Sortiranje i selekcija

© Goodrich, Tamassia, Goldwasser

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

2021.

Sortiranje

- sortiranje: izmena redosleda elemenata u kolekciji tako da budu poređani od najmanjeg ka najvećem
- videli smo da red sa prioritetom može da posluži za sortiranje
    - **selection sort** je $O(n^2)$
    - **insertion sort** je $O(n^2)$
    - **heap sort** je $O(n \log n)$

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

- merge sort je $O(n \log n)$, kao i heap sort
- za razliku od heap sorta
  - ne koristi pomoćnu strukturu podataka (RSP)
  - pristupa podacima sekvencijalno – pogodno za sortiranje podataka na disku

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)

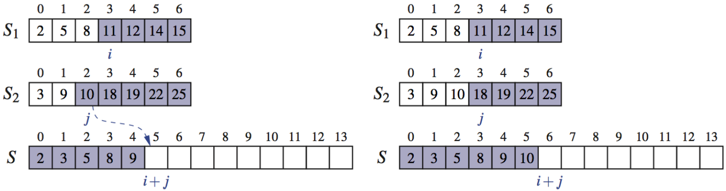
Input: sekvenca A i B sa $n/2$ elemenata svaka

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**
  $S.addLast(B.remove(B.first()))$

Spajanje sortiranih sekvenci

```
def merge(s1, s2, s):
  i = j = 0
  while i + j < len(s):
    if j == len(s2) or (i < len(s1) and s1[i] < s2[j]):
      s[i+j] = s1[i]
      i += 1
    else:
      s[i+j] = s2[i]
      j += 1
```

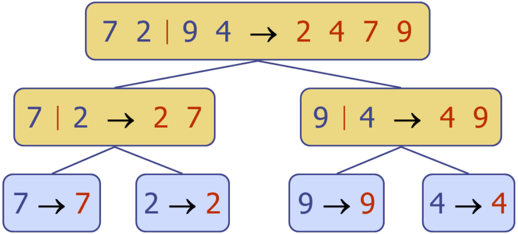


Merge sort u Pythonu

```
def merge_sort(s):
  n = len(s)
  if n < 2:
    return  # već sortirano
  # podeli
  mid = n//2
  s1 = s[0:mid]
  s2 = s[mid:n]
  # rekurzija
  merge_sort(s1)
  merge_sort(s2)
  # vladaj
  merge(s1, s2, s)
```

