# SORTIRANJE

# PROBLEM SORTIRANJA



- **Šta je to sortiranje?**
- Jedan od najšire proučavanih problema u računarstvu
- Sortiranje je osnova za mnoge algoritme i troši veliki deo računarskog vremena u mnogim tipičnim aplikacijama
- Postoje mnoge verzije problema sortiranja, kao i desetine algoritama za sortiranje
- Retko kad je to samo niz brojeva već deo neke veće strukture
- Algoritmi za sortiranje su primer kako apstrakcijom svodimo problem na jednostavniju (matematičku) formu koju rešavamo algoritmom
- Ulaz je nesortiran niz (brojeva) od n brojeva: $\langle a_1, a_2, \dots, a_n \rangle$
- Izlaz je permutacija ulaznog niza: $\langle a'_1, a'_2, \dots, a'_n \rangle$, $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

# ALGORITMI ZA SORTIRANJE



- Postoji ogroman broj algoritama za sortiranje:
  - Quicksort
  - Merge sort
  - In-place merge sort
  - Heapsort
  - Insertion sort
  - Introsort
  - Selection sort
  - Timsort
  - Cubesort
  - Shell sort
  - Bubble sort
  - Binary tree sort
  - Cycle sort
  - Library sort
  - Patience sorting
  - Smoothsort
  - Strand sort
  - Tournament sort
  - Cocktail sort
  - Comb sort
  - Gnome sort
  - UnShuffle Sort
  - Franceschini's method
  - Block sort
  - Odd–even sort
  - Bucket sort

# HEAPSORT

# HEAPSORT ALGORITAM …

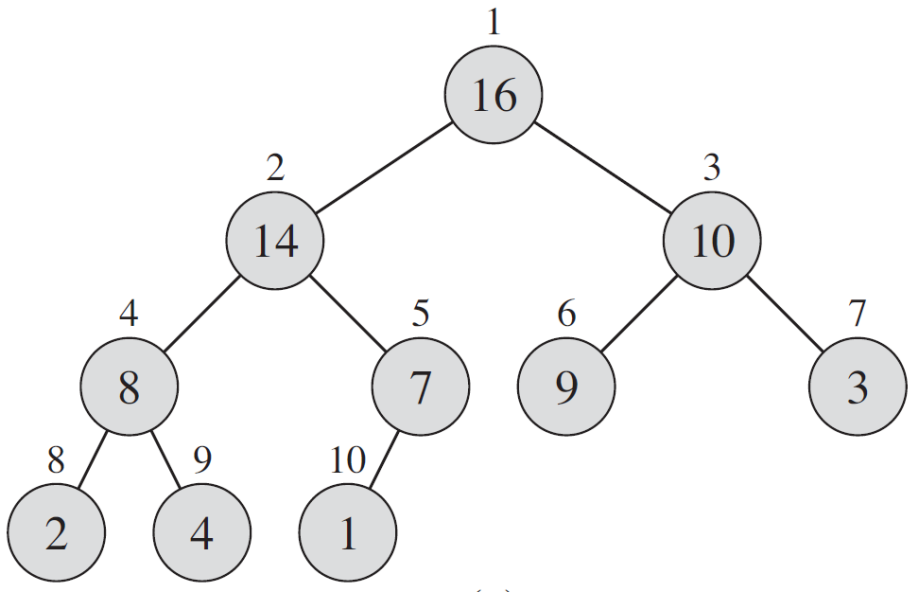


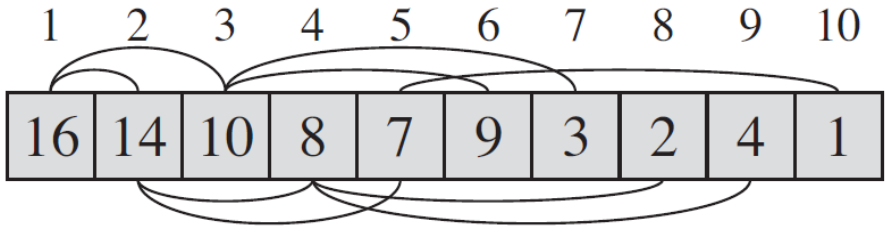
- Kompleksnost je $\boldsymbol{O}(\boldsymbol{n}\log\boldsymbol{n})$
- Sortiranje u mestu, u samom nizu, dok se samo konstantan broj elemenata nalaza van niza tokom primene algoritma
- Uvodimo još jednu tehniku dizajna algoritma, strukturu podataka
  - Pojednostavljujemo i prilagođavamo se problemu tako što biramo strukturu podataka koja nam olakšava rad sa datim problemom
  - Koristimo heap strukturu za implementaciju Heapsort algoritma
  - Vrlo efikasan način da se organizuje red sa prioritetima
  - Sam pojam heap strukture je nastao iz heapsort algoritma, ali se danas odnosi na „garbage-collected storage“
  - Poznata još i kao „gomila“
  - Naravno u primerima, i ItA knjizi, misli se na strukturu, ne na deo programskog jezika, ili virtualne mašine

# … HEAPSORT ALGORITAM …



- Heap struktura (binarna) je niz koji se može posmatrati kao skoro kompletno binarno stablo
- Svaki čvor stabla odgovara elementu niza
- Stablo je potpuno popunjeno, osim možda u listovima, ali se i oni popunjavaju s leve strane ka desnoj do određene tačke
- Niz ***A*** koji predstavlja heap je objekat koji ima 2 atributa:
  - ***A.length***, koji predstavlja broj elemenata u nizu
  - ***A.heap-size***, koji predstavlja koliko elemenata iz heap-a je smešteno u niz



# … HEAPSORT ALGORITAM …



- U suštini, ne moraju svi elementi niza biti elementi heap-a, što znači ako je $A[1\,..\,A.length]$, onda je $0 \leq A.heap - size \leq A.length$
- Koren heap-a se nalazi na lokaciji $A[1]$
- Ako znamo da je indeks nekog čvora ***i***, onda u nizu možemo pronaći pozicije njegovog nadređenog čvora i njemu podređenih čvorova primenom jednostavnih procedura **PARENT*(i)***, **LEFT*(i)***, **RIGHT*(i)***

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

- To je razlog zašto stablo mora biti popunjeno
- Kako da brzo izračunamo **LEFT*(i)*** i **RIGHT*(i)*** kao jednu operaciju?
- Primenom bitwise operatora (pomeranje za jedan bit u levo za **LEFT*(i)*** i pomeranje za jedan bit u levo plus dodavanje jednog bita na poslednje mesto za **RIGHT*(i)***)

# … HEAPSORT ALGORITAM …



- Kako da brzo izračunamo **PARENT*(i)***?
- Primenom bitwise operacije za jedno mesto u desno
- Dobra implementacija koristi makro ili inline proceduru.
- Šta je to?

# … HEAPSORT ALGORITAM …



- Postoje 2 vrste binarnih heap struktura:
  - Maks-heap struktura
  - Min-heap struktura
- Podela na osnovu toga koji uslov moraju zadovoljiti elementi heap-a
- Kod Maks-heap strukture to je uslov da ni jedan čvor nema vrednost veću od njegovog nadređenog čvora, $\boldsymbol{A[Parent(i)] \geq A[i]}$
- Najveći element u Maks-heap strukturi je koren
- Kod Min-heap strukture je obrnuto, uslov je da ni jedan čvor nema manju vrednost od njegovog nadređenog čvora, $\boldsymbol{A[Parent(i)] \leq A[i]}$
- Najmanji element u Min-heap strukturi je koren
- Za heapsort algoritam koristi se Maks-heap struktura

# … HEAPSORT ALGORITAM



- Kako je heap struktura vrsta binarnog stabla, onda je visina stabla broj čvorova duž najdužeg puta od korena do lista
- Kako se heap bazira na binarnom stablu i kako su operacije nad binarnim stablom proporcionalne visini stabla, onda je njihova kompleksnost negde u rangu $\boldsymbol{O}(\log \boldsymbol{n})$
- Generalno je visina binarnog stabla $\Theta(\log \boldsymbol{n})$
- Heapsort algoritam oslanja se na karakteristike Maks-heap strukture, onda je važno održati karakteristike Maks-heap strukture (implementiraćemo algoritam za ovo)
- Potrebno je implementirati i algoritam za izgradnju Maks-heap strukture iz proizvoljnog niza
- I na kraju sam algoritam za sortiranje

# ODRŽAVANJE KARAKT. MAKS-HEAP-A …



- Napravićemo MAX-HEAPIFY proceduru
- Ulaz: niz ***A***, i indeks ***i*** elementa iz niza
- Pretpostavka je da **LEFT*(i)*** i **RIGHT*(i)*** podstabla imaju osobinu Maks-heap strukture, ali $A[i]$ je možda manji od svojih podređenih čvorova što kvari uslov Maks-heap strukture
- MAX-HEAPIFY procedura omogućuje da se $A[i]$ spusti niže niz heap kako bi se ponovo uspostavila osobina Maks-heap strukture, kako bi uslov bio ispunjen u potpunosti, tj. i da celo ***i*** podstablo bude u skladu sa Maks-heap strukturom

# … ODRŽAVANJE KARAKT. MAKS-HEAP-A …



```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

# … ODRŽAVANJE KARAKT. MAKS-HEAP-A …



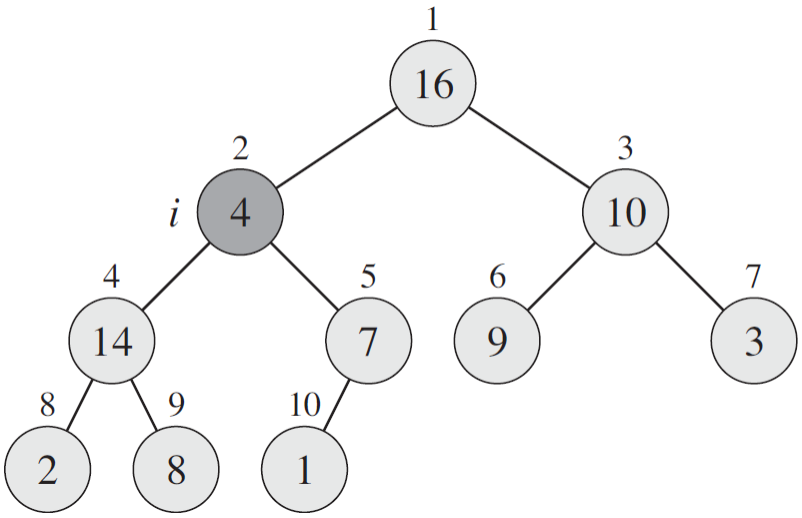
- Algoritam je rekurzivan
- U svakom koraku se određuje koji je najveći od $A[i]$, $A[LEFT(i)]$, i $A[RIGHT(i)]$ i njegov indeks se čuva
- Ako je najveći $i$-ti element, onda heap već ima osobinu Maks-heap strukture
- U suprotnom, neko od podređenih čvorova je najveći, i $A[i]$ će se zameniti sa $A[largest]$ tako da $i$-ti element i njegovi podređeni čvorovi zadovoljavaju osobinu Maks-heap strukture
- Posle ove zamene $largest$ sadrži sada originalnu $A[i]$ vrednost
- Novo podstablo koje počinje sa $A[i]$ može da kvari osobinu Maks-heap strukture, pa je potrebno i to propustiti kroz MAX-HEAPIFY proceduru

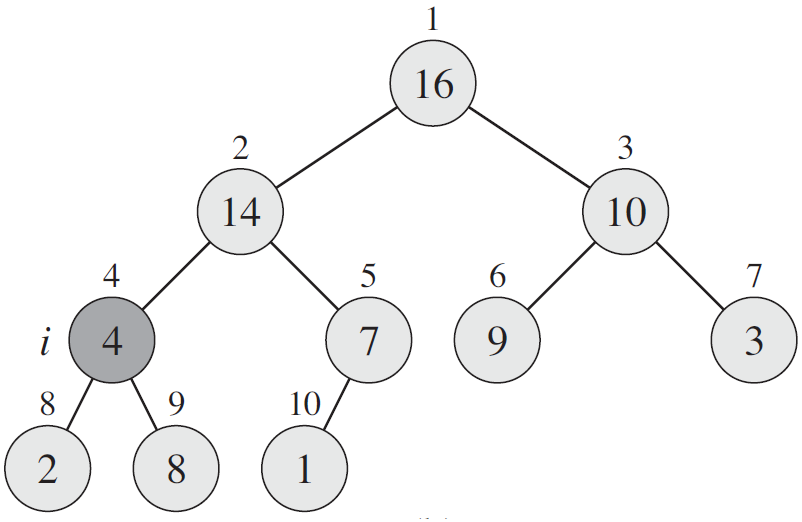
```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

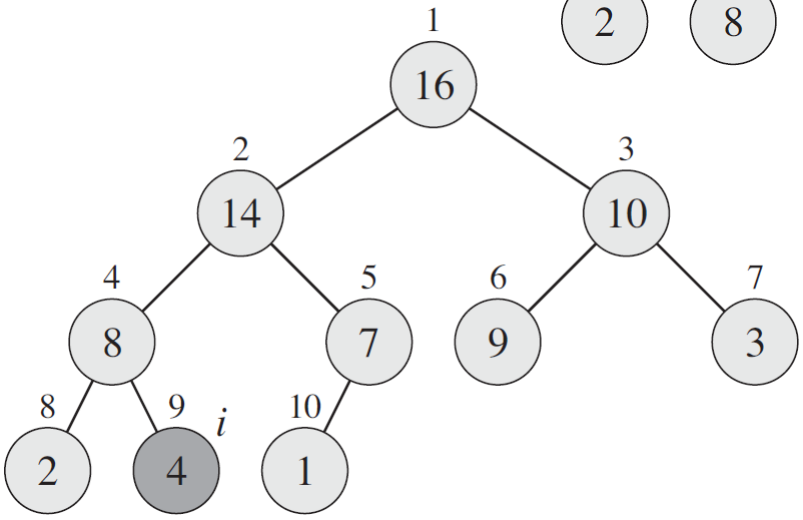
# … ODRŽAVANJE KARAKT. MAKS-HEAP-A …



- Primer rada MAX-HEAPIFY procedure



# … ODRŽAVANJE KARAKT. MAKS-HEAP-A



- Vreme izvršavanja MAX-HEAPIFY procedure na jednom nivou, tj. vreme potrebno da se sredi $\boldsymbol{A[i]}$, $\boldsymbol{A[LEFT(i)]}$, i $\boldsymbol{A[RIGHT(i)]}$ iznosi $\boldsymbol{\Theta(1)}$
- Na to treba dodati vreme potrebno da se sredi sada sledeće podstablo nastalom zamenom ***i***-tog čvora i ***largest*** čvora
- Podstablo u najgorem slučaju ima samo polovinu listova, te rekurzivni poziv smanjuje problem $\boldsymbol{2n/3}$
- Na osnovu svega ovoga vreme izvršavanja je:

$$\boldsymbol{T(n) = T\left(\frac{2n}{3}\right) + \Theta(1)}$$

  - $\boldsymbol{a = 1}$ i $\boldsymbol{b = 3/2}$, $\boldsymbol{f(n) = 1}$ tako da imamo $\boldsymbol{n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1}$
  - Važi 2 slučaj master teoreme, $\boldsymbol{f(n) = \Theta\left(n^{\log_b a}\right) = \Theta(1)}$, te je rešenje $\boldsymbol{T(n) = \Theta(\log n)}$

# IZGRADNJA HEAP-A …



- Transformacija proizvoljnog niza $\boldsymbol{A[1 \ldots n]}$ u heap, pri čemu je $\boldsymbol{n = A.length}$ je vrlo jednostavna i može se realizovati primenom MAX-HEAPIFY procedure
- U svakom nizu koji predstavlja heap poslednjih $\boldsymbol{A\left[\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \ldots n\right]}$ predstavlja listove binarnog stabla, i svaki je za sebe heap sa samo jednim elementom
- To znači da je potrebno proći kroz preostale čvorove i primeniti MAX-HEAPIFY procedure nad svakim od njih
- Primenom prethodno opisanog algoritma nastaje algoritam za izgradnju heap strukture, BUILD-MAX-HEAP

