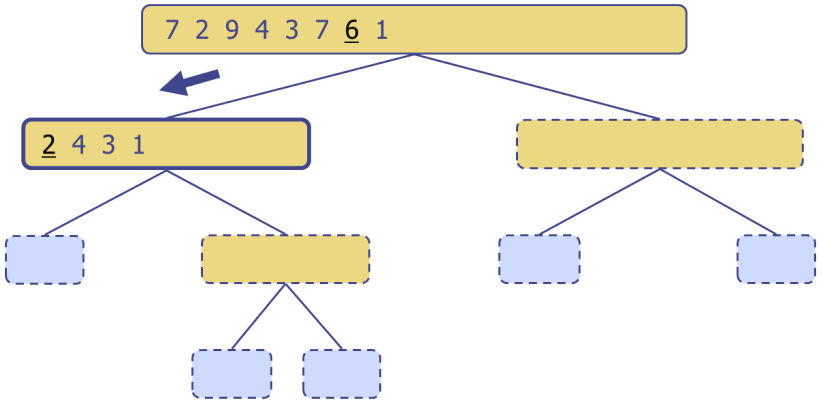
Primer sortiranja

- izbor pivota



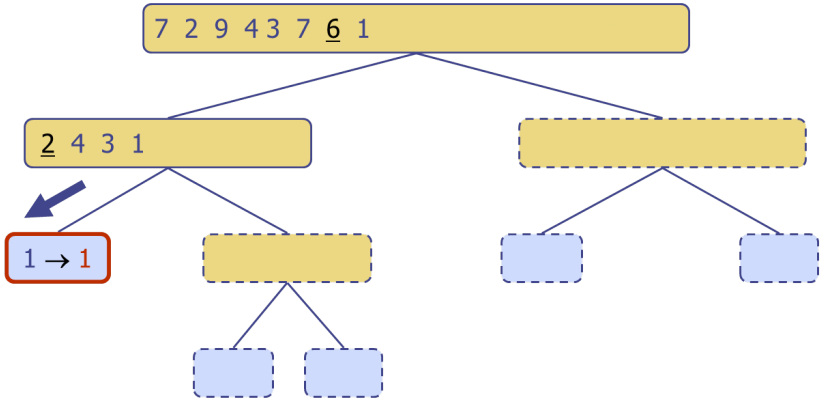
Primer sortiranja

- particija, rekurzija, izbor pivota



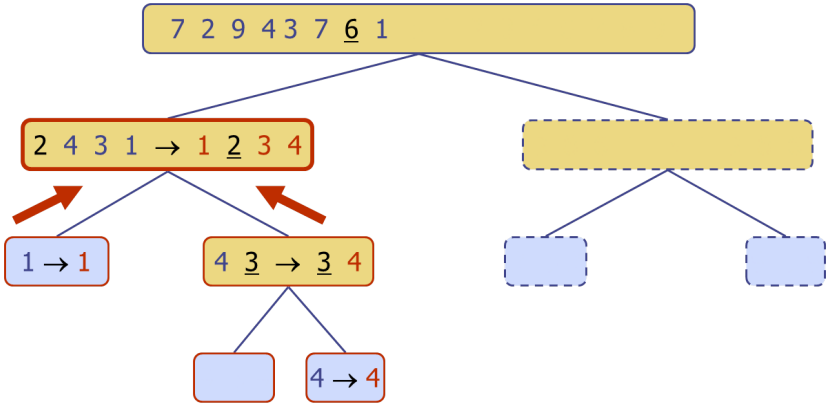
Primer sortiranja

- particija, rekurzija, bazni slučaj



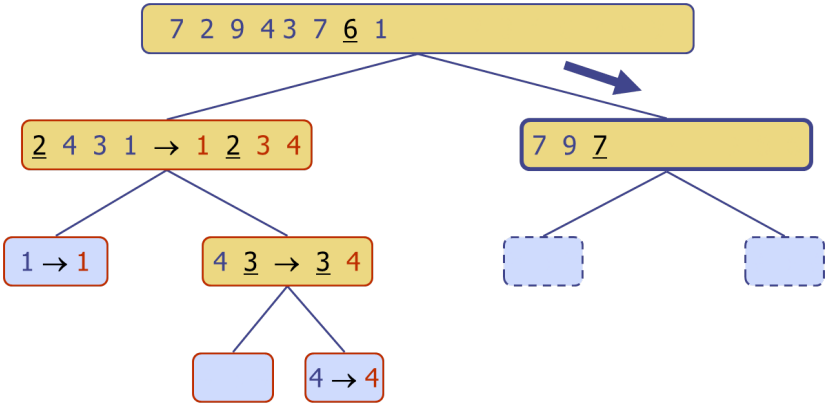
Primer sortiranja

- rekurzija, ..., bazni slučaj, spajanje



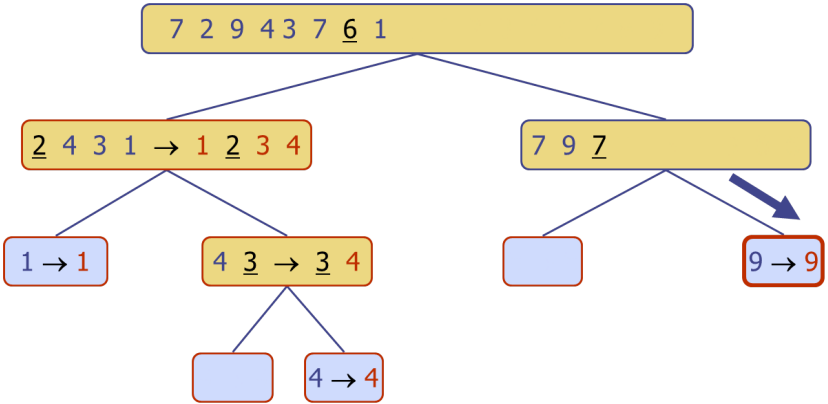