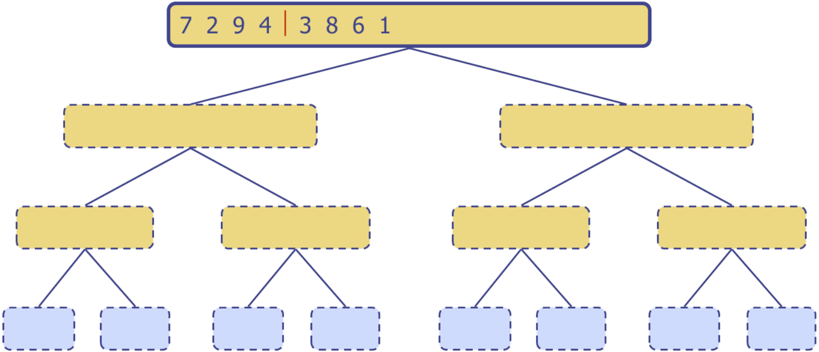
Stablo sortiranja

- 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



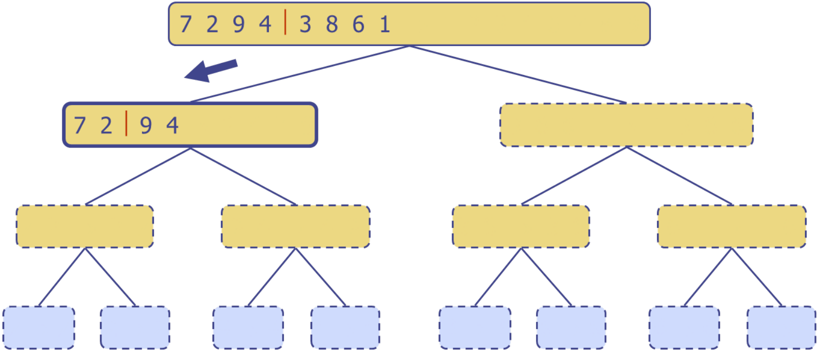
Primer sortiranja

- podela



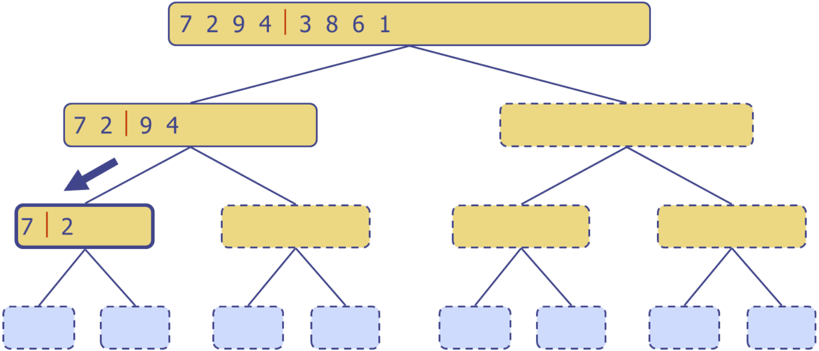
Primer sortiranja

- rekurzija, podela



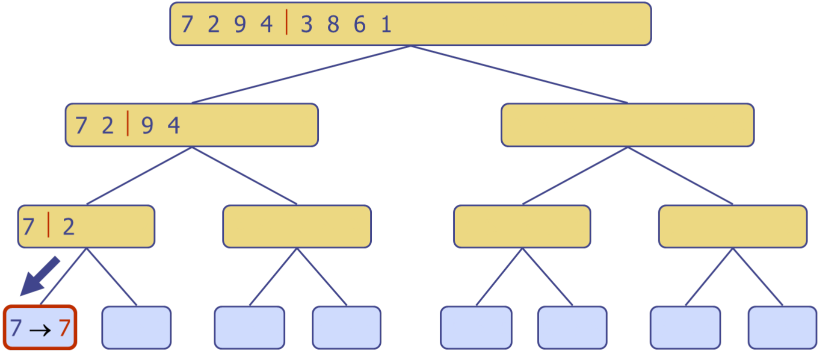
Primer sortiranja

- rekurzija, podela



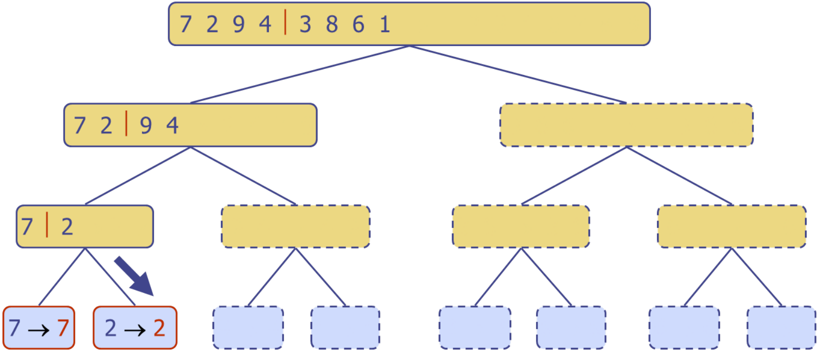
Primer sortiranja

- rekurzija, bazni slučaj



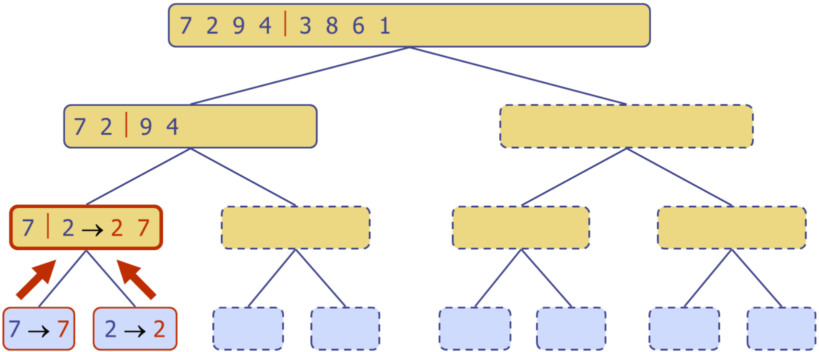
Primer sortiranja

- rekurzija, bazni slučaj



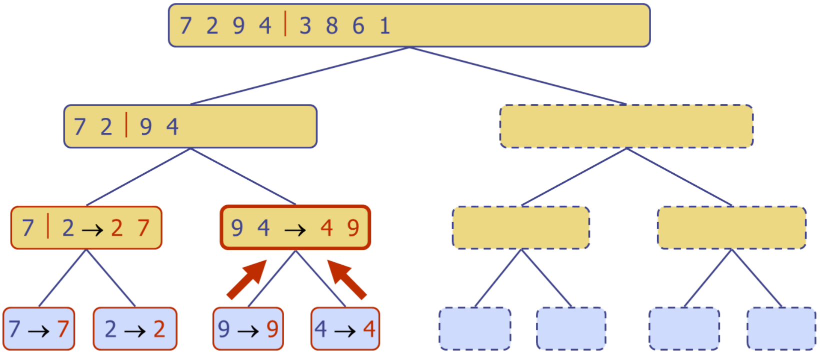