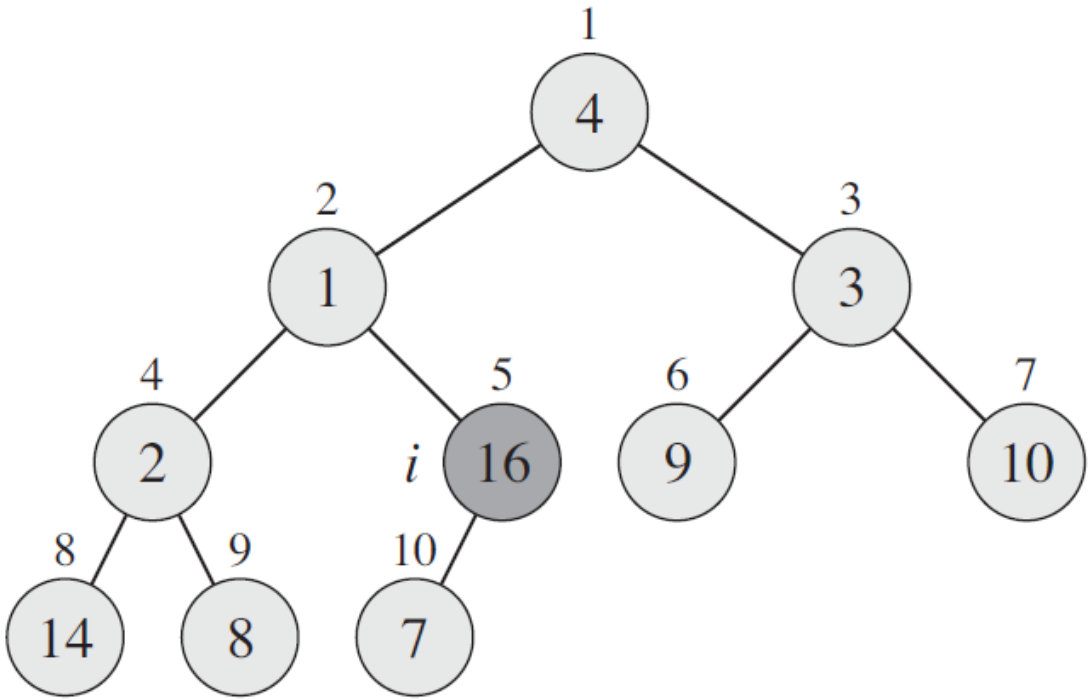
```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

# … IZGRADNJA HEAP-A …



- Primer kako algoritam radi

| $A$ | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|

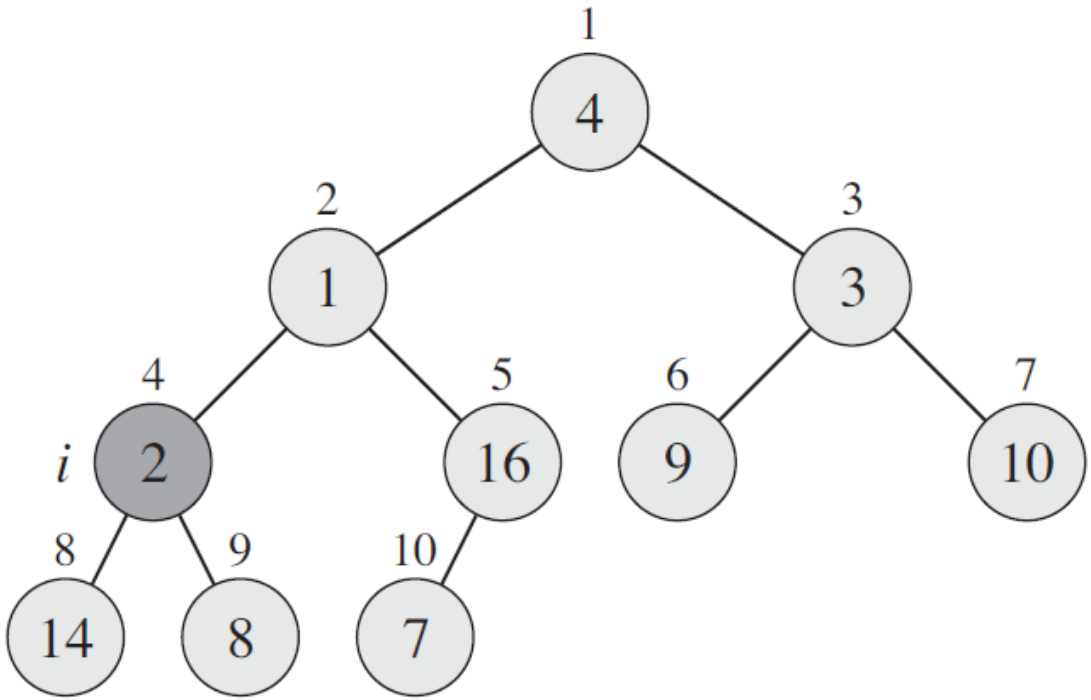


# … IZGRADNJA HEAP-A …



- Primer kako algoritam radi

| $A$ | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|

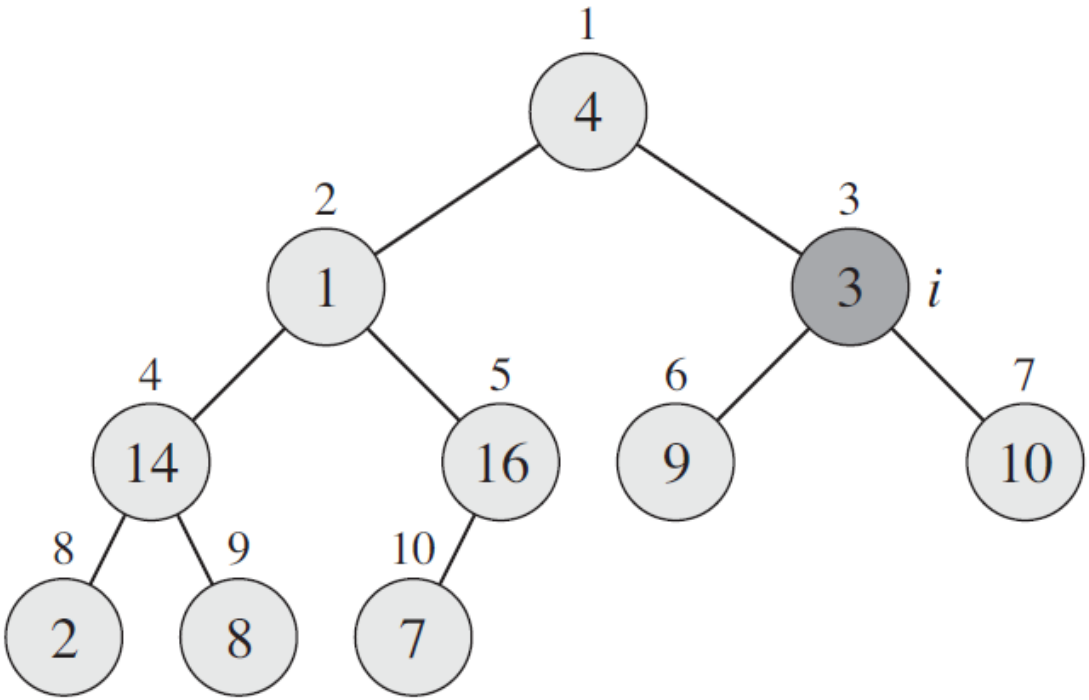


# … IZGRADNJA HEAP-A …



- Primer kako algoritam radi

| $A$ | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|

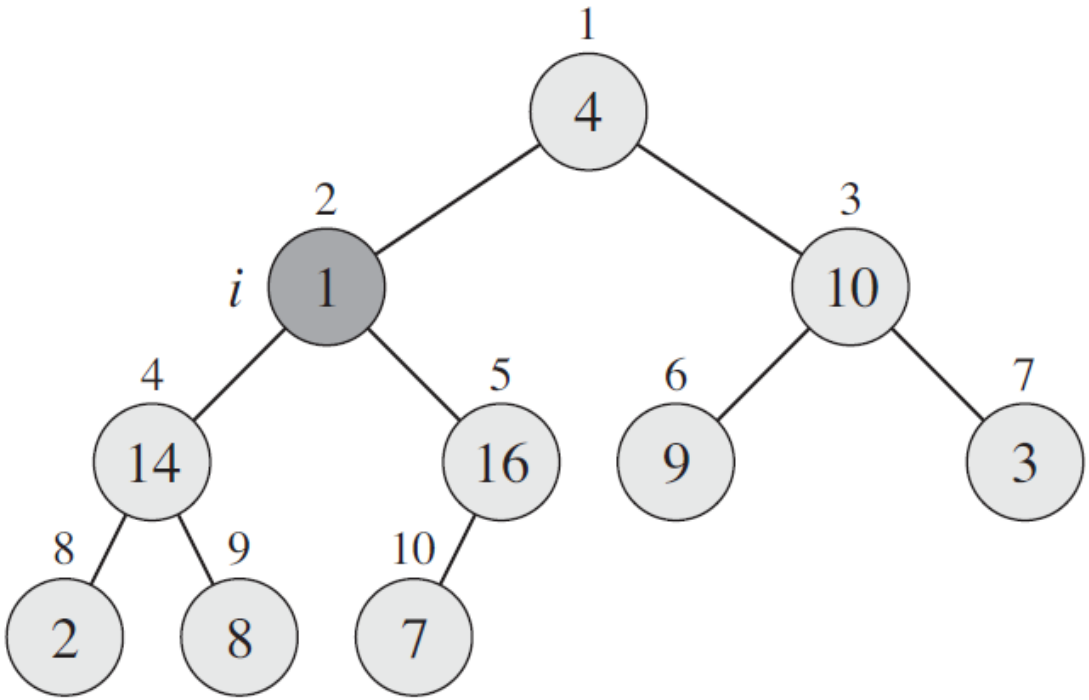


# … IZGRADNJA HEAP-A …



- Primer kako algoritam radi

| $A$ | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|

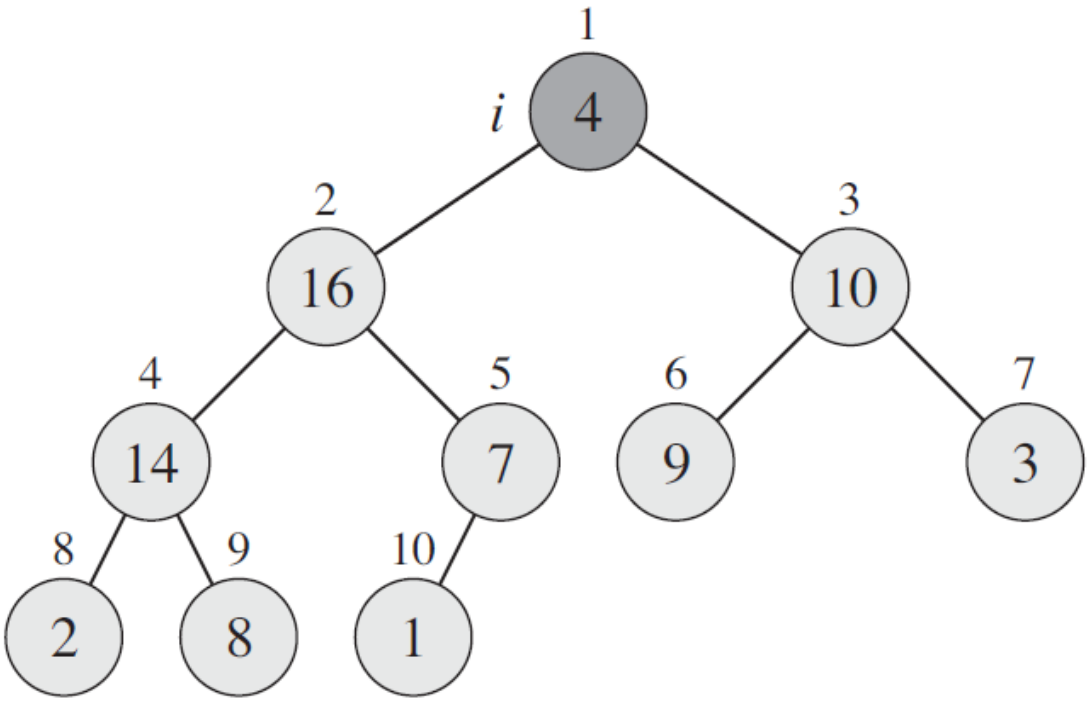


# … IZGRADNJA HEAP-A …



- Primer kako algoritam radi

| $A$ | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|

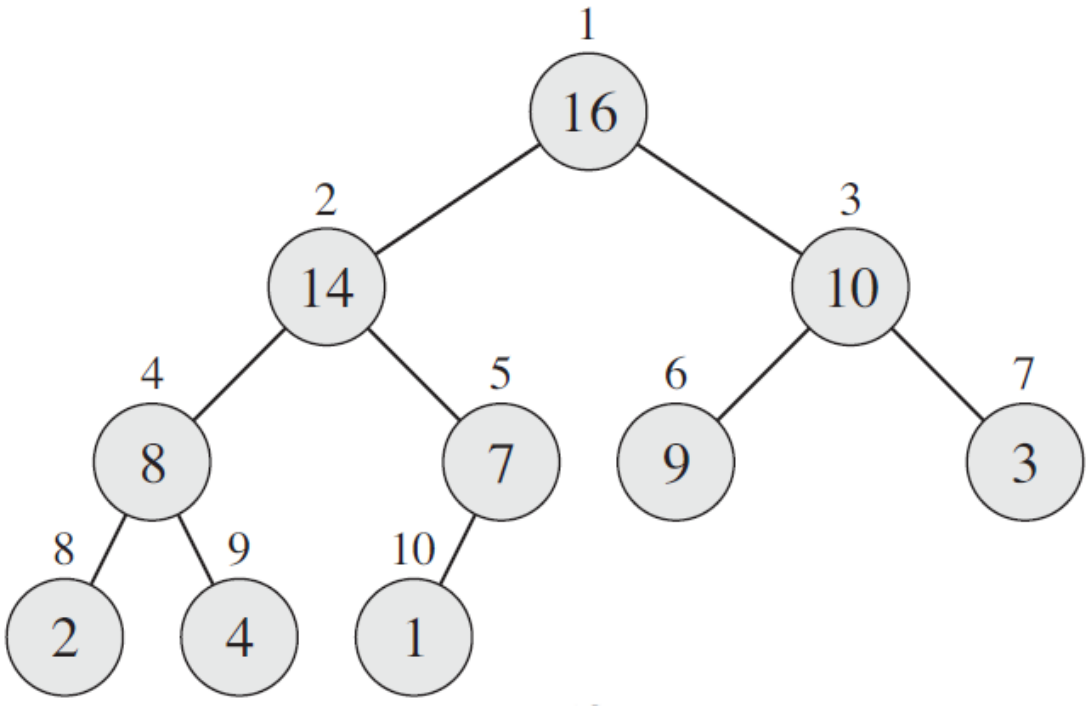


# … IZGRADNJA HEAP-A …



- Primer kako algoritam radi

| A | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|



# … IZGRADNJA HEAP-A …



- Ispravnost algoritma i pristupa, može se dokazati invarijantošću petlje
- Na početku svake iteracije ***for*** petlje svi čvorovi ***i + 1***, ***i + 2***, …, ***n*** su koreni Maks-heap strukture,

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

  - tj. njihova podstabla su uređena kao binarna stabla i održavaju osobinu Maks-heap strukture

- Dokaz?
  - **inicijalizacija**
  - **održivost**
  - **završavanje**

# … IZGRADNJA HEAP-A …



- **Inicijalizacija**
  - Pre prve iteracije ***for*** petlje $\boldsymbol{i = \left\lfloor \frac{n}{2} \right\rfloor}$ svaki čvor $\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, \ldots, n$ je list i samim tim koren trivijalne Maks-heap strukture

- **Održivost**
  - Pre poziva MAX-HEAPIFY procedure listovi ***i*** čvora su i jedan i drugi imaju veći indeks od njega, a pošto su prethodno obrađeni, oni su ujedno i čvorovi svojih Maks-heap struktura, što znači da je zadovoljen uslov da su svi čvorovi ***i + 1***, ***i + 2***, …, ***n*** su koreni Maks-heap strukture
  - Poziv MAX-HEAPIFY procedure će se pobrinuti da, ako je potrebno dođe do obrtanja čvora ***i*** sa njegovim listovima koji su veći od njega, pa će i ***i*** čvor postati koren Maks-heap strukture
  - Ali što je najvažnije, MAX-HEAPIFY procedure neće upropastiti osobinu Maks-heap strukture podređenim čvorovima i stablima, tako da smanjivanje ***i*** iteratora za jedan održava invarijantnost petlje

# … IZGRADNJA HEAP-A



- **Završavanje**
  - Kada se petlja završi, ***i = 0***.
  - Praćenjem inverijantnosti petlje, svaki čvor ***1***, ***2***, …, ***n*** je koren Maks-heap struktura a naročito čvor 1

- Vreme izvršavanja se može lako izračunati
  - Svaki poziv MAX-HEAPIFY procedure košta $\boldsymbol{O}(\log \boldsymbol{n})$
  - Kako BUILD-MAX-HEAP procedura pravi $\boldsymbol{O}(\boldsymbol{n})$ takvih poziva
  - Setite se matematike asimptotkse notacije. Koje je rešenje?
  - Vreme izvršavanja je $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$, ali nije strogo
  - Da li se može dobiti i strožija?
  - $\boldsymbol{O}(\boldsymbol{n})$ (pogledati ItA za dokaz i objašnjenje)

# HEAPSORT ALGORITAM …



- Algoritam započinje izgradnjom Maks-heap strukture upotrebom BUILD-MAX-HEAP procedure
- Kako je sada najveći element na $\boldsymbol{A[1]}$, treba ga staviti na njegovo konačno mesto zamenom sa $\boldsymbol{A[n]}$
- Izbacivanjem $\boldsymbol{A[n]}$ elementa, tako što se ***A.heap-size*** umanji za jedan, struktura je i dalje Maks-heap, samo novi koren možda kvari uslov Maks-heap strukture
- Potrebno je primeniti MAX-HEAPIFY(A, 1) i dobiti Maks-heap strukturu od niza $\boldsymbol{A[1 \ldots n-1]}$
- Sad treba ponoviti zamenu $\boldsymbol{A[1]}$ sa poslednjim elementom u nizu $\boldsymbol{A[n-1]}$, ponoviti izbacivanje poslednjeg elementa i nastaviti postupak sve dok nisu svi elementi sortirani

# … HEAPSORT ALGORITAM …



```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

- Izvršavanje BUILD-MAX-HEAP procedure traje $\boldsymbol{O(n \log n)}$, a poziv svake od $\boldsymbol{n-1}$ MAX-HEAPIFY procedura traje $\boldsymbol{O(\log n)}$ što znači da čitav HEAPSORT algoritam traje $\boldsymbol{O(n \log n)}$

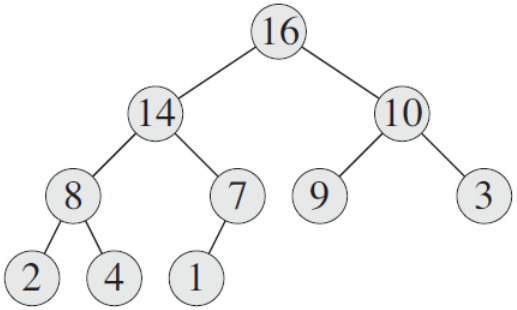
# … HEAPSORT ALGORITAM …



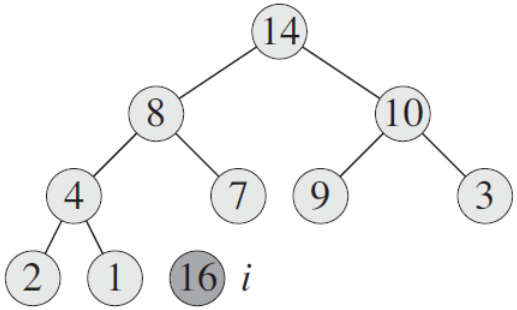
- Primer:

| $A$ | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|

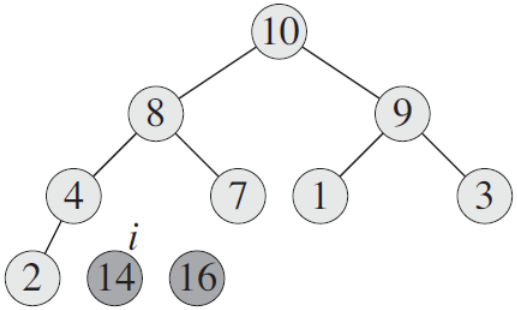
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```



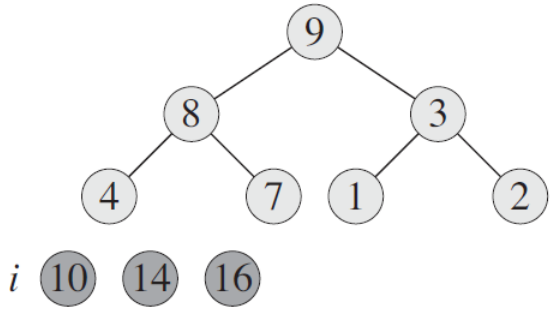
(a)



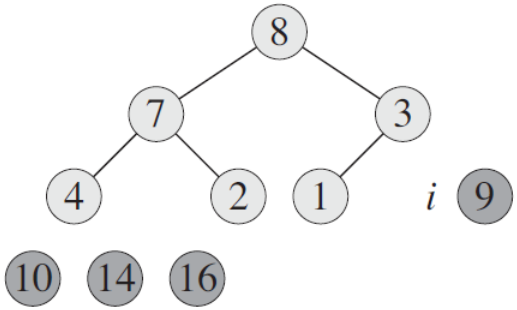
(b)



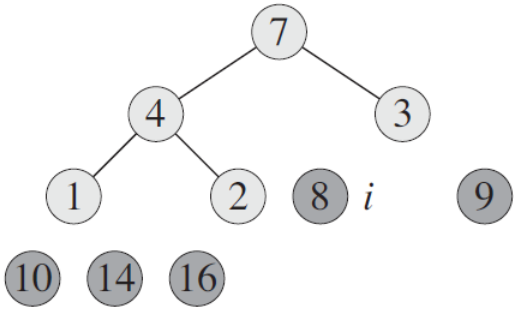
(c)



(d)



(e)



(f)