Primer sortiranja

- rekurzija, izbor pivota



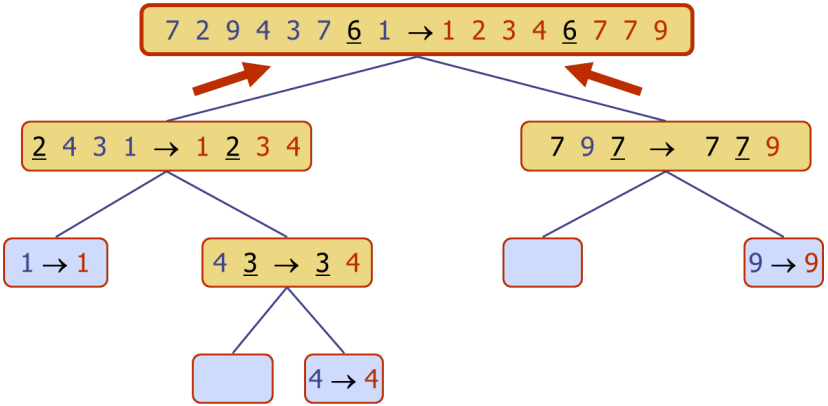
Primer sortiranja

- particija, ..., rekurzija, bazni slučaj



Primer sortiranja

- spajanje, spajanje

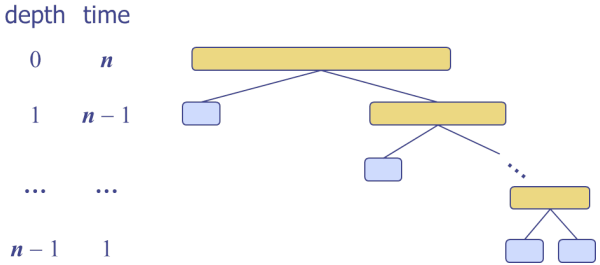


Performanse u najgorem slučaju

- najgori slučaj: kada je pivot najveći ili najmanji element
- jedan od L ili G ima dužinu 0 a drugi dužinu $n-1$
- vreme izvršavanja je tada proporcionalno sumi