Primer sortiranja

- spajanje



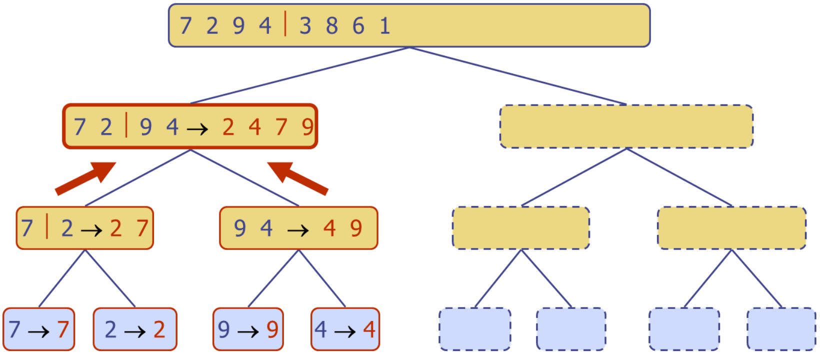
Primer sortiranja

- rekurzija, ..., bazni slučaj, spajanje



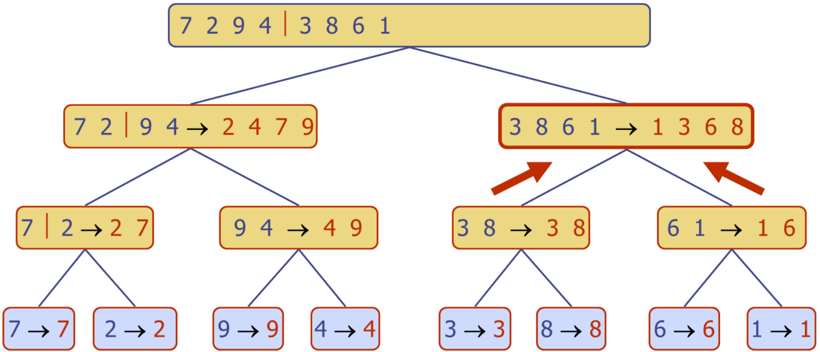
Primer sortiranja

- spajanje



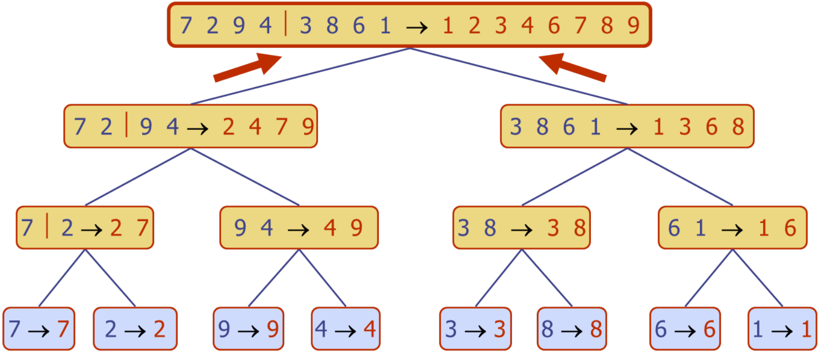
Primer sortiranja

- rekurzija, ..., spajanje, spajanje



Primer sortiranja

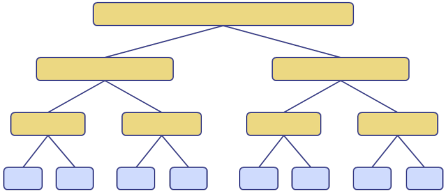
- spajanje



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)
heap	$O(n \log n)$	▷ brz ▷ in-place ▷ za velike sekvence (1K-1M)
merge	$O(n \log n)$	▷ brz ▷ sekvencijalan ▷ za ogromne sekvence ($>$ 1M)