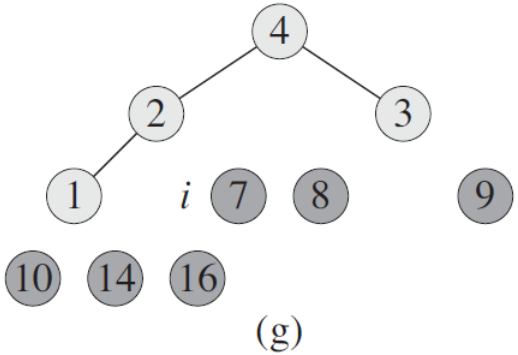
# … HEAPSORT ALGORITAM



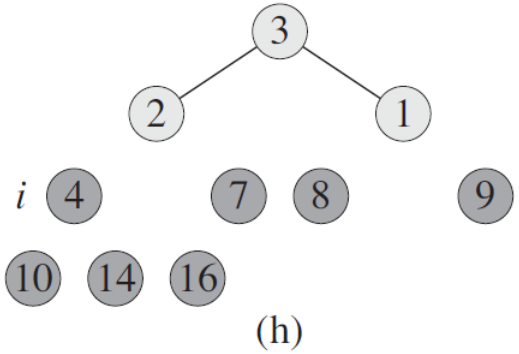
- Primer:

| $A$ | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|

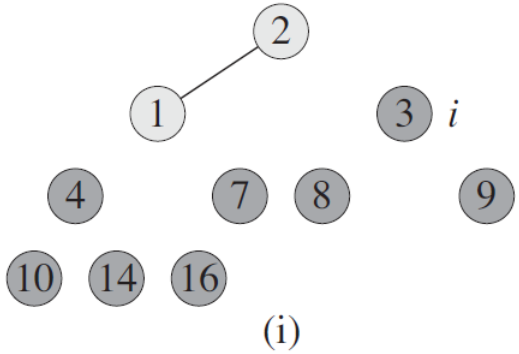
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```



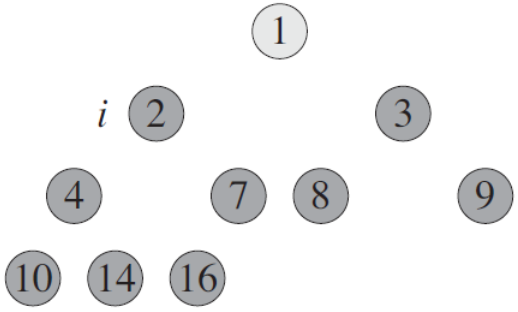
(g)



(h)



(i)



(j)

| $A$ | 1 | 2 | 3 | 4 | 7 | 8 | 9 | 10 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|

(k)

# REDOVI PRIORITETA

# … REDOVI PRIORITETA …



- Iako je Heap sort odličan algoritam, postoje bolji algoritmi koji su se bolje pokazali u praksi
- Međutim sama Heap struktura ima veliki broj primena
- Jedna od vrlo popularnih primena Heap strukture i sortiranja oko nje jeste red prioriteta (*priority que*)
- Red prioriteta je struktura u kojoj se čuva skup $S$ nekih elemenata
- Svakom elementu je pridružen ključ na osnovu kojeg se određuje prioritet elementa
- Kao i Heap struktura, i red prioriteta mogu biti maksimalni ili minimalni red prioriteta
  - Kod maksimalnog reda prioriteta, prioritet imaju elementi sa većom vrednošću ključa
  - Kod minimalnog reda prioriteta, prioritet imaju elementi sa manjom vrednošću ključa

# … REDOVI PRIORITETA …



- Za primer se koristi maksimalni red prioriteta koji koristi Maks-heap strukturu
- Minimalni red prioriteta je sličan samo se kao osnova koristi Min-heap struktura
- Maksimalni red prioriteta podržava sledeće operacije:
  - $\boldsymbol{Insert}(\boldsymbol{S}, \boldsymbol{x})$ – ubacivanje elementa $\boldsymbol{x}$ u red što je ekvivalentno operaciji $\boldsymbol{S} = \boldsymbol{S} \cup \{\boldsymbol{x}\}$
  - $\boldsymbol{Maximum}(\boldsymbol{S})$ – vraća element iz reda sa najvećim ključem
  - $\boldsymbol{Extract - Maximum}\ (\boldsymbol{S})$ – briše i vraća element iz reda sa najvećim ključem
  - $\boldsymbol{Increase - Key}(\boldsymbol{S}, \boldsymbol{x}, \boldsymbol{k})$ – povećava vrednost ključa elementa $\boldsymbol{x}$ u redu $\boldsymbol{S}$ na vrednost $\boldsymbol{k}$
- Minimalni red prioriteta podržava slične operacija: $\boldsymbol{Insert}$, $\boldsymbol{Minimum}$, $\boldsymbol{Extract - Minimum}$, $\boldsymbol{Decrease - Key}$

# … REDOVI PRIORITETA …



- $\boldsymbol{Maximum(S)}$ baziran na Maks-heap strukturi, HEAP-MAXIMUM se izvršava u $\Theta(\mathbf{1})$ vremena

```
HEAP-MAXIMUM(A)
1  return A[1]
```

# … REDOVI PRIORITETA …



- $\boldsymbol{Extract - Maximum(S)}$ baziran na Maks-heap strukturi, EXTRACT-HEAP-MAXIMUM vrlo je sličan Heap sort algoritmu

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

- Proverava se prvo da li Maks-heap struktura ima uopšte elemenata

# … REDOVI PRIORITETA …



- Proverava se da li Maks-heap struktura ima uopšte elemenata
- Iščita se prvi element u izlaznu promenljivu
- Vrednost poslednjeg elementa se upiše na mesto prvog elementa

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

- Smanji se veličina Maks-heap strukture za jedan (efektivno se vrši brisanje poslednjeg elementa, ali kako je on kopiran u prvi, to znači da se obrisao prvi element)
- Zatim se poziva MAX-HEAPIFY procedura kako bi se ponovo uspostavila osobina Maks-heap strukture
- Vreme izvršavanja ovog algoritma je $\boldsymbol{O}(\log \boldsymbol{n})$, jer se sve preko MAX-HEAPIFY procedure izvršava u konstantnom vremenu

# … REDOVI PRIORITETA …



- $\boldsymbol{Increase - Key(S, x, k)}$ baziran na Maks-heap strukturi, HEAP-INCREASE-KEY procedura

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

- Indeks $\boldsymbol{i}$ u Maksimalnom redu prioriteta identifikuje elemenat kojem želimo da povećamo ključ
- Prvo se proveri da li je nova vrednost veća od stare
- Zatim se vrednost ključa postavi na novu vrednost $\boldsymbol{A[i] = key}$

# … REDOVI PRIORITETA …



- Kako povećanje vrednosti ključa može da pokvari osobinu Maks-heap strukture, procedura gura ključ gore uz stablo do korena i postavlja ga na svoje mesto slično koracima iz Insertion sortiranja

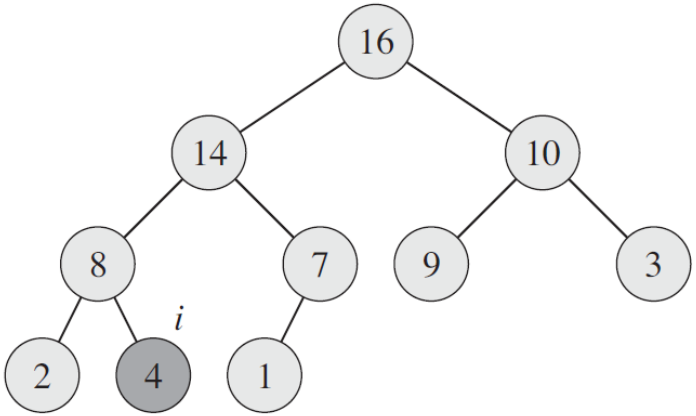
```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

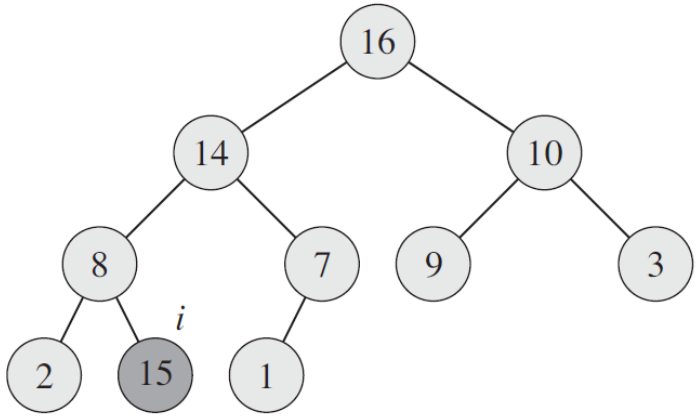
- Kako se ključ kreće uz stablo, poredi se sa svojim nadređenim čvorom i ako je veći, zamenjuje njihova mesta
- Algoritam nastavlja dalje sve dok ne naiđe na prvi čvor čija je vrednost veća od ključa koji se povećao
- Vreme izvršavanja ovog algoritma nad $n$-elemenata Maks-heap strukture je $O(\log n)$, jer putanja koju treba ispratiti dugačka $O(\log n)$

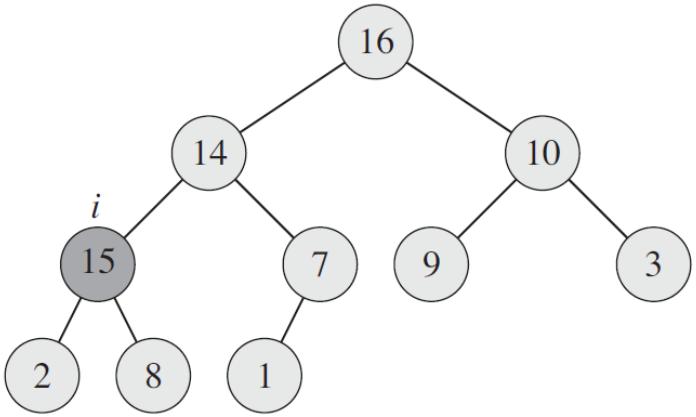
# … REDOVI PRIORITETA …

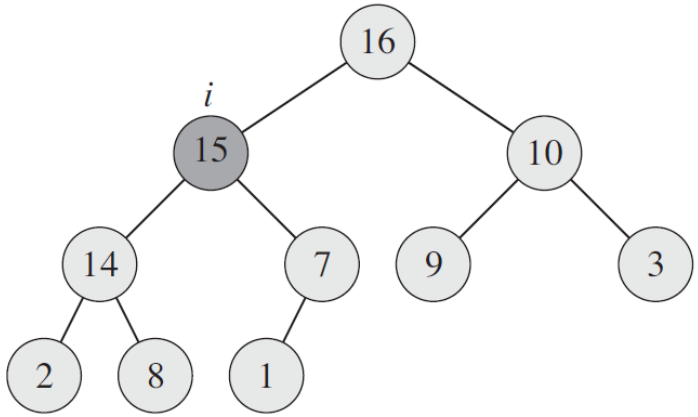


- Primer kako radi HEAP-INCREASE-KEY procedura



# … REDOVI PRIORITETA



- $\boldsymbol{Insert(S, x)}$ baziran na Maks-heap strukturi je MAX-HEAP-INSERT procedura

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

- Algoritam je vrlo jednostavan
  - Prvo se na kraj Maks-heap strukture doda novi element čiji se ključ stavi na najmanju moguću vrednost
  - Zatim se poziva HEAP-INCREASE-KEY procedura kako bi se vrednost ključa novog elementa postavila na željenu vrednost i kako bi se održala osobina Maks-heap strukture
- Očigledno je vreme izvršavanja ovog algoritma $\boldsymbol{O}(\log \boldsymbol{n})$. Zašto?

# QUICKSORT

# QUICKSORT ALGORITAM …



- Ima **worse-case** vreme izvršavanja od $\Theta(\boldsymbol{n^2})$ nad nizom od $\boldsymbol{n}$ elemenata
- Međutim, u praksi ovaj algoritam pobeđuje mnoge jer ima očekivano vreme izvršavanja, u proseku, od $\Theta(\boldsymbol{n}\log\boldsymbol{n})$
- Međutim, što je još važnije, konstantne vrednosti skrivene u $\Theta(\boldsymbol{n}\log\boldsymbol{n})$ su vrlo male
- Ima prednost da radi takozvano sortiranje u mestu (*sorting in place*)
- Zavadi-pa-vladaj
- Predložio ga C.A.R. Hoare, 1962.

# … QUICKSORT ALGORITAM …



- Opis algoritma nad podnizom $A[p \ldots r]$:

1. **Divide**: podeli podniz $A[p \ldots r]$ na dva podniza (neki može da bude i prazan), tako da su svi elementi podniza $A[p \ldots q-1]$ manji ili jednaki $A[q]$ i svi elementi podniza $A[q+1 \ldots r]$ veći ili jednaki $A[q]$. $q$ se računa kao deo ***divide*** procedure

| $A[p \ldots q-1] \leq$ | $A[q]$ | $\leq A[q+1 \ldots r]$ |
|---|---|---|

2. **Conquer**: sortiraj podnizove $A[p \ldots q-1]$ i $A[q+1 \ldots r]$ rekurzivnim pozivanjem QUICKSORT algoritma nad njima
3. **Combine**: sortirani podnizovi se kombinuju na trivijalan način

- **Ključ je linearno vreme izvršavanja procedure za parcionisanje**

# … QUICKSORT ALGORITAM …



- Sledeća procedura implementira quicksort algoritam

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

- Da bi se sortirao čitav niz A, inicijalni poziv izgleda $\boldsymbol{QUICKSORT(A, 1, A.length)}$

# … QUICKSORT ALGORITAM …



- Očigledno je ključ procedura za deljenje niza jer preuređuje podniz $\boldsymbol{A[p \ldots r]}$, ali u mestu

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

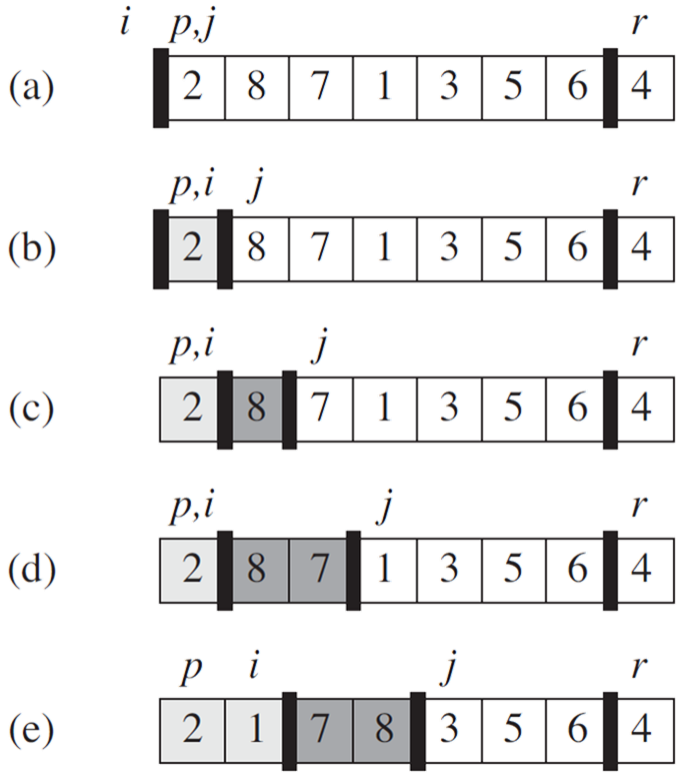
- Procedura uvek selektuje $\boldsymbol{x = A[r]}$ kao pivot tačka za parcionisanje podniza $\boldsymbol{A[p \ldots r]}$

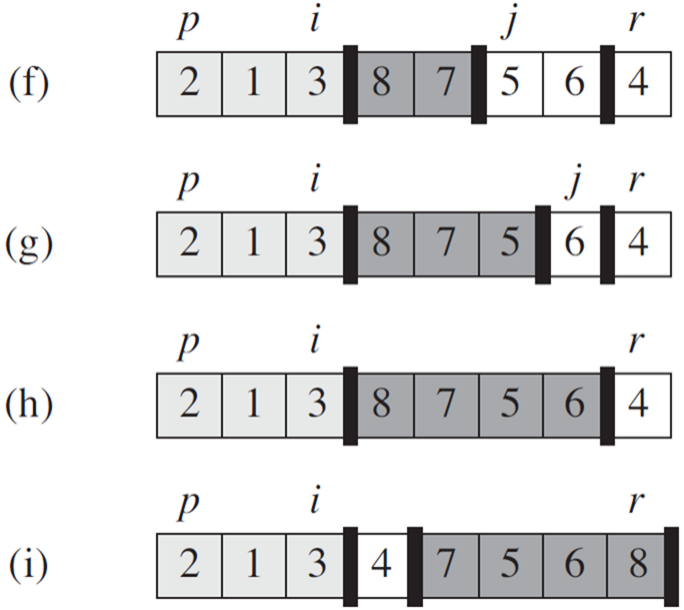
# … QUICKSORT ALGORITAM …



- Primer kako radi PARTION algoritam

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```



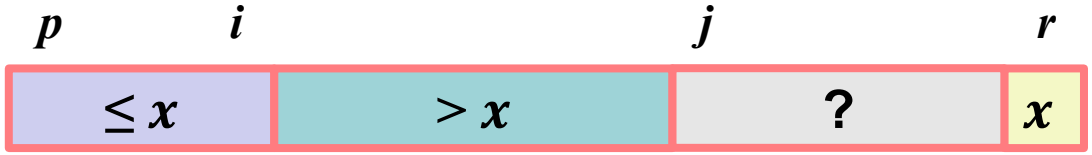



# … QUICKSORT ALGORITAM …



```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

- Tokom izvršavanja procedure podniz se deli na četiri dela tako da svaki deo zadovoljava sledeće:




- Na početku svake iteracije ***for*** petlje svaki deo i dalje zadržava iste osobine
- Važi sledeća invarijantnost petlje:
  - Na početku svake iteracije (3-6 linija), za ***k***-ti element niza važi
  1. Ako je $p \leq k \leq i$, onda je $A[k] \leq x$
  2. Ako je $i + 1 \leq k \leq j - 1$, onda je $A[k] > x$
  3. Ako je $k = r$, onda je $A[k] = x$
- Elementi između $j$ i $r - 1$ ne zadovoljavaju ni jedan od uslova (još)

# … QUICKSORT ALGORITAM …



```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

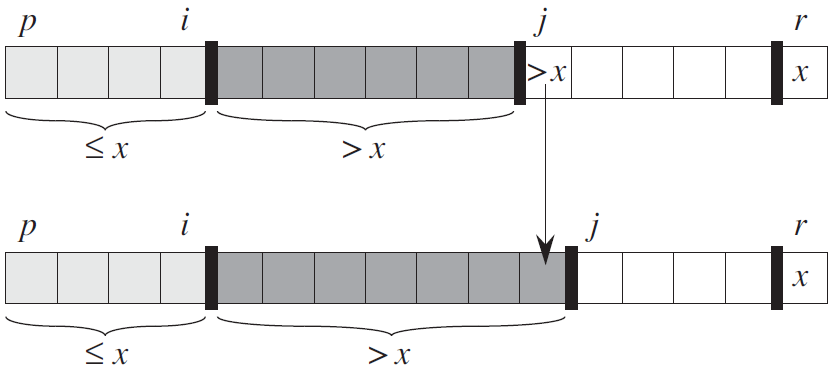
- Dokaz invarijantnosti petlje
- **Inicijalizacija**
  - Pre inicijalizacije petlje, $\boldsymbol{i = p - 1}$ i $\boldsymbol{j = p}$
  - Kako između $\boldsymbol{p}$ i $\boldsymbol{i}$ nema nikakve vrednosti i kako nikakva vrednost ne leži između $\boldsymbol{i + 1}$ i $\boldsymbol{j - 1}$, prva dva uslova invarijantnosti petlje su zadovoljena
  - Dodela vrednosti u 1. liniji zadovoljava treći uslov

# … QUICKSORT ALGORITAM …