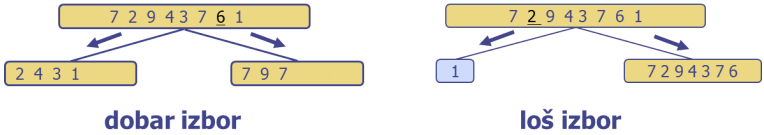
$$n + (n-1) + ... + 2 + 1$$

- $\Rightarrow$ najgori slučaj je $O(n^2)$



Očekivane performanse

- posmatrajmo rekurzivni poziv za sekvencu dužine s
  - dobar izbor: dužine L i G su obe manje od $s \cdot 3/4$
  - loš izbor: L ili G ima dužinu veću od $s \cdot 3/4$

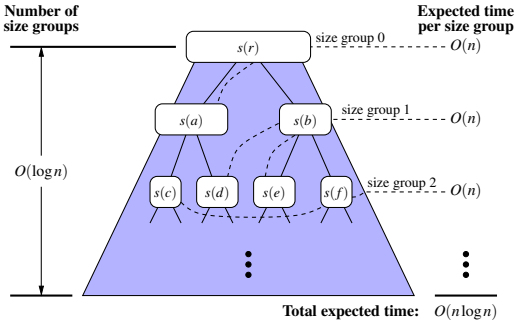


- slučajan izbor pivota je dobar sa verovatnoćom $1/2$
  - $1/2$ mogućih pivota su dobar izbor

Očekivane performanse

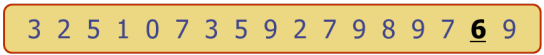
- iz verovatnoće: očekivani broj bacanja novčića da bismo dobili k glava je $2k$
- za čvor dubine i očekujemo
  - $i/2$ predaka su dobri izbori
  - veličina ulazne sekvence za tekući poziv je najviše $n \cdot (3/4)^{i/2}$

- za čvor dubine $2\log_{4/3} n$ očekivana veličina ulaza je 1
- očekivana visina stabla je $O(\log n)$
- broj operacija za čvorove iste dubine je $O(n)$
- $\Rightarrow$ ukupno očekivano vreme quick sorta je $O(n \log n)$

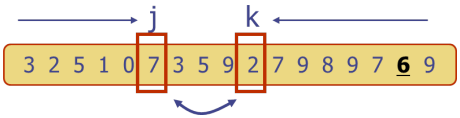


In-place particija

- koristimo indekse j i k da podelimo S na dva dela: L i $E \cup G$



- ponavljaj dok se j i k ne mimoiđu ($k < j$)
  - pomeraj j u desno dok ne naiđemo na element $\geq x$
  - pomeraj k u levo dok ne naiđemo na element $< x$
  - zameni elemente na pozicijama j i k



Algoritmi za sortiranje (za sada)

algoritam	**vreme**	**napomene**
selection	$O(n^2)$	▷ in-place ▷ spor (dobar za male ulaze)
insertion	$O(n^2)$	▷ in-place ▷ spor (dobar za male ulaze)
merge	$O(n \log n)$	▷ sekvencijalan ▷ brz (dobar za ogromne ulaze)
quick	$O(n \log n)$ očekivano	▷ in-place, randomized ▷ najbrži (dobar za velike ulaze)

Bucket sort

- S je sekvenca n parova (ključ, element) sa ključevima u opsegu $[0, N-1]$
- bucket sort koristi ključeve kao indekse u pomoćnom nizu sekvenci (kanti) B
  - **faza 1**: isprazni S premeštanjem svakog (k, o) u svoju kantu $B[k]$
  - **faza 2**: za $i = 0, \ldots, N-1$ premesti elemente kante $B[i]$ na kraj S
- analiza:
  - faza 1 je $O(n)$
  - faza 2 je $O(n + N)$