Merge sort: nerekurzivna varijanta 1

```python
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 $_2$

```python
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
```

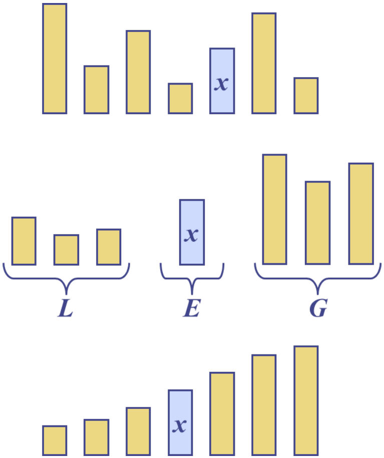
Quick sort

- 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)

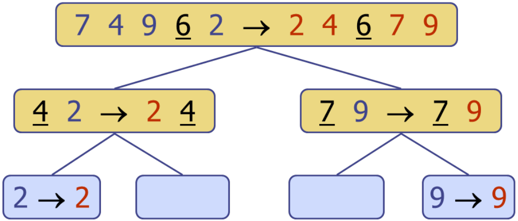
Input: sekvenca S i pozicija pivota p
Output: sekvence L, E, G

```
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 $O(n)$

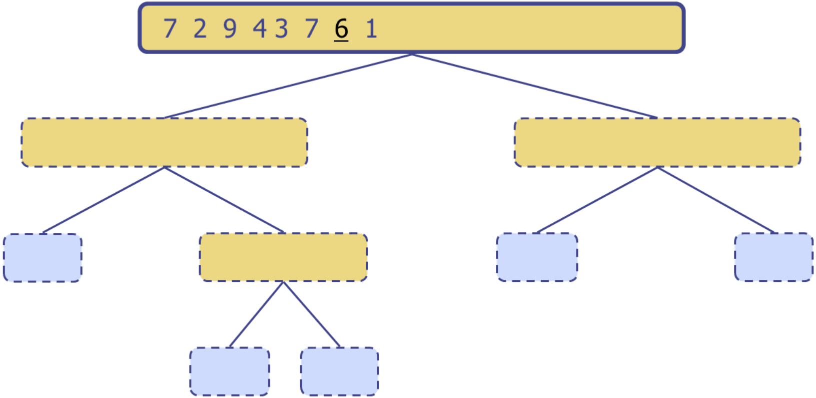
Stablo sortiranja

- 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



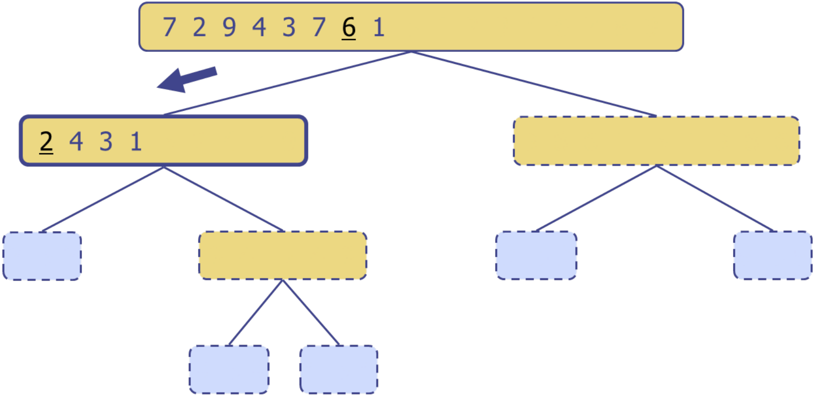
Primer sortiranja

- izbor pivota



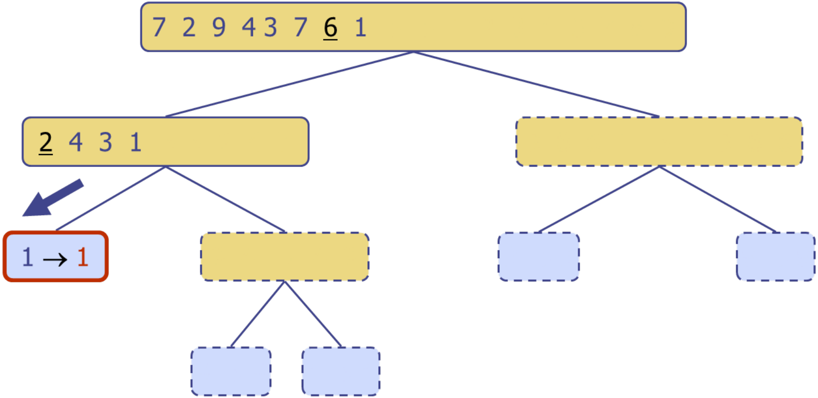
Primer sortiranja

- particija, rekurzija, izbor pivota



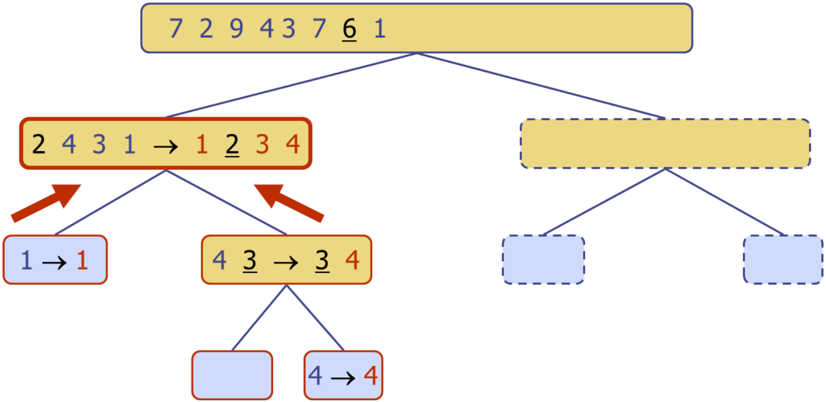
Primer sortiranja

- particija, rekurzija, bazni slučaj



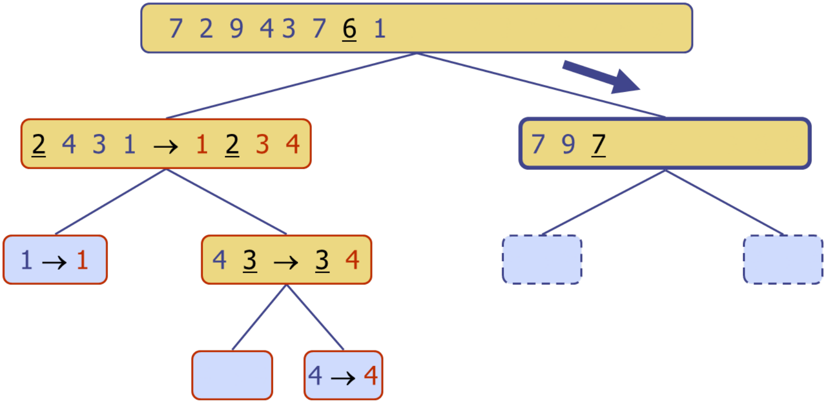
Primer sortiranja

- rekurzija, ..., bazni slučaj, spajanje



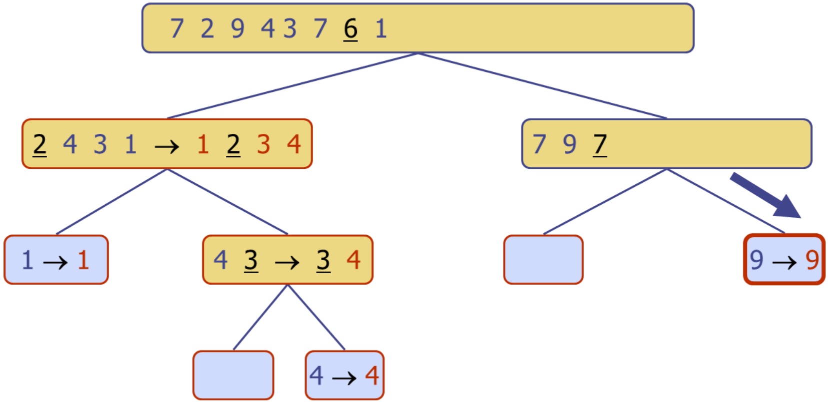
Primer sortiranja

- rekurzija, izbor pivota



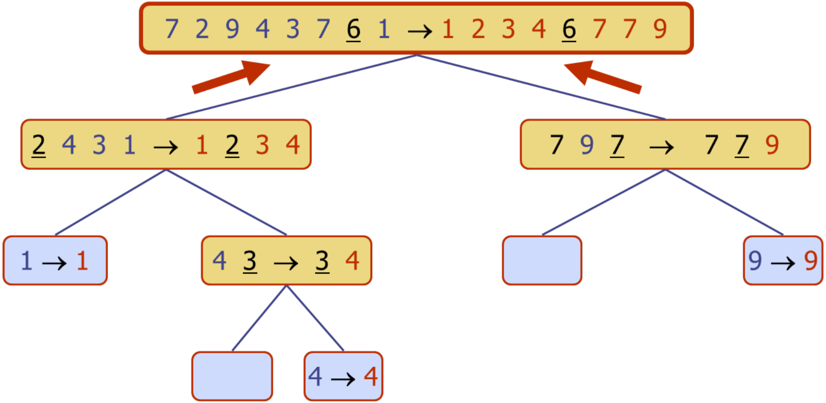