```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

- Dokaz invarijantnosti petlje (nastavak)
- **Održivost**
  - Ako je $A[j] > x$ jedina moguća akcija je da se poveća $j$, tada drugi uslov važi za $A[j-1]$ i sve ostalo se ne menja



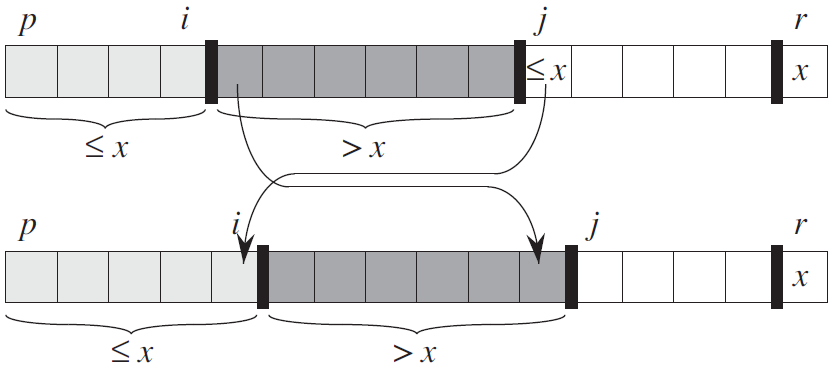

  - U zavisnosti od ispunjenosti uslova u liniji 4

# … QUICKSORT ALGORITAM …



```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

- Dokaz invarijantnosti petlje (nastavak)
- **Održivost (nastavak)**
  - Ako je $A[j] \leq x$, povećava se $i$, $A[i]$ i $A[j]$ menjaju mesto i povećava se $j$




  - Zbog zamene mesta, sada je $A[i] \leq x$, te je prvi uslov zadovoljen
  - Zbog zamene, sada je i $A[j-1] > x$, jer elemenat koji je prebačen u $A[j-1]$ na osnovu invarijantnosti petlje svakako veći od $x$

# … QUICKSORT ALGORITAM



```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

- Dokaz invarijantnosti petlje (nastavak)
- **Završavanje**
  - Na kraju je $j = r$
  - Svaki element niza je u jednoj od 3 grupe iz invarijantnosti petlje
  - Niz je podeljen u tri grupe: elementi koji su manji ili jednaki $x$, veći od $x$, i singlton koji sadrži samo $x$
- Poslednje dve linije PARTION procedure završavaju tako što menjaju pivot elemenat sa prvim elementom s leve strane većim od njega
- Na taj način izlaz iz procedure zadovoljava specifikaciju stavljenu pred ***divide*** korak, zapravo posle 2. koraka QUICKSORT algoritma, $A[q]$ je strogo manji od svih elemenata podniza $A[q + 1 \ldots r]$
- Vreme izvršavanja PARTION procedure nad nizom $A[p \ldots r]$ je $\Theta(n)$

# QUICKSORT ALGORITAM – ANALIZA …



- Pretpostaviti da su svi elementi ulaznog niza različiti
- Za nizove sa dupliranim vrednostima, u praksi, postoje daleko bolji, efikasniji algoritmi
- U najgorem slučaju particija niza traje

$$\boldsymbol{T(n) = \Theta(n^2)}$$

- Šta bi bio najgori mogući raspored elemenata?
- Da su sortirani u obrnutom redosledu
- Zašto je to problem?
- Jedna strana particije uvek nema elemenata
- Te se particija realizuje ili oko min ili oko maks elementa
- To znači da se problem uvek smanjuje samo za jedan element

# … QUICKSORT ALGORITAM – ANALIZA …



- Te se particija realizuje ili oko minimalnog ili oko maksimalnog elementa
- To znači da se problem uvek smanjuje samo za jedan element
- Kako da to dokažemo?
- Probajte metodu supstitucije
- Koja rekurentna jednačina opisuje QUICKSORT ALGORITAM?

$$\boldsymbol{T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)}$$

gde je $\boldsymbol{q}$ iz opsega $\boldsymbol{0}$ i $\boldsymbol{n-1}$

- PARTITION deli problem na dva dela čija je ukupna veličina $\boldsymbol{n-1}$
- Pretpostavimo $\boldsymbol{T(n) \le cn^2}$, za neku konstantu $\boldsymbol{c}$

# … QUICKSORT ALGORITAM – ANALIZA …



$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

- Pretpostavimo $T(n) \le cn^2$, za neku konstantu $c$

$$T(n) \le \max_{0 \le q \le n-1} (c \cdot q^2 + c \cdot (n-q-1)^2) + \Theta(n)$$

$$= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n)$$

- $q^2 + (n-q-1)^2$ ostvaruje maksimum na krajnjim vrednostima opsega $0 \le q \le n-1$

$$\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$$

- odavde sledi

$$T(n) \le cn^2 - c(2n+1) + \Theta(n)$$

$$\le cn^2$$

# … QUICKSORT ALGORITAM – ANALIZA …



- Te se particija realizuje ili oko minimalnog ili oko maksimalnog elementa
- To znači da se problem uvek smanjuje samo za jedan element

$$\begin{aligned} \boldsymbol{T(n)} &= \boldsymbol{T(0)} + \boldsymbol{T(n-1)} + \Theta(\boldsymbol{n}) \\ &= \Theta(\mathbf{1}) + \boldsymbol{T(n-1)} + \Theta(\boldsymbol{n}) \\ &= \boldsymbol{T(n-1)} + \Theta(\boldsymbol{n}) \\ &= \Theta(\boldsymbol{n^2}) \end{aligned}$$

**(aritmetička serija)**

# … QUICKSORT ALGORITAM – ANALIZA …



- Stablo rekurzije

$$T(n) = T(0) + T(n-1) + cn$$

# … QUICKSORT ALGORITAM – ANALIZA …



- Stablo rekurzije

$$T(n) = T(0) + T(n-1) + cn$$

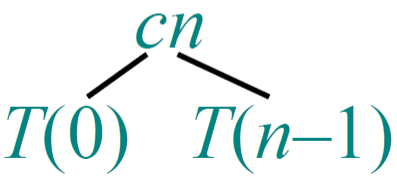
$T(n)$

# … QUICKSORT ALGORITAM – ANALIZA …



- Stablo rekurzije

$$T(n) = T(0) + T(n-1) + cn$$

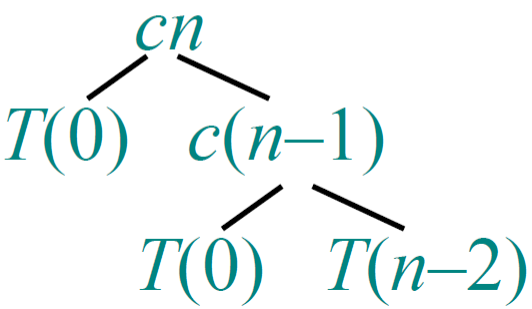


# … QUICKSORT ALGORITAM – ANALIZA …



- Stablo rekurzije

$$T(n) = T(0) + T(n-1) + cn$$

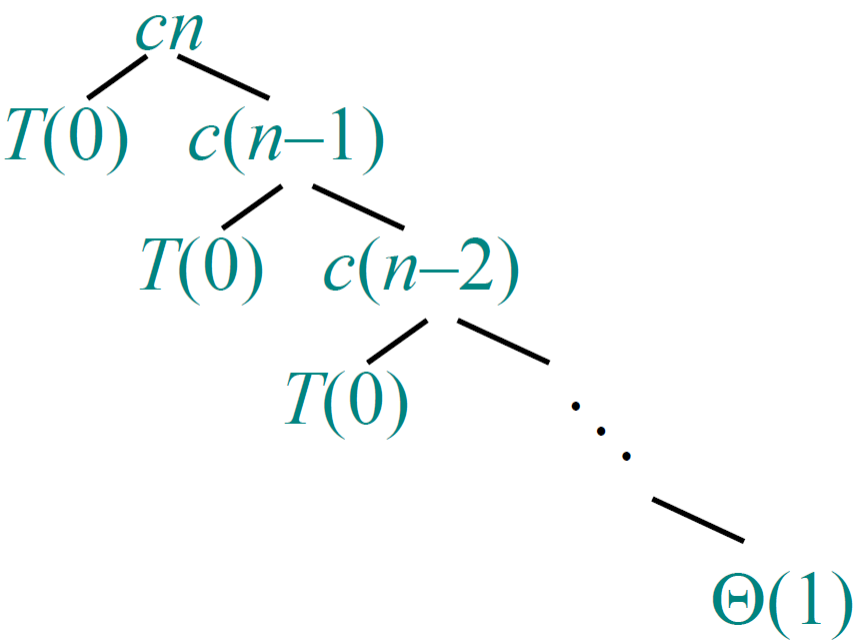


# … QUICKSORT ALGORITAM – ANALIZA …



- Stablo rekurzije

$$T(n) = T(0) + T(n-1) + cn$$

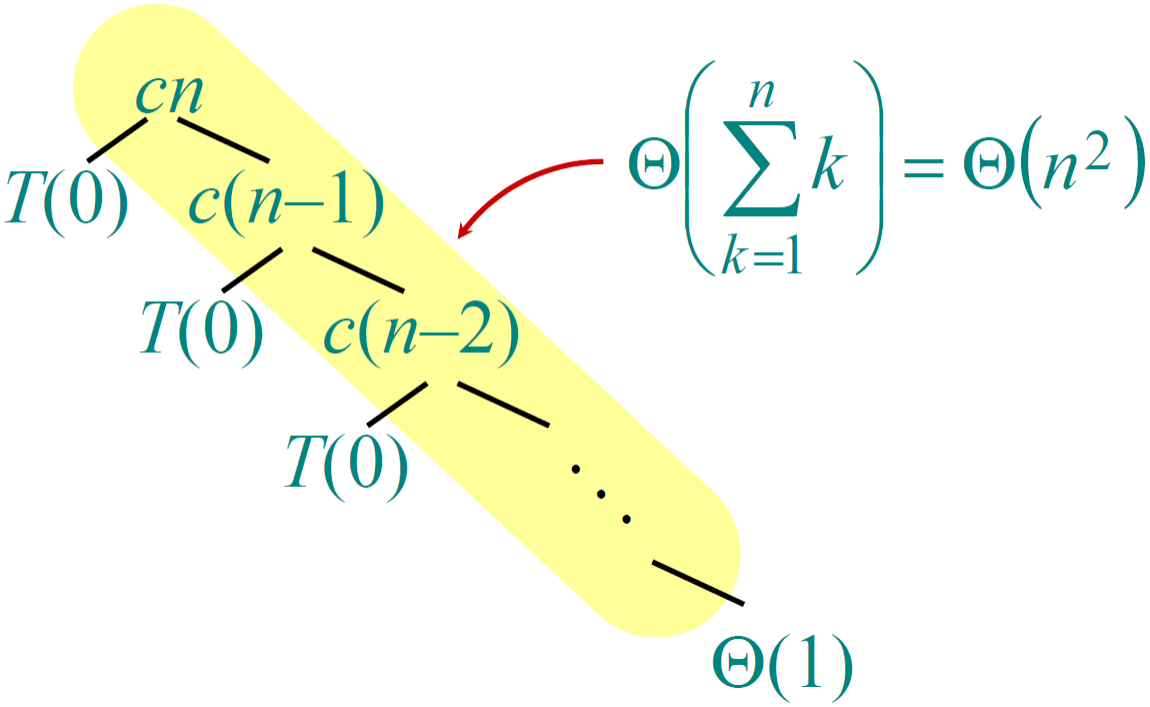


# … QUICKSORT ALGORITAM – ANALIZA …



- Stablo rekurzije

$$T(n) = T(0) + T(n-1) + cn$$

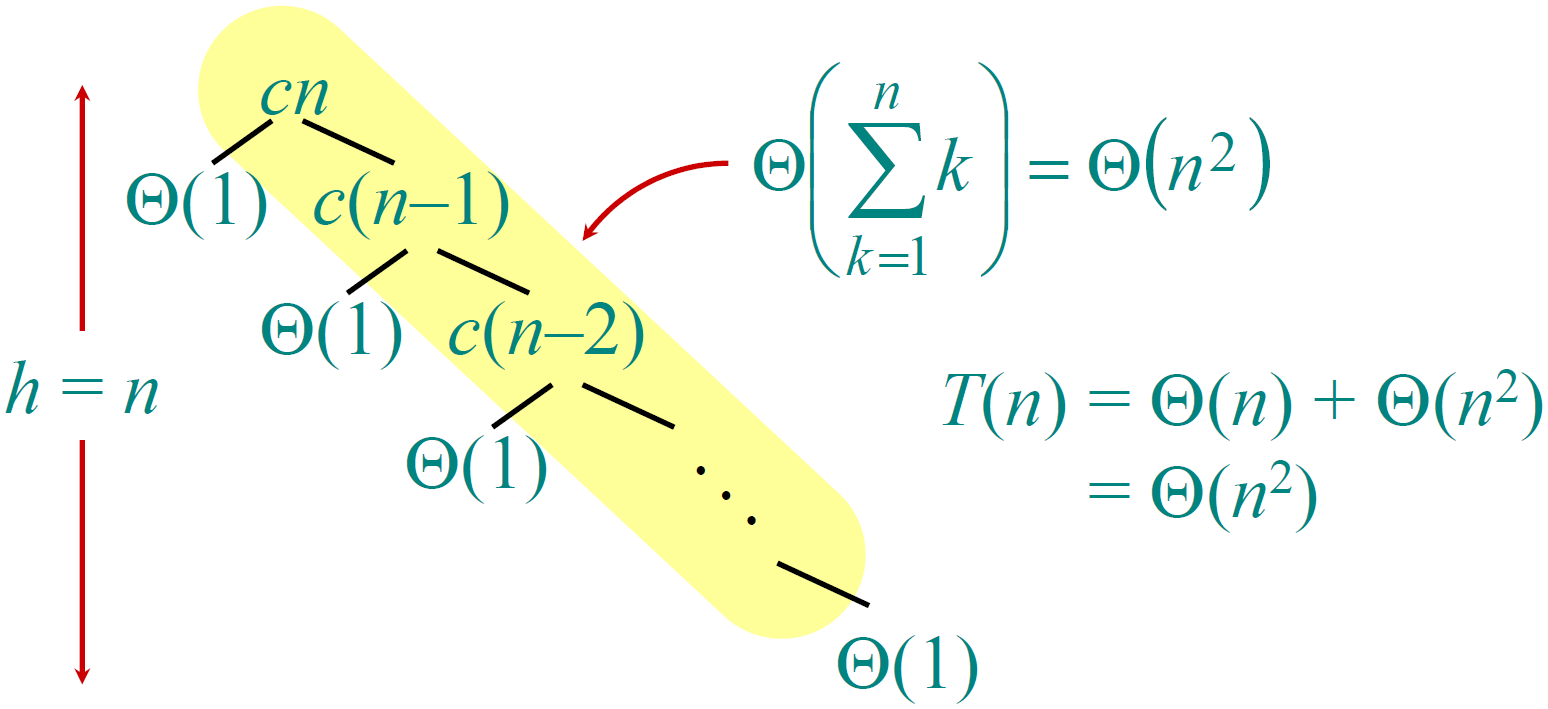


# … QUICKSORT ALGORITAM – ANALIZA …



- Stablo rekurzije

$$T(n) = T(0) + T(n-1) + cn$$



# … QUICKSORT ALGORITAM – ANALIZA …



- U najboljem slučaju, kada smo ekstremno srećni, PARTITION deli niz na dva jednaka dela

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n)$$ ***(isto kao za merge sortiranje)***

- Problem se deli na dva potproblema, veličine $\left\lfloor \frac{n}{2} \right\rfloor$ i $\left\lceil \frac{n}{2} \right\rceil - 1$, ali zaokruživanje ignorišemo

- Prosečno vreme izvršavanja QUICKSORT algoritma je bliže idealnom vremenu izvršavanja nego najgorem

- To proizilazi iz toga što balansiranost niza značajno utiče na rekurentnu jednačinu (tj. oslikava se u samoj jednačini)

# … QUICKSORT ALGORITAM – ANALIZA …



- Šta ako algoritam za parcionisanje niza uvek deli niz na jednake delove u odnosu, npr. 9 prema 1?

$$T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + \Theta(n)$$

- Šta je rešenje ove rekurentne jednačine?
- Kako da ga rešimo?
- Da probamo stablo rekurzije?