- $\Rightarrow$ bucket sort je $O(n + N)$

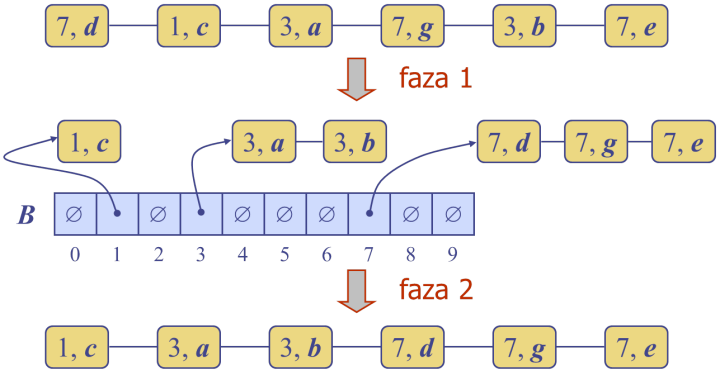
Bucket sort

bucketSort(S)

Input: sekvenca S sa int ključevima u opsegu $[0, N-1]$
Output: sortirana S

```
B ← niz od N praznih sekvenci
for all e in S do
    k ← e.key()
    S.remove(e)
    B[k].addLast(e)
for i ← 0 to N − 1 do
    for all e in B[i] do
        B[i].remove(e)
        S.addLast(e)
```

Primer: opseg ključeva [0,9]



Osobine bucket sorta

- **tip ključa**: ključevi su isključivo integeri
- **stabilno sortiranje**: čuva se poredak među elementima sa istom vrednošću ključa
- proširenja
  - ključevi u opsegu $[a, b]$: element sa ključem k se smešta u $B[k - a]$
  - string ključevi iz konačnog skupa D mogućih vrednosti – sortiramo stringove, pa njihove indekse koristimo kao ključeve

Leksikografski poredak

- d-torka je sekvenca od d ključeva $(k_1, \ldots, k_d)$
- ključ k_i je i-ta dimenzija torke
- primer:
  - Dekartove koordinate 3D tačke su 3-torke
- leksikografski poredak dve d-torke se definiše rekurzivno:

$$(x_1, \ldots, x_d) < (y_1, \ldots, y_d)$$

$$\Leftrightarrow$$

$$x_1 < y_1 \vee x_1 = y_1 \wedge (x_2, \ldots, x_d) < (y_2, \ldots, y_d)$$

- torke se porede po prvoj dimenziji, pa onda po drugoj, itd.

Leksikografsko sortiranje

- neka je C_i komparator koji poredi dve torke po njhovoj i-toj dimenziji
- neka je stableSort(S, C) algoritam koji koristi komparator C
- lexicographic sort sortira sekvencu d-torki u leksikografskom redosledu izvršavajući d puta algoritam stableSort, jednom za svaku dimenziju
- lexicographic sort je $O(dT(n))$ gde je $T(n)$ vreme stableSort-a

lexicographicSort(S)

Input: sekvenca S sa d-torkama
Output: sortirana S
  for $i \leftarrow d$ **to** 1 **do**
    stableSort(S, C_i)

Leksikografsko sortiranje: primer

(7, 4, 6) (5, 1, 5) (2, 4, 6) (2, 1, 4) (3, 2, 4)
(2, 1, 4) (3, 2, 4) (5, 1, 5) (7, 4, 6) (2, 4, 6)
(2, 1, 4) (5, 1, 5) (3, 2, 4) (7, 4, 6) (2, 4, 6)
(2, 1, 4) (2, 4, 6) (3, 2, 4) (5, 1, 5) (7, 4, 6)

Radix sort

- radix sort je specijalizacija leksikografskog sortiranja koje koristi bucket sort kao stabilni sort algoritam za svaku dimenziju
- radix sort je primenljiv na torke gde su ključevi u svakoj dimenziji integeri iz $[0, N-1]$
- radix sort je $O(d \cdot (n+N))$

radixSort(S, N)

Input: sekvenca S sa d-torkama
Output: sortirana S
  for $i \leftarrow d$ **to** 1 **do**
    bucketSort(S, N)

Radix sort za binarne brojeve

- posmatramo sekvencu od n b-bitnih integera

$$x = x_{b-1} \dots x_1 x_0$$

- ove elemente tretiramo kao b-torku sa integerima u opsegu $[0, 1]$
- primenimo radix sort za $N = 2$
- ova varijanta radix sorta je $O(bn)$
- $\Rightarrow$ možemo sortirati niz 32-bitnih integera u linearnom vremenu!

Primer

- sortiramo sekvencu 4-bitnih integera