Primer sortiranja

- particija, ..., rekurzija, bazni slučaj



Primer sortiranja

- spajanje, spajanje

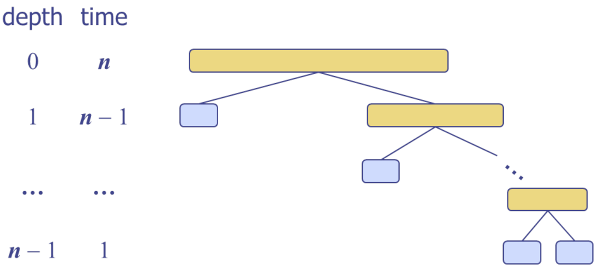


Performanse u najgorem slučaju

- 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

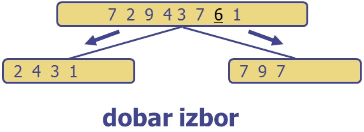
$$n + (n-1) + \ldots + 2 + 1$$

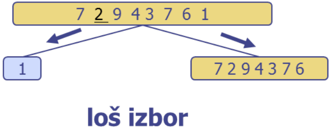
- $\Rightarrow$ najgori slučaj je $O(n^2)$



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 $s \cdot 3/4$



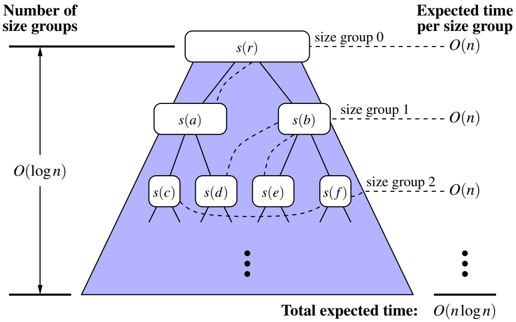


- slučajan izbor pivota je dobar sa verovatnoćom $1/2$
  - $1/2$ mogućih pivota su dobar izbor

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 $O(n \log n)$

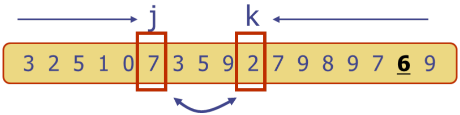


In-place particija

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

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9 (pivot = 6)

- 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 i i k



In-place quick sort

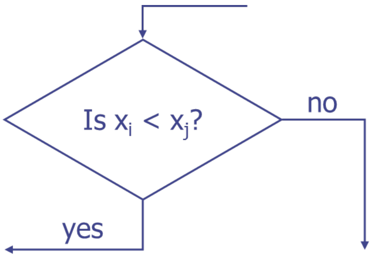
```python
def inplace_quick_sort(S, a, b):
  """Sort the list from S[a] to S[b] inclusive using quick-sort."""
  if a >= b: return                               # range is trivially sorted
  pivot = S[b]                                    # last element of range is pivot
  left = a                                        # will scan rightward
  right = b-1                                     # will scan leftward
  while left <= right:
    # scan until reaching value equal or larger than pivot (or right marker)
    while left <= right and S[left] < pivot:
      left += 1
    # scan until reaching value equal or smaller than pivot (or left marker)
    while left <= right and pivot < S[right]:
      right -= 1
    if left <= right:                             # scans did not strictly cross
      S[left], S[right] = S[right], S[left]       # swap values
      left, right = left + 1, right - 1           # shrink range

  # put pivot into its final place (currently marked by left index)
  S[left], S[b] = S[b], S[left]
  # make recursive calls
  inplace_quick_sort(S, a, left - 1)
  inplace_quick_sort(S, left + 1, b)
```