# … QUICKSORT ALGORITAM – ANALIZA …



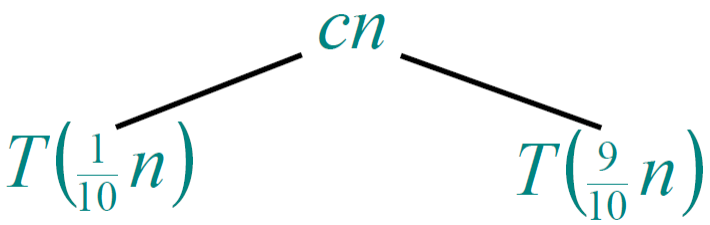
$$\boldsymbol{T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + \Theta(n)}$$

$$T(n)$$

# … QUICKSORT ALGORITAM – ANALIZA …



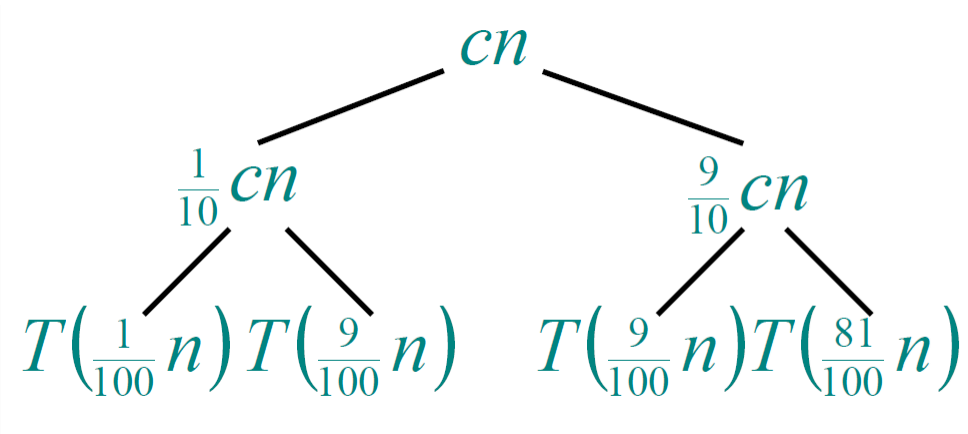
$$T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + \Theta(n)$$



# … QUICKSORT ALGORITAM – ANALIZA …



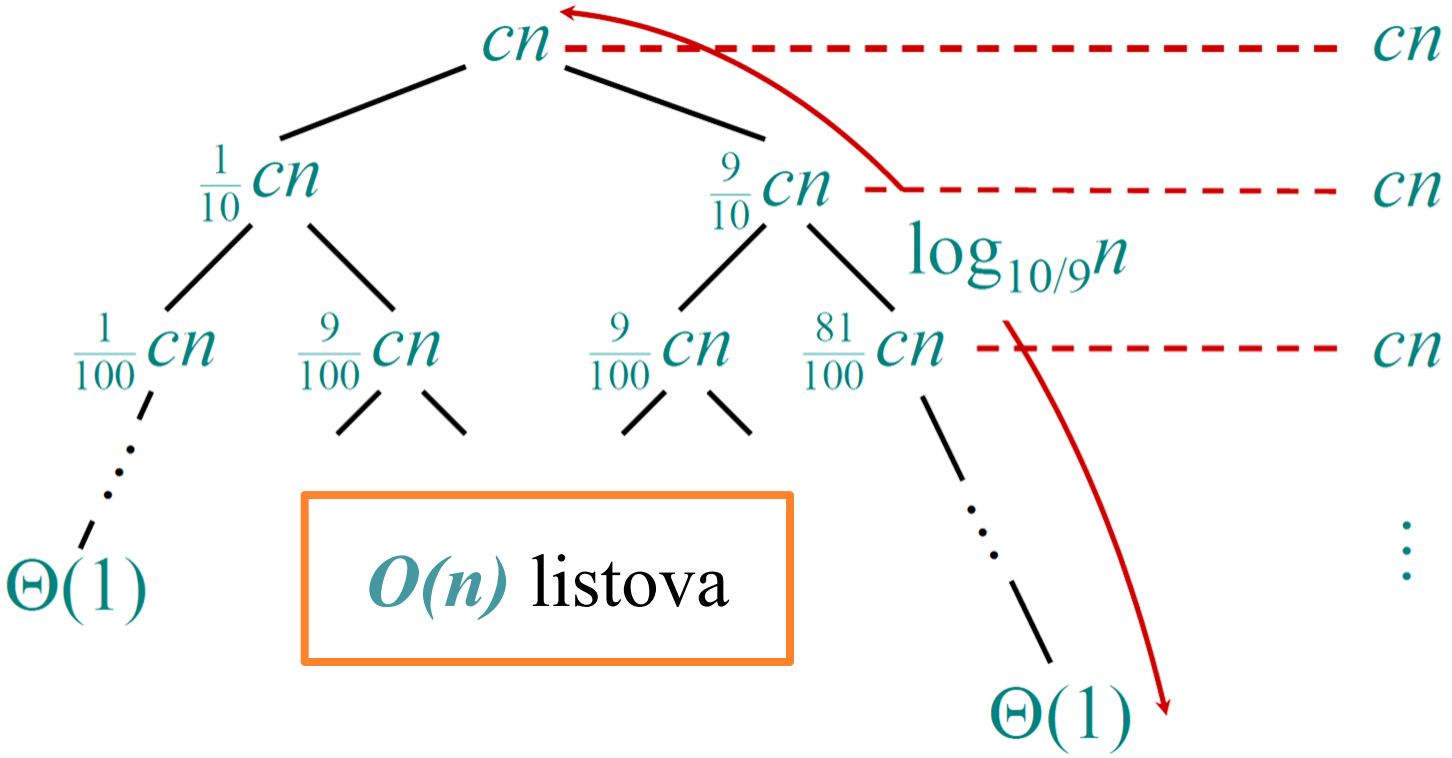
$$T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + \Theta(n)$$



# … QUICKSORT ALGORITAM – ANALIZA …



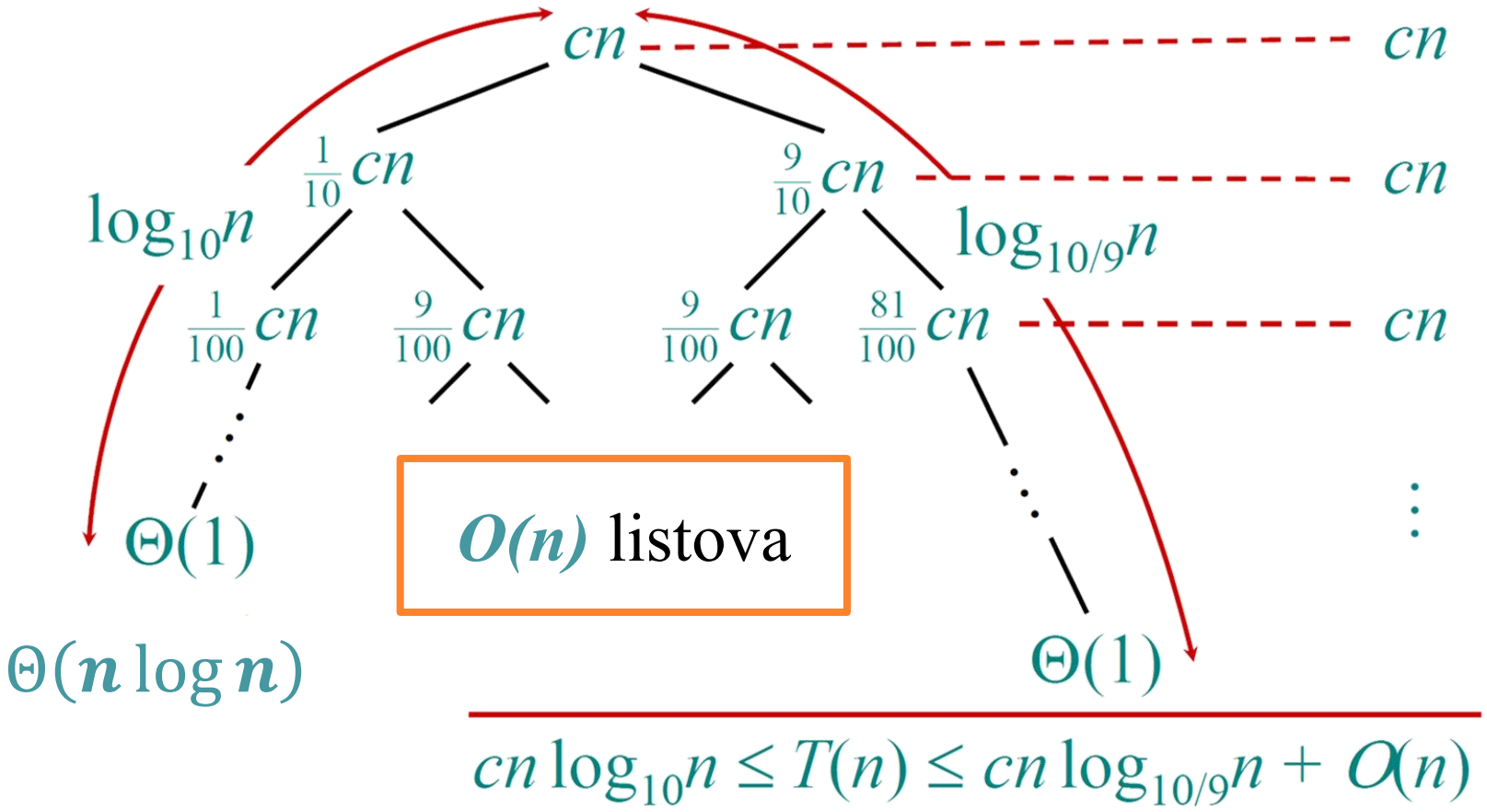
$$T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + \Theta(n)$$



# … QUICKSORT ALGORITAM – ANALIZA …



$$T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + \Theta(n)$$



# … QUICKSORT ALGORITAM – ANALIZA …



- Čak i da postoji situacija u kojoj u jednoj grani nemamo sreće a u drugoj imamo, imamo sledeće rekurentne jednačine

$$S(n) = 2N\left(\frac{n}{2}\right) + \Theta(n)$$

$$N(n) = S(n-1) + \Theta(n)$$

- Kako to rešiti?

$$S(n) = 2\left(S\left(\frac{n}{2} - 1\right) + \Theta\left(\frac{n}{2}\right)\right) + \Theta(n)$$

$$= 2S\left(\frac{n}{2} - 1\right) + \Theta(n)$$

$$= \Theta(n \log n)$$

- Kako da nateramo da sistem da uglavnom budemo srećni?

# … QUICKSORT ALGORITAM – ANALIZA



- Tako što uvedemo randomizaciju (slučajan algoritam)
- Slučajno generišemo poziciju oko koje radimo parcionisanje niza i problema
- Vreme izvršavanja ne zavisi od redosleda elemenat u nizu
- Ne pravimo nikakvu pretpostavku oko distribucije ulaznih vrednosti
- Nikakav poseban ulaz ne dovodi do **worst-case** scenarija
- **Worst-case** scenario zavisi isključivo od generatora slučajnih brojeva

# QUICKSORT ALGORITAM U PRAKSI



- QUICKSORT algoritam je odličan opšti algoritam za sortiranje
- U proseku je duplo brži algoritam od MERGE sortiranja
- Vrlo se lako može prilagoditi konkretnom problemu, tj. vrlo je podložan efikasnom ***code-tuning-u***
- Ponaša se odlično čak i sa keširanjem i u virtualnoj memoriji

# SHELLSORT

# SHELL SORTIRANJE …



- Ime dobila po svom autoru Donald-u Shell-u
- Predstavlja proširenje Insertion sortiranja (tj. bazirana je na Insertion sortiranju)
- Ideja je da se smanji broj pomeranja koji nastaje u Insertion sortiranju kada je potrebno element sa malom vrednošću pomeriti sa mesta pri kraju niza (gde se nalazi) na mesto pri početku niza
- Algoritam koristi Insertion sortiranje, ali na široko razmaknutim elementima
- Zatim se taj razmak (engl. *gap*) umanjuje i sortiraju se manje razmaknuti elementi
- Razmak na osnovu kojeg se vrši sortiranje predstavlja **interval** sortiranja

# … SHELL SORTIRANJE …



- Određivanje intervala u Shell sortiranju nije uopšte trivijalan problem
- Svaki interval koji uključuje u sebe razmak od 1 će na kraju dati tačan rezultat
- Ako interval sadrži premalo razmaka, to će usporiti prolaze
- Ako interval sadrži previše razmaka, doći će do prezasićenja (engl. *overhead*) – previše bespotrebnih prolaza koji ne rade ništa
- Originalni Šelov rad je predložio da se razmaci određuju polovljenjem ukupnog broja elemenata niza ($\boldsymbol{N}$):

  $$\left\lfloor \frac{\boldsymbol{N}}{\boldsymbol{2^k}} \right\rfloor$$

  gde će interval sadržati vrednosti: $\left\lfloor \frac{\boldsymbol{N}}{\boldsymbol{2}} \right\rfloor$, $\left\lfloor \frac{\boldsymbol{N}}{\boldsymbol{4}} \right\rfloor$, $\left\lfloor \frac{\boldsymbol{N}}{\boldsymbol{8}} \right\rfloor$, …, 1
- Pored Šelovog predloga postoje i drugi …

# … SHELL SORTIRANJE …



- Jedan primer računanja intervala je po Knutovoj formuli (D. Knuth):

  $\frac{\mathbf{3}^{k}-\mathbf{1}}{\mathbf{2}}, ali\ da\ nije\ veće\ od\ \left\lceil\frac{N}{3}\right\rceil$

- što se svodi na sledeću formulu:

  $\boldsymbol{h} = \boldsymbol{h}\cdot\mathbf{3}+\mathbf{1}$, gde je inicijalno $\boldsymbol{h} = \mathbf{0}$

- Interval se određuje sledećim algoritmom:

```
ShellSortInterval(A)
1:  while interval < A.length/3
2:      interval = interval * 3 + 1
3:  return interval
```

- Tako da su vrednosti intervala: 1, 4, 13, 40, 121, 364, …

# … SHELL SORTIRANJE …



- Sedgewick je predložio sledeću formulu:

  $4^k + 3 \cdot 2^{k-1} + 1$

  pri čemu niz razmaka započinje sa 1.
- Tako da se dobija sledeći niz razmaka: 1, 8, 23, 77, 281, …

- Pratt je predložio sledeću formulu:

  $2^p \cdot 3^q$
- Tako da se dobija sledeći niz razmaka: 1, 2, 3, 4, 6, 8, 9, 12, …

- Moguće je uzeti i empirijske vrednosti: 1, 4, 10, 23, 57, 132, 301, 701

- Izbor razmaka utiče i na kompleksnost samog algoritma

# … SHELL SORTIRANJE …



- U zavisnosti od izbora intervala zavisi i ukupan broj prolaza kroz niz
- U suštini sortiranje se vrši u malim nizovima kako bi se broj pomeranja kada se dođe do razmaka 1 minimizovao i sveo samo na 1 korak
- Pretpostaviti da imamo niz od 8 elemenata:
  $\langle a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8 \rangle$
- Ako se primeni originalna Šelova metoda za računanje razmaka, $\left\lfloor \frac{N}{2^k} \right\rfloor$, upotrebiće se razmaci: 4, 2, 1
- U **prvom prolazu** će se Inserion sortiranje primeniti nad nizovima:

  $\langle a_1, a_5 \rangle$, $\langle a_2, a_6 \rangle$, $\langle a_3, a_7 \rangle$, $\langle a_4, a_8 \rangle$,

  što će rezultovati nizom:

  $\langle aa_1, aa_2, aa_3, aa_4, aa_5, aa_6, aa_7, aa_8 \rangle$

# … SHELL SORTIRANJE …



- Na kraju 1. prolaza imamo niz: $\langle aa_1, aa_2, aa_3, aa_4, aa_5, aa_6, aa_7, aa_8 \rangle$
- U **drugom prolazu** će se Inserion sortiranje primeniti nad nizovima:

  $\langle aa_1, aa_3, aa_5, aa_7 \rangle$, $\langle aa_2, aa_4, aa_6, aa_8 \rangle$,

  što će rezultovati nizom:

  $\langle aaa_1, aaa_2, aaa_3, aaa_4, aaa_5, aaa_6, aaa_7, aaa_8 \rangle$
- U **trećem prolazu** će se Insertion sortiranje primeniti nad celim nizom:

  $\langle aaa_1, aaa_2, aaa_3, aaa_4, aaa_5, aaa_6, aaa_7, aaa_8 \rangle$
- Ako se primeni empirijski pristup u određivanju intervala, niz razmaka po kome će se računati je: 1, 4, što znači da će i broj prolaza biti manji ali će se više pojedinačnih zamena desiti na potencijalno većem razmaku

# … SHELL SORTIRANJE …



- Ako se primeni empirijski pristup u određivanju intervala, niz razmaka po kome će se računati je: 1, 4
- U **prvom prolazu** će se Inserion sortiranje primeniti nad nizovima:

  $\langle a_1, a_5 \rangle$, $\langle a_2, a_6 \rangle$, $\langle a_3, a_7 \rangle$, $\langle a_4, a_8 \rangle$,

  što će rezultovati nizom:

  $\langle aa_1, aa_2, aa_3, aa_4, aa_5, aa_6, aa_7, aa_8 \rangle$
- U **drugom prolazu** će se Inserion sortiranje primeniti nad celim nizom:

  $\langle aa_1, aa_2, aa_3, aa_4, aa_5, aa_6, aa_7, aa_8 \rangle$
- Broj prolaza će biti manji ali će se više pojedinačnih zamena desiti na potencijalno većem razmaku

# … SHELL SORTIRANJE …



- Shell primer nad istim nizom kao u primeru za Insertion sortiranje:

  $[5\ 2\ 4\ 6\ 1\ 3]$
- Primenjujemo originalni Šelov pristup za računanje razmaka
- Kako niz ima 6 elemenata, dobijamo 2 razmaka: 1 i 3
- U **prvom prolazu** sortiramo nizove:

  $[5\ 6]$, $[2\ 1]$ i $[4\ 3]$
- Što rezultuje sledećim nizovima:

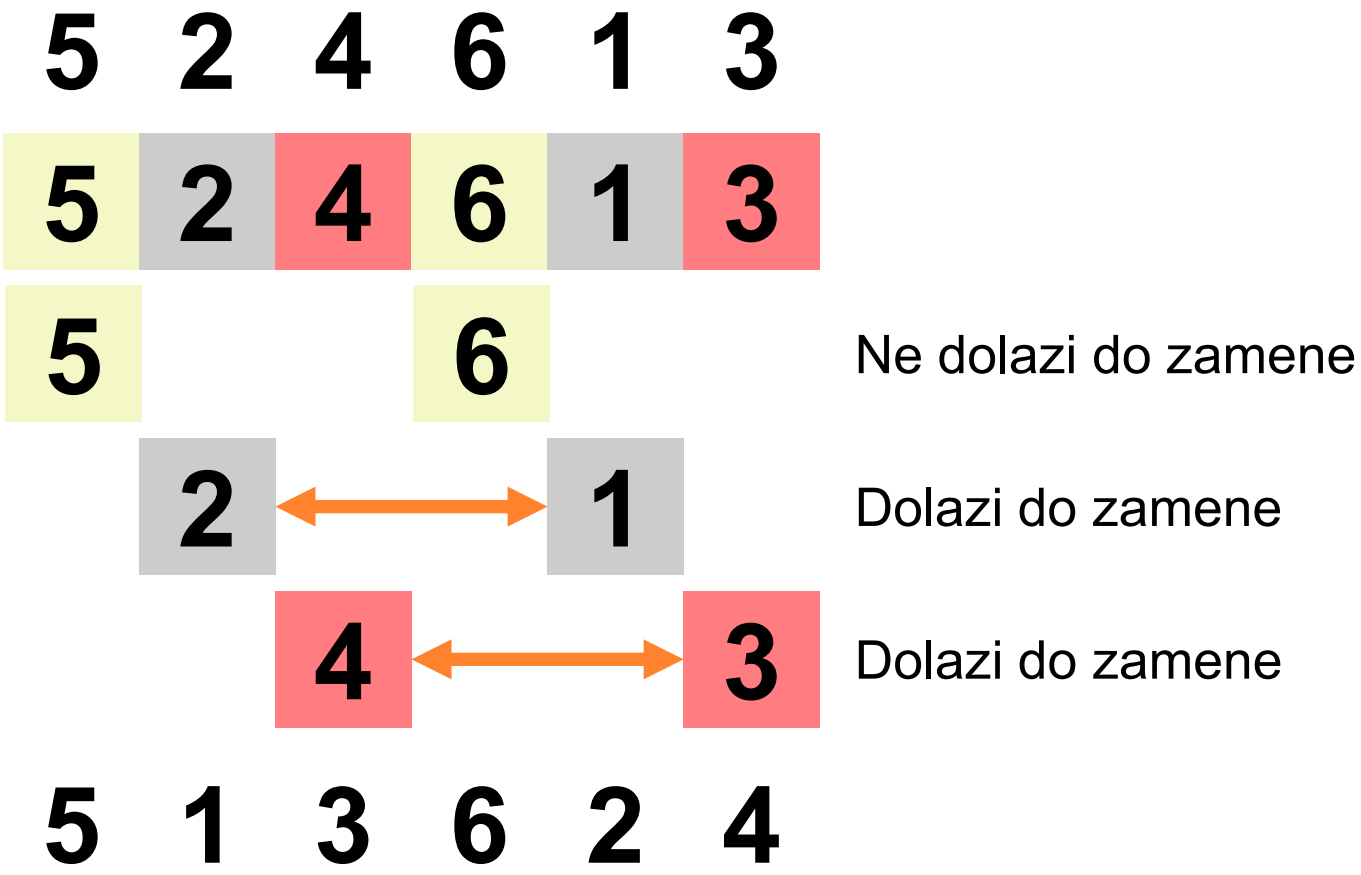
  $[5\ 6]$, $[1\ 2]$ i $[3\ 4]$
- Odnosno nizom:

  $[5\ 1\ 3\ 6\ 2\ 4]$
- Vizuelno čitav proces ima sledeći izgled…

# … SHELL SORTIRANJE …



- Vizuelno čitav proces ima sledeći izgled:



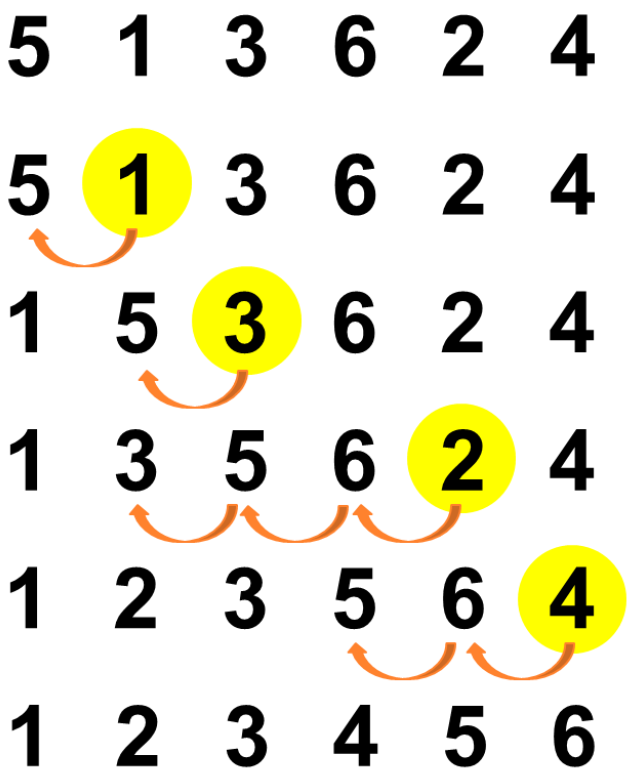
# … SHELL SORTIRANJE …



- U drugom prolazu radi se klasično Insertion sortiranja (jer je korak 1) nad nizom koji je rezultat prvog prolaza:

  [5 1 3 6 2 4]

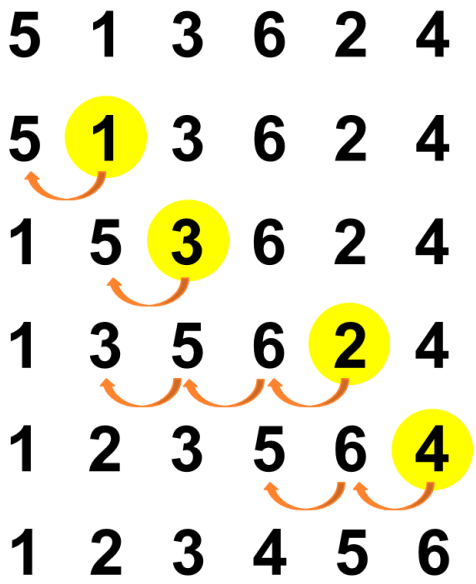
- Vizuelno čitav proces ima sledeći izgled…

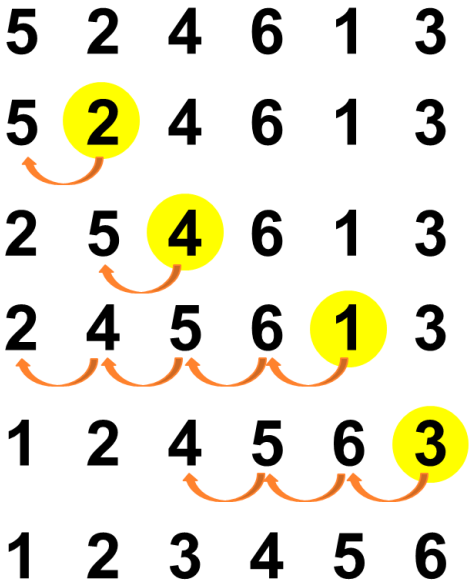


# … SHELL SORTIRANJE …



- Ako se pogleda originalno Insertion sortiranje nad datim ulazom




- I nije se dobilo puno, jer je broj pomeranja ostao isti (ako se kod Shell sortiranja uzmu u obzir i prva dva pomeranja u prvom prolazu)
- Razlog za to?

# … SHELL SORTIRANJE …



- Unapređenje Shell algoritma za sortiranje se ne vidi na malim ulazima
- Osmišljeno je da pomogne kod velikih ulaza
- Za niz iz primera Knutova formula bi dovela do intervala od samo jednog razmaka, 1 (kao i kod mnogih drugih načina računanja razmaka)
- Ovaj algoritam je najefikasniji za ulaze srednje veličine
- Kompleksnost može ići od $\boldsymbol{O(n)}$ u najboljem slučaju do $\Theta(\boldsymbol{n^2})$ u najgorem slučaju
- Kompleksnost sa Knutovom formulom za određivanje razmaka je $\Theta(\boldsymbol{n^{\frac{3}{2}}})$
- Kompleksnost sa Sedvikovom formulom za određivanje razmaka je $\Theta(\boldsymbol{n^{\frac{4}{3}}})$

# … SHELL SORTIRANJE …



- Shell algoritma ima sledeći oblik:

```
SHELLSORT(A)
1:   interval = CREATESHELLSORTINTERVAL(A)
2:   while interval > 0
3:       for i = interval to A.length − 1
4:           key = A[i + 1]
5:           j = i + 1
6:               while j > interval and A[j − interval] > key
7:                   A[j] = A[j − interval]
8:                   j = j − interval
9:               A[j] = key
10:      interval = REDUCESHELLSORTINTERVAL(interval)
```

# … SHELL SORTIRANJE …



- Osnova Shell algoritma ostaje ista bez obzira kako se računa interval zbog čega je u primeru to predstavljeno kao poziv odgovarajućih podalgoritma
- Kako bi ti podalgoritmi izgledali za slučaj kada se interval računa prema Knutovoj formuli prikazano je sledećim podalgoritmima:

```
SHELLSORT(A)
1:  interval = CREATESHELLSORTINTERVAL(A)
2:  while interval > 0
3:      for i = interval to A.length − 1
4:          key = A[i + 1]
5:          j = i + 1
6:              while j > interval and A[j − interval] > key
7:                  A[j] = A[j − interval]
8:                  j = j − interval
9:              A[j] = key
10:     interval = REDUCESHELLSORTINTERVAL(interval)
```

```
CREATESHELLSORTINTERVAL(A)
1:  while interval < A.length/3
2:      interval = interval * 3 + 1
3:  return interval
```

```
REDUCESHELLSORTINTERVAL(interval)
1:  return (interval − 1)/3
```

# … SHELL SORTIRANJE



- Shell algoritam kreće od najvećeg razmaka u datom intervalu i postepeno ga smanjuje sve dok njegova vrednost ne padne ispod nule (poslednji prolaz će biti za razmak 1)
- Za svaki razmak se prolazi kroz ceo niz u pravcu kraja niza (desno od datog elementa)
- U svakom trenutku se trenutna vrednost (aktivnog) elementa niza poredi sa vrednostima elemenata niza koji se nalaze levo od aktivnog elementa i koji su udaljeni od njega $\boldsymbol{n \cdot razmak}$ ($\boldsymbol{interval}$ u algoritmu)
- Vrednost trenutno aktivnog elementa niza se pomera ulevo sve dok je ta vrednost manja od svih ostalih elemenata niza

```
SHELLSORT(A)
1:  interval = CREATESHELLSORTINTERVAL(A)
2:  while interval > 0
3:      for i = interval to A.length − 1
4:          key = A[i + 1]
5:          j = i + 1
6:              while j > interval and A[j − interval] > key
7:                  A[j] = A[j − interval]
8:                  j = j − interval
9:              A[j] = key
10:     interval = REDUCESHELLSORTINTERVAL(interval)
```

```
CREATESHELLSORTINTERVAL(A)
1:  while interval < A.length/3
2:      interval = interval * 3 + 1
3:  return interval
```

```
REDUCESHELLSORTINTERVAL(interval)
1:  return (interval − 1)/3
```

# TIM SORT

# TIM SORT …



- Hibridni algoritam
- Stabilan algoritam (održava pozicije istih vrednosti)
- Dizajniran da radi dobro za realne probleme koji se često sreću
- Implementirao ga je Tim Peters 2001. godine za potrebe Python programskog jezika na osnovu rada Peter McIlroy-a „Optimistic Sorting and Information Complexity“ iz 1993. godine
- Koristi se kao primarni algoritam za sortiranje u Pajtonu (od verzije 2.3 do verzije 3.10), ali i za sortiranje neprimitivnih tipova u Java SE 7, na Androidu, GNU Octave, Swift-u i Rust-u (inspiracija za algoritam)
- Algoritam je adaptivne prirode
- Kombinuje Insertion i Merge sortiranje
- Kompleksnost mu je u najgorem slučaju $\boldsymbol{O}(\boldsymbol{n}\log\boldsymbol{n})$, dok je u najboljem $\boldsymbol{O}(\boldsymbol{n})$

# … TIM SORT …



- Osnova Tim sortiranja je Merge sortiranje koje je samo po sebi već dovoljno optimizovano za sortiranje poređenjem
- Kako je osnova Merge sortiranja procedura spajanja dva već sortirana podniza, postavlja se pitanje kako se Merge sortiranje može unaprediti u realnim situacijama?
  - Ubrzavanjem samog spajanja?
  - Smanjivanjem ukupnog broj spajanja?
  - Izbegavanjem Merge sortiranja u situacijama kada postoji bolji algoritam za sortiranje?
- U suštini, sve tri stvari
  - Zbog rekurzivne implementacije, lako se prebaciti na drugi algoritam za sortiranje (kada su nizovi male dužine ima smisla izbeći Merge sortiranje, npr. Insertion sortiranjem)

# … TIM SORT …



- Timsort je dizajniran da koristi prednosti parcijalnog redosleda koji već postoji u većini podataka iz realnog sveta
- Algoritam prvo analizira podatke (iterativno se kreće kroz ulazni niz), pri čemu traži prirodno sortirane prolaze (*natural runs*), podnizove od najmanje dva elementa

[2, 3, 4, 17, 94]

- Prolazi su ili neopadajući (svaki element je jednak ili veći od prethodnog) ili striktno opadajući (svaki element je manji od prethodnog)

[**2, 3, 4**, 17, 94]

- Opadajući prolaz mora biti strogo opadajući, kako bi se kasnije elem. obrnuli jednostavnim zamenama sa oba kraja konvergirajući u sredini (metod je stabilan ako se elementi prezentuju u striktno opadajućem redosledu)

[3, 2, 1, 9, 17, 34]

[**1, 2, 3**, 9, 17, 34]

- Posle obavljanja ovakvog prolaza u datom nizu, Timsort ga prihvata i procesuira, i onda traži sledeći prolaz

# … TIM SORT …



- Prirodni prolazi su sasvim uobičajeni u realnim podacima, ali mogu biti različitih dužina
- Koja će se tehnika sortiranja primeniti zavisi od veličine prolaza (za prolaze malih dužina koristi se insertion sortiranje), zato je adaptivan
- Veličina prolaza se poredi sa minimumom veličine prolaza (miniprolaz) koja u suštini zavisi od dužine samog niza
- Miniprolaz se bira tako da broj poziva funkcija ograniči na neki razuman broj
- Za niz koji ima manje od 64 elementa, miniprolaz je veličine niza, redukujući timsort na sortiranje umetanjem
- Za veće nizove, miniprolaz se bira u rangu između 32 i 64 elementa, tako da je veličina niza, podeljena miniprolazom, jednaka ili malo manja od stepena dvojke

# … TIM SORT …



- Veličina niza u stepenu dvojke se bira zato što je merge algoritam najefikasniji kada su podnizovi veličine stepena dvojke ili malo manji, dok je manje efikasan za veličine koje su malo veće od stepena dvojke
- Veličina podniza se određuje tako što se uzima 6 najznačajnijih bitova broja koji predstavlja veličinu niza, dodaje jedan ako je bilo koji od preostalih bitova postavljen, i koristi taj rezultat kao miniprolaz
- Na ovaj način algoritam radi i kad je veličina niz  64 ili manja (u kom slučaju se za miniprolaz uzima dužinu niza)

# … TIM SORT …



- Kod slučajnih nizova (sa prirodnim podacima), prolazi će verovatno imati manje elemenata od miniprolaza
- Određen broj uspešnih elemenata se bira, i **sortiranje umetanjem povećava veličina prolaza do miniprolaza**, tako da većina prolaza u slučajnom nizu, ili postanu, ili su u veličini miniprolaza
- Rezultat su efikasna, balansirana spajanja
- To dovodi do razumnog broja poziva funkcija pri implementaciji algoritma
- Kada su dužine prolaza optimizovane, prolazi se spajaju
- Kada je prolaz pronađen, algoritam gura njegovu baznu adresu i dužinu na stek

# … TIM SORT …



- Paralelno sa potragom za prolazom, algoritam spaja susedne prolaze primenom Merge sortiranja
- Prilikom spajanja algoritam je podešen da se vodi računa o stabilnosti i balansiranosti procesa spajanja
- Da bi se održala stabilnost, algoritam spaja samo uzastopne prolaze
- Gledaju se tri najviša prolaza na steku koji su nesortirani
- Ako, na primer, $\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{Z}$, predstavljaju dužine tri krajnja prolaza na steku, algoritam spaja prolaze tako da sledeća dva pravila budu zadovoljena: $|\boldsymbol{Z}| > |\boldsymbol{Y}| + |\boldsymbol{X}|$ i $|\boldsymbol{Y}| > |\boldsymbol{X}|$
- Ako pravila nisu zadovoljena, $\boldsymbol{Y}$ se spaja sa manjim od $\boldsymbol{X}$ i $\boldsymbol{Z}$, i ispunjenost pravila se ponovo testira
- Spajanje se nastavlja dok oba pravila nisu ispunjena
- Tada algoritam određuje sledeći prolaz

# … TIM SORT …



- Prethodna pravila su napravljena sa ciljem održavanja veličine prolaza što je moguće bliže jedna drugoj da bi se balansirala spajanja
- Samo mali broj prolaza se pamti, pošto je stek specifične veličine
- Algoritam koristi sveže pojave prolaza za spajanje u keš memoriji
- Kompromis se postiže tako što se odlaže spajanje, a koristi sveže pojavljivanje u keš memoriji
- Sam algoritam odlučivanja kada da se desi spajanje je relativno jednostavan
- Kada se dođe do kraja podataka, algoritam spaja uzastopna dva prolaza na vrhu steka sve dok ne ostane samo jedan prolaz, koji je zapravo sortirani niz

# … TIM SORT …



- Spajanje se ne realizuje bukvalno u mestu (in-place), već skoro (almost in-place), tj. pomoću privremene memorije (originalno spajanje u mestu ima prevelik *overhead*)
- Veličina privremene memorija odgovara veličini manjeg od dva prolaza
- Algoritam kopira manji od dva prolaza u privremenu memoriju i koristi originalnu memoriju i memoriju drugog prolaza da smesti sortirani rezultat
- Jednostavni algoritam sortiranja radi sa leva na desno, ili sa desna na levo, u zavisnosti koji je prolaz manji, na privremenoj i originalnoj memoriji većeg prolaza
- Krajnji sortirani prolaz se smešta u originalnu memoriju dva inicijalna prolaza
- Timsort traži odgovarajuće pozicije za početni element jednog niza u drugom koristeći adaptaciju binarne pretrage

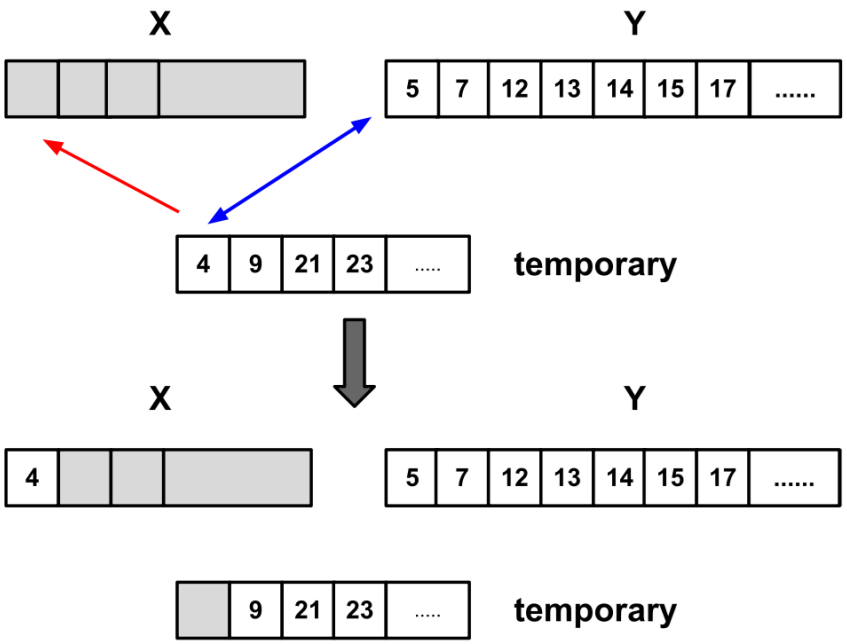
# … TIM SORT …

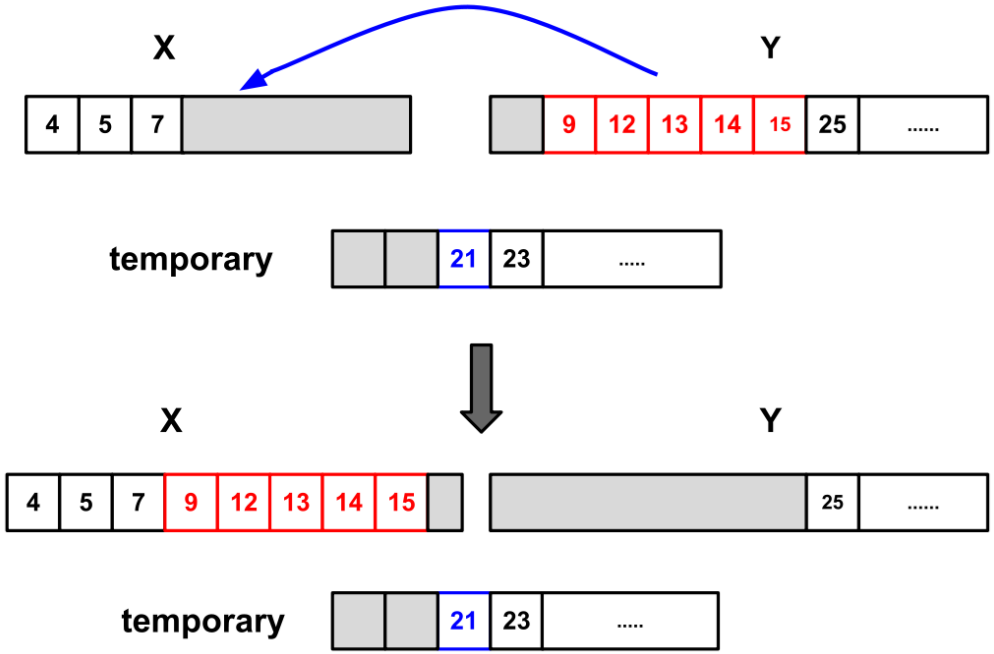


- Na primer, dva prolaza $A$ i $B$ trebaju biti spojeni, a $A$ je manji prolaz
- U ovom slučaju binarna pretraga gleda u $A$ da nađe prvi element $a'$ veći od prvog elementa u $B$
- $A$ i $B$ već sortirani individualno
- Kada je $a'$ je pronađen, algoritam može ignorisati elemente pre te pozicije dok ubacuje element u $B$
- Slično, algoritam takođe traži najmanji element $b'$ u $B$ koji je veći od poslednjeg (najvećeg) elementa u $A$
- Elementi posle $b'$ mogu takođe biti ignorisani pri spajanju
- Ova preliminarna pretraga nije efikasna za visok stepen nasumičnosti podataka, ali je efikasna za druge situacija, pa je prema tome, uključena

# … TIM SORT …




X
Y
5 7 12 13 14 15 17 .....
4 9 21 23 .....
temporary
X
Y
4
5 7 12 13 14 15 17 .....
9 21 23 .....
temporary


X
Y
4 5 7
9 12 13 14 15 25 .....
temporary
21 23 .....
X
Y
4 5 7 9 12 13 14 15
25 .....
temporary
21 23 .....