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)
heap	$O(n \log n)$	▷ in-place ▷ brz (dobar za velike 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)

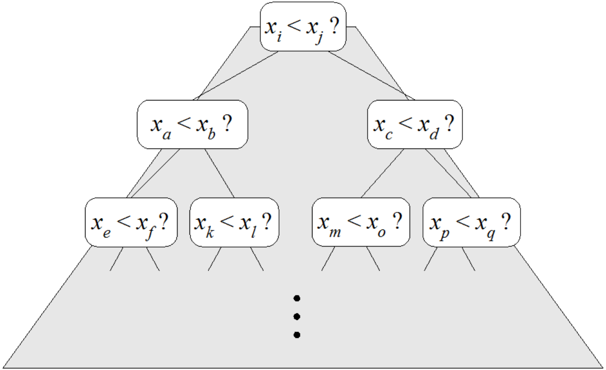
Sortiranje zasnovano na poređenju

- mnogi algoritmi se zasnivaju na poređenju elemenata
  - porede se parovi elemenata
  - bubble, selection, insertion, heap, merge, quick...
- postoji li donja granica za vreme potrebno ovakvim algoritmima koji sortiraju $x_1, x_2, \dots, x_n$?



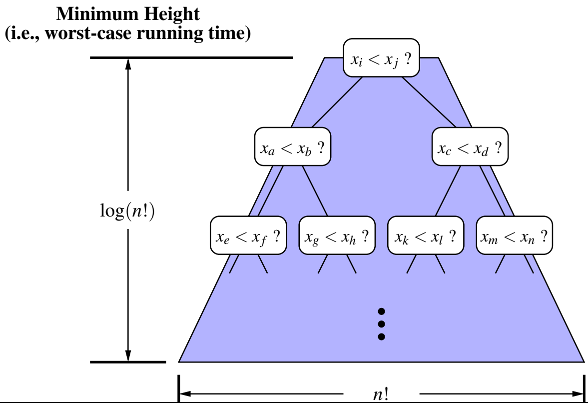
Brojanje operacija poređenja

- svako pokretanje algoritma odgovara jednoj putanji koren→list u stablu odlučivanja



Stablo odlučivanja

- visina stabla odlučivanja je donja granica za vreme sortiranja
- svaka permutacija ulaza vodi do posebnog lista
- stablo ima $n!$ listova $\Rightarrow$ visina stabla je barem $O(\log(n!))$



Granica brzine

- svaki algoritam koji se zasniva na poređenju je barem $O(\log(n!))$
- prema tome, vreme izvšavanja svakog algoritma je najmanje

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2}$$

- tj. svaki ovakav algoritam je $\Omega(n \log n)$

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 $O(n+N)$

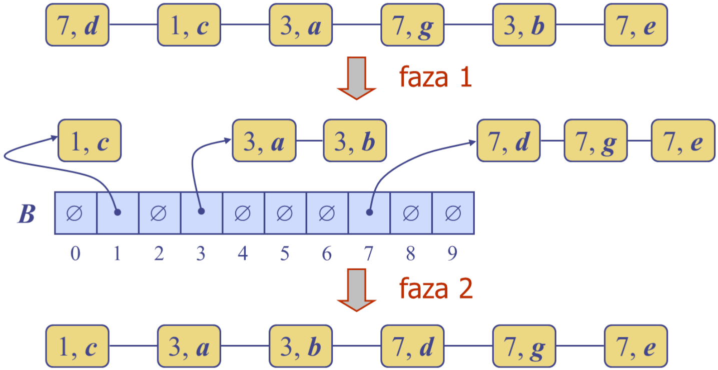
Bucket sort

bucketSort(S)

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

$B \leftarrow$ niz od N praznih sekvenci
for all e in S **do**
  $k \leftarrow e$.key()
  S.remove(e)
  $B[k]$.addLast(e)
for $i \leftarrow 0$ **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 $(k_1, \dots, k_d)$
- ključ k_i je i-ta dimenzija torke
- primer:
  - Dekartove koordinate 3D tačke su 3-torke
- leksikografski poredak dve d-torke se definiše rekurzivno:

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

$$\Leftrightarrow$$

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

- 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 ← 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