# … TIM SORT …



- Prilikom individualnog spajanja broji se koliko uzastopnih elemenata je odabrano iz istog prolaza
- Kada ova vrednost dostigne određeni prag, MIN_GALOP, spajanje se prebacuje u „galopirajući mod“ kako bi se iskoristilo na prolazu unutar jednog podniza
- U ovom modu se koristi adaptacija binarne pretrage, da se utvrdi gde prvi element manjeg niza treba biti smešten u većem nizu (i suprotno)
- U galopirajućem modu, algoritam traži prvi element jednog niza u drugom
- Ovo se radi poređenjem prvog elementa (inicijalnog elementa) sa $\mathbf{2}^{k} - \mathbf{1}$-ti elementom drugog prolaza (prvim, trećim i tako dalje), tako da se dobije rang elemenata između kojih će inicijalni element biti
- Ovo sužava rang binarne pretrage, prema tome povećava efikasnost

# … TIM SORT …



- Galopiranje se pokazalo efikasnijim u svim slučajevima osim u slučajevima specifično dugih prolaza
- Nasumični podaci obično imaju kraće prolaze
- U slučajevima kada se galopiranje pokazuje manje efikasno od binarne pretrage, izlazi se iz galopirajućeg moda
- Galopiranje je korisno jedino kada početni element prolaza nije među prvih 7 elemenata drugog prolaza – ovo implicira MIN_GALOP od 7
- Da bi se izbegla zadržavanja galopirajućeg moda, merge funkcija podešava vrednost min_galopa
- Ako je element iz niza koji trenutno vraća elemente, min_galop se smanjuje za jedan
- Inače, vrednost se povećava za jedan, pa smanjuje verovatnoću prelaska u galopirajući mod
- Kada se ovo uradi, u slučaju nasumičnih podataka, vrednost min_galopa se povaćava toliko da se nikad ne ulazi u galopirajući mod

# … TIM SORT …



- Kompleksnost:
  - Algoritam sortiranja baziran na poređenju ne može biti brži od $\boldsymbol{O(n \log n)}$
  - Na realnim podacima, timsort često zahteva mnogo manje od $\boldsymbol{O(n \log n)}$, ide do $\boldsymbol{O(n)}$, jer koristi činjenicu da su podskupovi skupa već sortirani
  - U slučaju da su podaci potpuno obrnuti u odnosu na smer sortiranja, timsort se približava teoretskom limitu $\Theta(\boldsymbol{n \log n})$
- Tim sort je dobar jer je:
  - Adaptivan
  - Vodi računa o već sortiranim delovima (neće deliti već sortirane delove niza)
- Ograničenje, može biti memorijski zahtevan

# … TIM SORT …



- Kako izgleda sam algoritam:

```
TIMSORT(A)
01: if A.length < 2 then
02:     return
03: minrun = COMPUTE-MINRUN(A.length)
04: runs = EMPTY-STACK()
05: i = 1
06: while i ≤ A.length
07:     run_len = COUNT-RUN-AND-MAKE-ASCENDING(A, i)
08:     if run_len < minrun then
09:         forced_len = MIN(minrun, A.length − i)
10:         INSERTION-SORT(A, i, i + forced_len)
11:         run_len = forced_len
12:     PUSH(runs, (i, run_len))
13:     MERGE-COLLLAPSE(A, runs)
14:     i = i + run_len
15: MERGE-FORCE-COLLAPSE(A, runs)
```

# … TIM SORT …



- prvo se odredi kolika će biti najmanja veličina prolaza:

```
COMPUTE-MINRUN(n)
01: r = 0
02: while n ≥ 64
03:     r = r BITWISE-OR (n BITWISE-AND 1)
04:     n = FLOOR(n/2)
05: return n + r
```

# … TIM SORT …



- Pronalazak sortiranih podnizova:

COUNT-RUN-AND-MAKE-ASCENDING($A$, $start$)

```
01: if start = A.length then
02:     return 1
03: i = start + 1
04: if A[i] < A[start] then
05:     while i ≤ A.length AND A[i] < A[i − 1]
06:         i = i + 1
07:     REVERSE(A, start, i)
08: else
09:     while i ≤ A.length AND A[i] ≥ A[i − 1]
10: i = i + 1
11: return i − start
```

# … TIM SORT …



```
MERGE-COLLLAPSE(A, runs)
01: while runs.size > 1
02:     n = runs.size
03:     if n ≥ 3 then
04:         X = runs[n − 2]
05:         Y = runs[n − 1]
06:         Z = runs[n]
07:         if X.length ≤ Y.length + Z.length then
08:             if X.length < Z.length then
09:                 MERGE-AT(A, runs, n − 2)
10:             else
11:                 MERGE-AT(A, runs, n − 1)
12:         else if Y.length < Z.length then
13:             MERGE-AT(A, runs, n − 1)
14:         else break
15:     else
16:         Y = runs[n − 1]
17:         Z = runs[n]
18:         if Y.length < Z.length then
19:             MERGE-AT(A, runs, n − 1)
20:         else break
```

- Svaki put kada se doda novi prolaz na stek proverava se da li je održana dobra balansiranost steka

```
MERGE-AT(A, runs, i)
01: rs1 = runs[i].start
02: rl1 = runs[i].length
03: rs2 = runs[i + 1].start
04: rl2 = runs[i + 1].length
05: MERGE(A, rs1, rl1, rs2, rl2)
06: runs[i] = (rs1, rl1 + rl2)
07: REMOVE(runs[i + 1])
```

# … TIM SORT



- Na kraju se spoje svi preostali prolazi:

```
MERGE-FORCE-COLLAPSE(A, runs)
01: while runs.size > 1
02:     n = runs.size
03:     if n ≥ 3 AND runs[n − 2].length < runs[n].length then
04:         MERGE-AT(A, runs, n − 2)
05:     else
06:         MERGE-AT(A, runs, n − 1)
```

```
TIMSORT(A)
01: if A.length < 2 then
02:     return
03: minrun = COMPUTE-MINRUN(A.length)
04: runs = EMPTY-STACK()
05: i = 1
06: while i ≤ A.length
07:     run_len = COUNT-RUN-AND-MAKE-ASCENDING(A, i)
08:     if run_len < minrun then
09:         forced_len = MIN(minrun, A.length − i)
10:         INSERTION-SORT(A, i, i + forced_len)
11:         run_len = forced_len
12:     PUSH(runs, (i, run_len))
13:     MERGE-COLLLAPSE(A, runs)
14:     i = i + run_len
15: MERGE-FORCE-COLLAPSE(A, runs)
```

# SORTIRANJE U LINEARNOM VREMENU

# BRZINA SORTIRANJA NA BAZI POREĐENJA …



- Svi algoritmi za sortiranje do sada su funkcionisali na bazi tehnike poređenja vrednosti
  - Koristili su poređenje da odrede relativno uređenje elemenata
  - Insertion sortiranje, marge sortiranje, heapsort sortiranje, quicksort sortiranje, shell sortiranje, tim sortiranje
- Ovakvi algoritmi se zovu algoritmi za sortiranje na bazi poređenja (*comparison sort*)
- Najbolje vreme koje može da se očekuje za **worst-case** situaciju kod sortiranja na bazi poređenja (*comparison sort*) je $\Theta(\boldsymbol{n} \log \boldsymbol{n})$
- Postavlja se pitanje, da li može bolje?

# … BRZINA SORTIRANJA NA BAZI POREĐENJA …



- Kod algoritama za sortiranje na bazi poređenja niz $\langle a_1, a_2, \ldots, a_n \rangle$ od n brojeva, odluka o preraspoređivanje elemenata $a_i$ i $a_j$ u nizu se donosi na osnovu sledećih kombinacija $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, $a_i > a_j$
- Ako pretpostavimo da se svi elementi niza razlikuju, gubimo potrebu za poređenjem tipa $a_i = a_j$
- Takođe, sva preostala poređenja su međusobno ekvivalentna, i dovoljno je uraditi samo jedno od njih, npr. $a_i \leq a_j$, da bi se saznao odnos između dva elementa
- Sortiranje na bazi poređenja može se apstraktno predstaviti u obliku stabla odlučivanja
- Stablo odlučivanja je kompletno binarno stablo i simulira kakva će biti performansa algoritma za sortiranje kada se donese odluka na osnovu poređenja

# … BRZINA SORTIRANJA NA BAZI POREĐENJA …



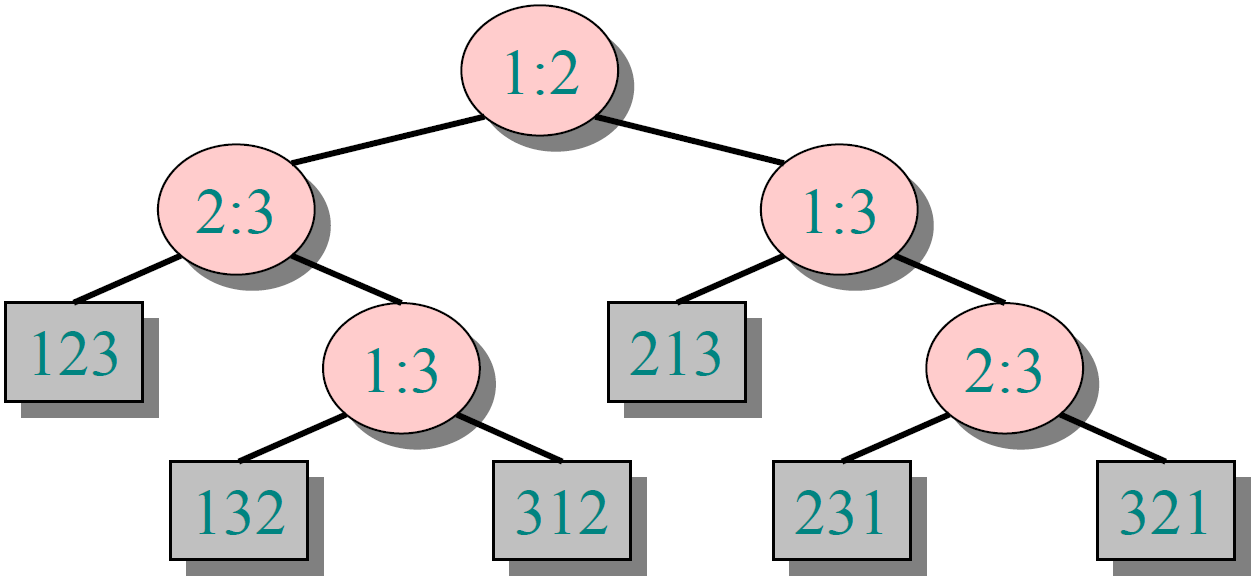
- Svaki čvor u stablu je označen sa ***i:j*** za neku ***i*** i ***j*** vrednost iz opsega $\mathbf{1} \leq \boldsymbol{i}, \boldsymbol{j} \leq \boldsymbol{n}$, gde je ***n*** broj elemenata niza
- Leva strana stabla predstavlja dodatna poređenja ako je $\boldsymbol{a_i} \leq \boldsymbol{a_j}$
- Desna strana stabla predstavlja dodatna poređenja ako je $\boldsymbol{a_i} \geq \boldsymbol{a_j}$
- Izvršavanje algoritma se posle svodi na obilazak stabla do listova
- Svaki list predstavlja neku od permutacija inicijalnog niza $\langle \boldsymbol{\pi}(\mathbf{1}), \boldsymbol{\pi}(\mathbf{2}), \ldots, \boldsymbol{\pi}(\boldsymbol{n}) \rangle$
- Kada se stigne do lista stabla, dobija se neka permutacija niza i uspostavlja se uređenje $\boldsymbol{a_{\pi(1)}} \leq \boldsymbol{a_{\pi(2)}} \leq \cdots \leq \boldsymbol{a_{\pi(n)}}$