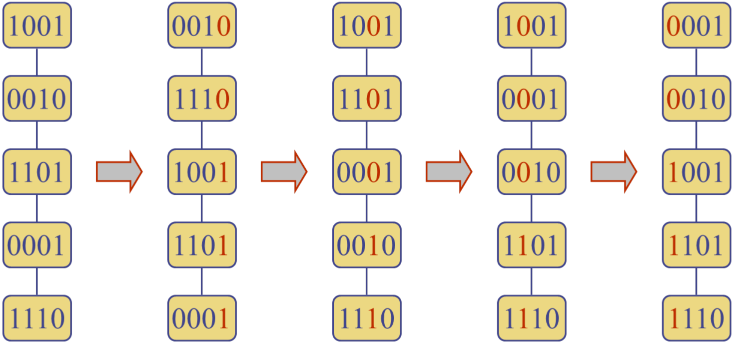
- posmatramo sekvencu od n b-bitnih integera

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

- 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



Problem selekcije

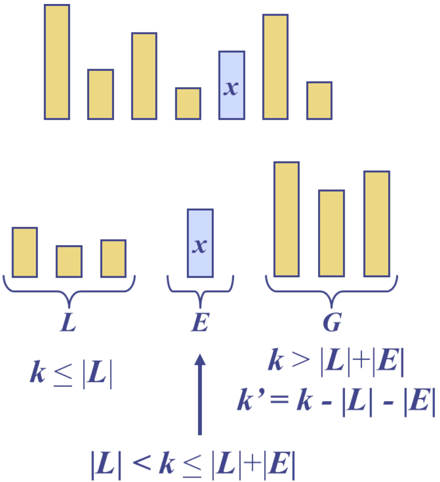
- za dati integer k i n elemenata $x_1, \ldots, x_n$ treba naći k-ti najmanji element
- možemo sortirati niz za $O(n \log n)$ vreme i pristupiti k-tom elementu

k=3 7 4 9 <u>6</u> 2 → 2 4 <u>6</u> 7 9

- da li može brže?

Quick select

- quick select je randomized algoritam zasnovan na **prune-and-search** principu
  - prune: izaberi slučajan element x (pivot) i podeli S na L, E i G
  - search: zavisno od k, ili je element u E ili moramo da tražimo rekurzivno u L ili G



Particija ulaza

partition(S, p)

Input: sekvenca S i pozicija pivota p

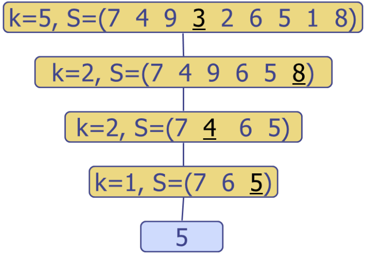
Output: sekvence L, E, G

```
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
```

isto kao kod quick sorta $\Rightarrow$ particija je $O(n)$

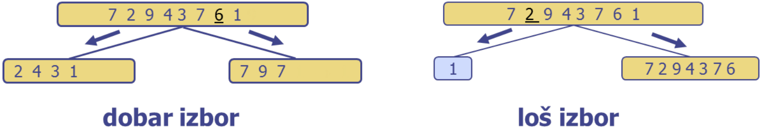
Vizuelizacija quick selecta

- quick select možemo prikazati kao putanju rekurzije
- svaki čvor je jedan rekurzivni poziv quickSelecta, čuva k i preostalu sekvencu



Očekivane performanse

- posmatrajmo rekurzivni poziv quick selecta 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

(1) iz verovatnoće: očekivani broj bacanja novčića da bismo dobili k glava je $2k$

(2) iz verovatnoće: očekivanje je linearna funkcija

- $E(X+Y) = E(X) + E(Y)$
- $E(cX) = cE(x)$

- neka je $T(n)$ vreme quick selecta
- prema (2): $T(n) = T(3n/4) + n \cdot$ (očekivani broj izbora do dobrog)
- prema (1): $T(n) = T(3n/4) + bn$
- tj. $T(n)$ je geometrijska progresija

$$T(n) \leq 2bn + 2b(3/4)n + 2b(3/4)^2 n + 2b(3/4)^3 n \ldots$$

- dakle $T(n)$ je $O(n)$
- možemo rešiti problem selekcije u linearnom vremenu

Deterministička selekcija

- možemo uraditi selekciju i u najgorem slučaju za $O(n)$
- osnovna ideja: rekurzivno koristimo quick select da bismo našli dobrog pivota za quick select
  - podeli S na 5 delova dužine $n/5$
  - nađi median u svakom podskupu
  - nađi median „malih" mediana rekurzivno