# … BRZINA SORTIRANJA NA BAZI POREĐENJA …



- Sortirati niz

$\langle a_1, a_2, \ldots, a_n \rangle$



# … BRZINA SORTIRANJA NA BAZI POREĐENJA …

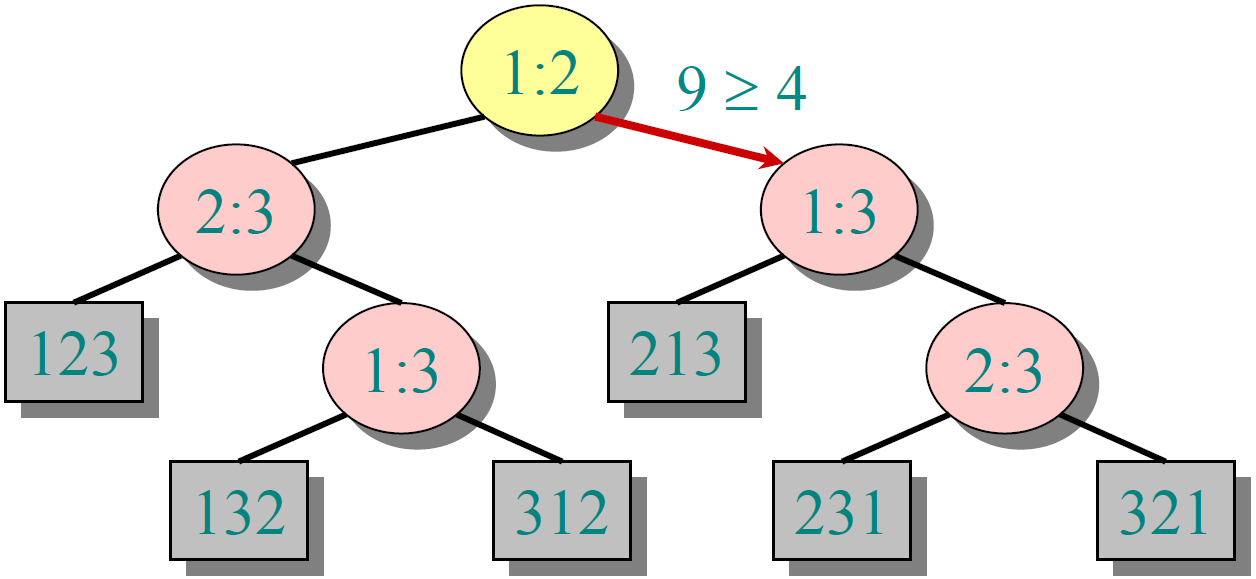


- Sortirati niz

  $\langle a_1, a_2, \ldots, a_n \rangle$

- Primer

  $\langle 9, 4, 6 \rangle$



# … BRZINA SORTIRANJA NA BAZI POREĐENJA …

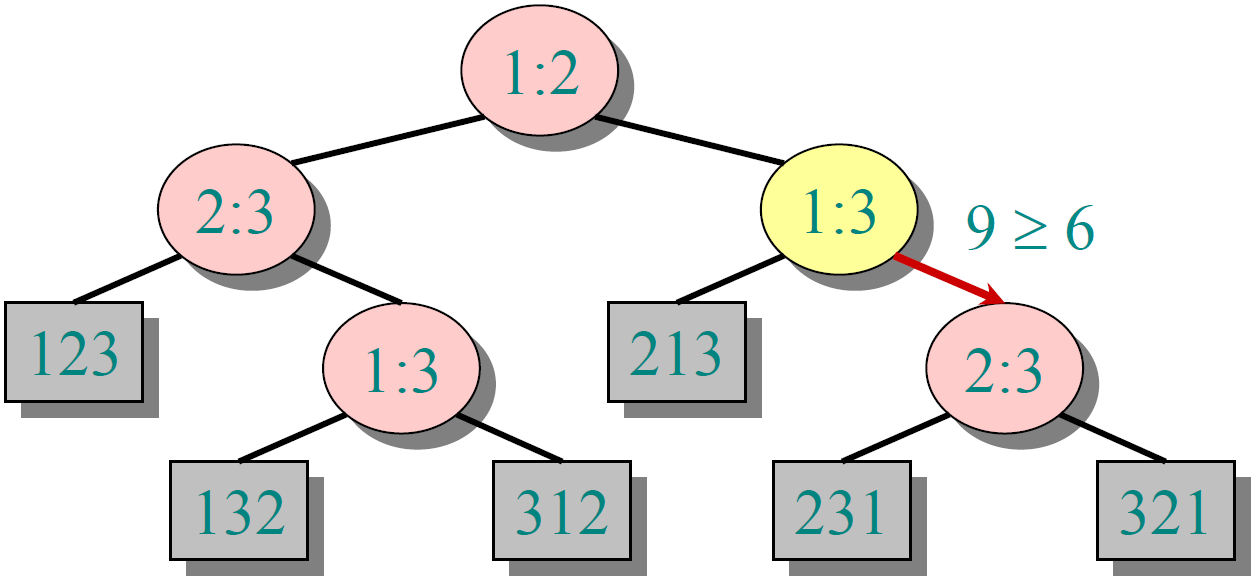


- Sortirati niz

  $\langle a_1, a_2, \ldots, a_n \rangle$

- Primer

  $\langle 9, 4, 6 \rangle$



# … BRZINA SORTIRANJA NA BAZI POREĐENJA …

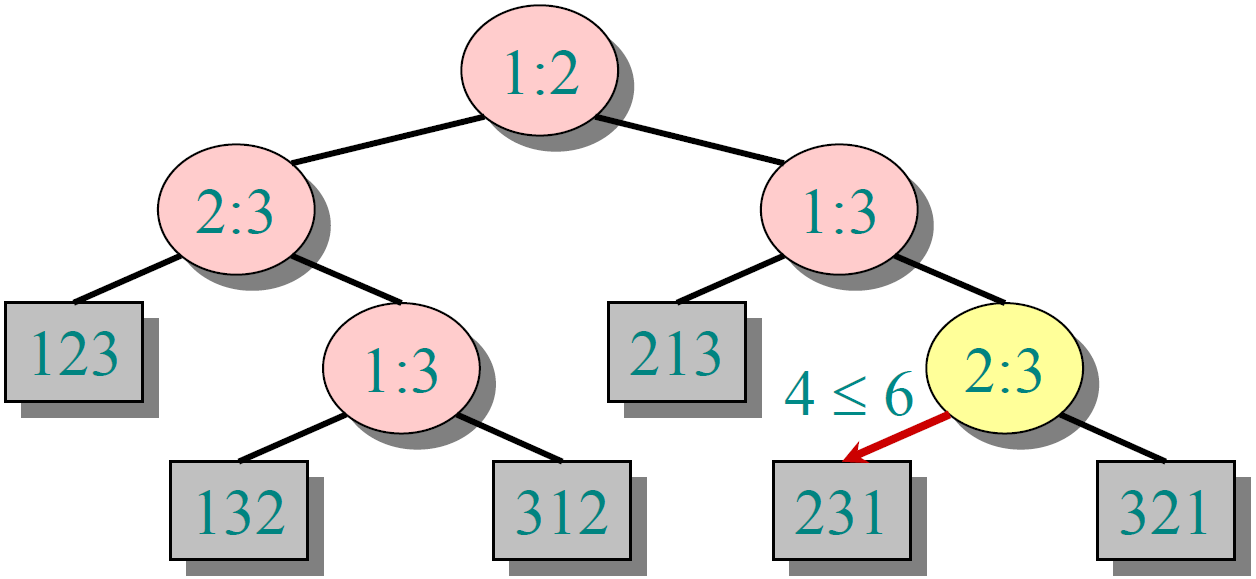


- Sortirati niz

  $\langle a_1, a_2, \ldots, a_n \rangle$

- Primer

  $\langle 9, 4, 6 \rangle$



# … BRZINA SORTIRANJA NA BAZI POREĐENJA …

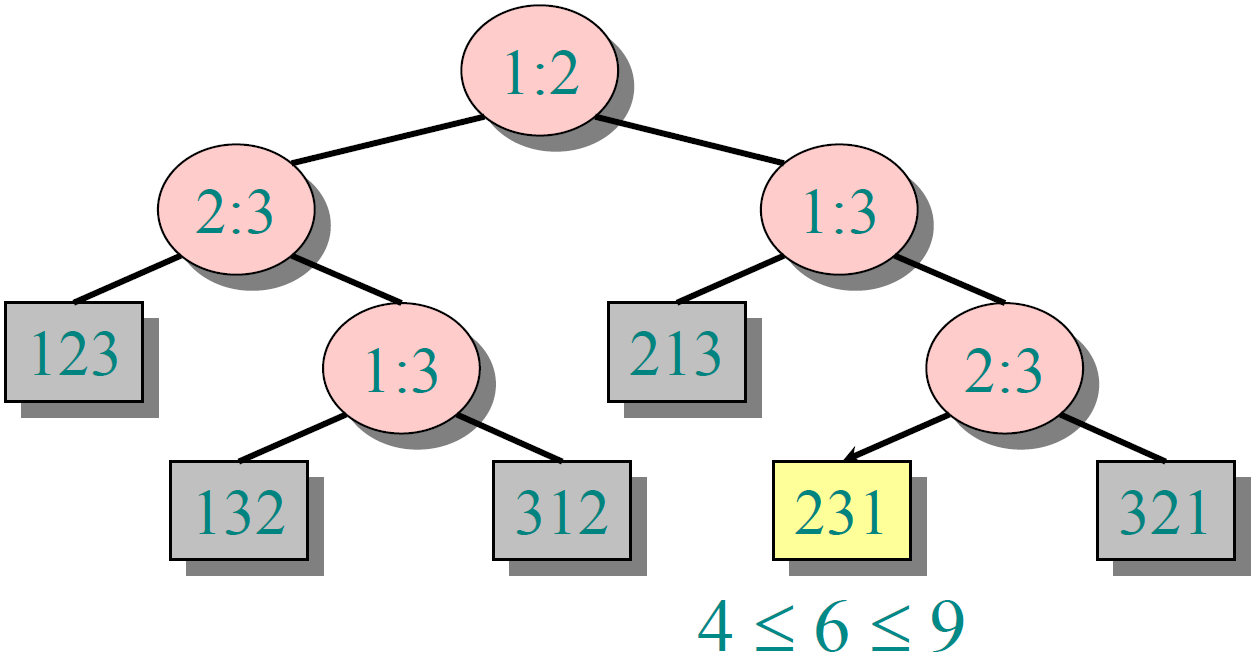


- Sortirati niz

  $\langle a_1, a_2, \ldots, a_n \rangle$

- Primer

  $\langle 9, 4, 6 \rangle$



# … BRZINA SORTIRANJA NA BAZI POREĐENJA …



- Stablo odlučivanja može da modeluje izvršavanje bilo kog algoritma za sortiranje na bazi poređenja
  - Jedno stablo za svaki ulaz veličine ***n***
  - Posmatramo algoritam kako se deli svaki put kada poredimo dva elementa
  - Stablo sadrži u sebi sva moguća poređenja koja mogu da se jave tokom poređenja
  - Vreme izvršavanja algoritma jednaka je dužini putanje duž stabla
  - **worste-case** je jednak visini stabla

# … BRZINA SORTIRANJA NA BAZI POREĐENJA …



- Teorema
  Bilo koji algoritam za sortiranje na bazi poređenja zahteva $\Omega(\boldsymbol{n}\log\boldsymbol{n})$ u **worste-case** scenariju
- Dokaz
  - Stablo mora da koristi ***n!*** listova, pošto postoji ***n!*** mogućih permutacija
  - Binarno stablo veličine $\boldsymbol{h}$ ima manje od $\leq \boldsymbol{2^h}$ listova
  - Dakle $\boldsymbol{n}! \leq \boldsymbol{2^h}$

$$\begin{aligned}\boldsymbol{h} &\geq \log(\boldsymbol{n}!)\\ &\geq \log\left(\left(\frac{\boldsymbol{n}}{\boldsymbol{e}}\right)^{\boldsymbol{n}}\right)\\ &= \boldsymbol{n}\cdot\log\boldsymbol{n} - \boldsymbol{n}\cdot\log\boldsymbol{e}\\ &= \Omega(\boldsymbol{n}\log\boldsymbol{n})\end{aligned}$$

Stirlingova aproksimacija:

$$\boldsymbol{n}! = \sqrt{2\pi\boldsymbol{n}}\left(\frac{\boldsymbol{n}}{\boldsymbol{e}}\right)^{\boldsymbol{n}}\left(1+\Theta\left(\frac{1}{\boldsymbol{n}}\right)\right)$$

# … BRZINA SORTIRANJA NA BAZI POREĐENJA



- Posledica
  Heapsort i Quicksort su asimptotski optimalni algoritmi za sortiranje na bazi poređenja

- Dokaz
  - **worste-case** vreme za oba algoritma je $\boldsymbol{O(n \log n)}$
  - što odgovara najnižoj donjoj granici od $\Omega(\boldsymbol{n \log n})$ za **worste-case** vreme

# COUNTING SORT

# COUNTING SORT …



- Nema poređenja među elementima
- Podrazumeva da je ulaz niz celih brojeva u opsegu ***0*** do ***k***, gde je ***k*** neki ceo broj
- Counting algoritam za sortiranje za svaki element ***x*** određuje koliko ima elemenata manjih od ***x***
- I onda smešta ***x*** tačno na njegovu odgovarajuću poziciju (indeks)
  - Ako ima 10 elemenata manjih od ***x***, onda njega treba smestiti na 11 poziciju
  - Algoritam se mora modifikovati ako postoje iste vrednosti u nizu
- Pretpostavka:
  - Ulaz je niz $A[1 \ldots n]$, gde je $n = A.length$
  - Potrebna su još dva niza: $B[1 \ldots n]$ koji sadrži sortirani izlaz i
  - Niz $C[0 \ldots k]$ koji predstavlja privremeno skladište podataka

# … COUNTING SORT …



```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 2 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

# … COUNTING SORT …



- Prvo se inicijalizuje niz C sve na 0
- Zatim se pristupa svakom elementu niza i ako je njegova vrednost ***i***, inkrementiramo vrednost ***C[ i ]***
- Posle 5.og koraka sadrži koliko elemenata ***i*** ima u niz ***A[ i ]***

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 2 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

- Na linijama 7-8 se određuje tačno koliko ima elemenata manjih ili jednakih ***i***, za svako ***i = 0, 1, …, k*** (sumiranjem svih vrednosti do elementa ***i*** )
- Linije 10-12 postavljaju svaki element na njegovo odgovarajuće mesto

# … COUNTING SORT …



- Linije 10-12 postavljaju svaki element na njegovo odgovarajuće mesto

- Ako su svih ***n*** elemenata različitih, onda je ***C[ A[ j ] ]*** odgovarajuća i finalna pozicija elementa ***A[ j ]***

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 2 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

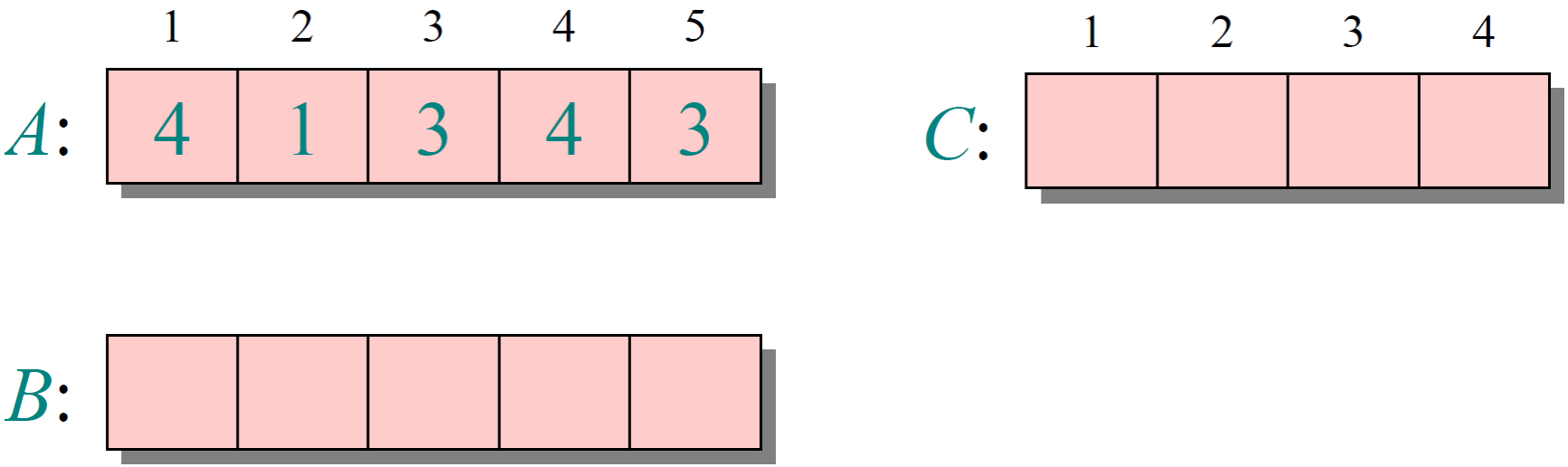
- Pošto može biti i duplikata, vrednost ***C[ A[ j ] ]*** se dekrementira svaki put kada u niz ***B*** smestimo element ***A[ j ]***

- Smanjivanje ***C[ A[ j ] ]*** dovodi do toga da sledeće element u nizu ***A*** koji ima istu vrednost kao ***A[ j ]***, ako postoji, ide na poziciju u izlaznom nizu neposredno ispred ***A[ j ]***

# … COUNTING SORT …



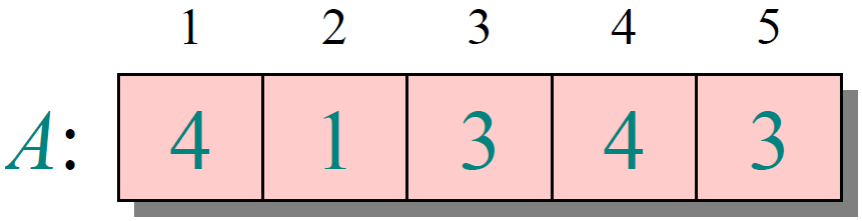
- Primer

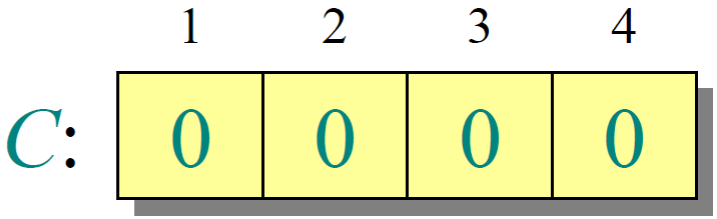


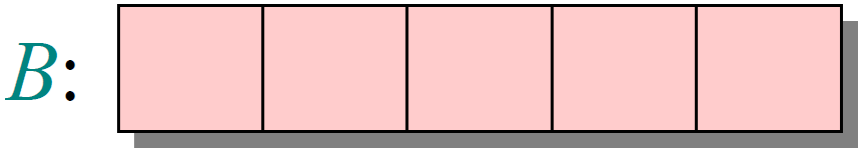
# … COUNTING SORT …



- Primer – prva petlja



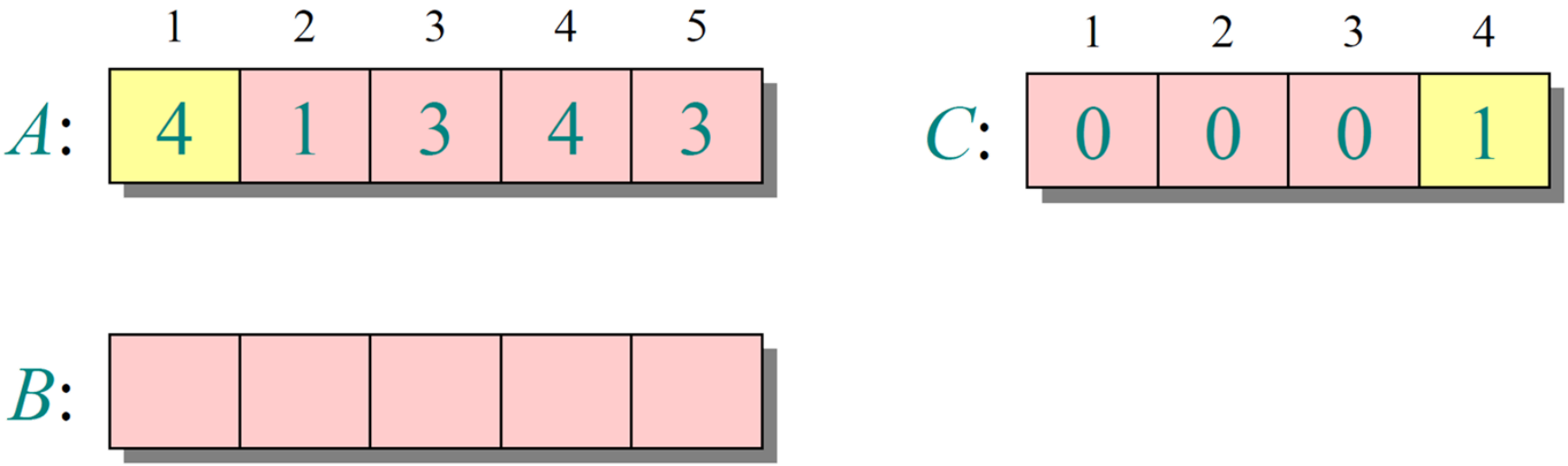




**for** $i = 0$ **to** $k$
$C[i] = 0$

# … COUNTING SORT …



- Primer – druga petlja



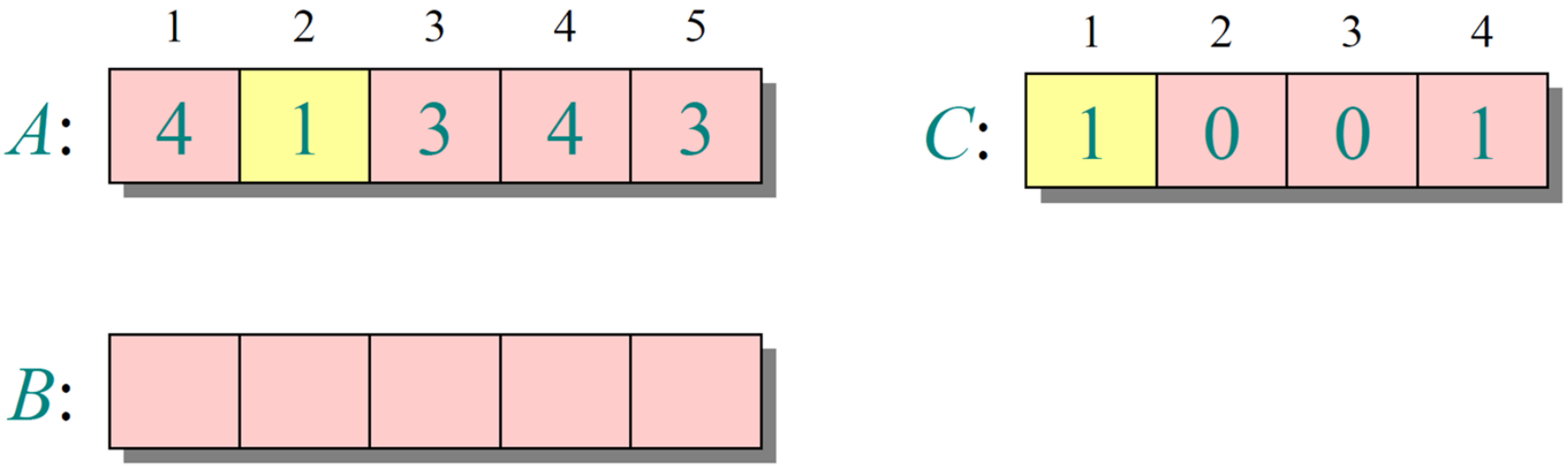

**for** $j = 1$ **to** $A.length$

$$C[A[j]] = C[A[j]] + 1$$

# … COUNTING SORT …



- Primer – druga petlja



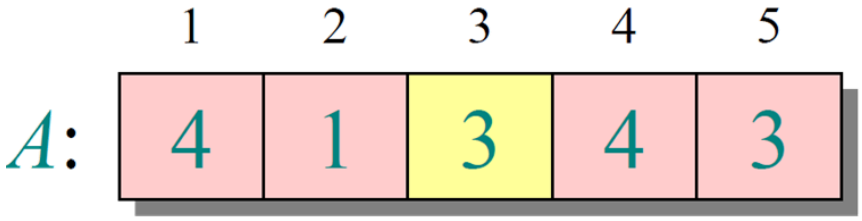

**for** $j = 1$ **to** $A.length$

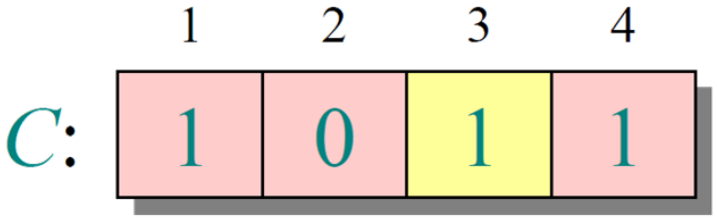
$$C[A[j]] = C[A[j]] + 1$$

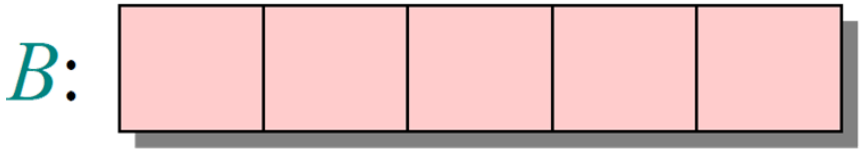
# … COUNTING SORT …



- Primer – druga petlja



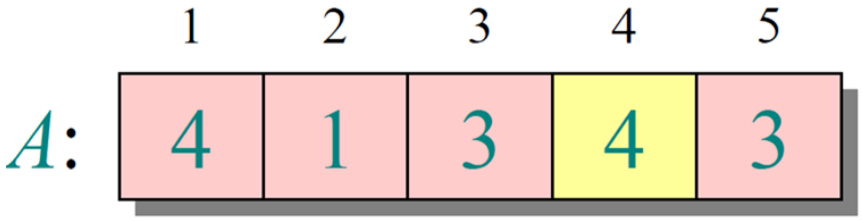

**for** $j = 1$ **to** $A.length$

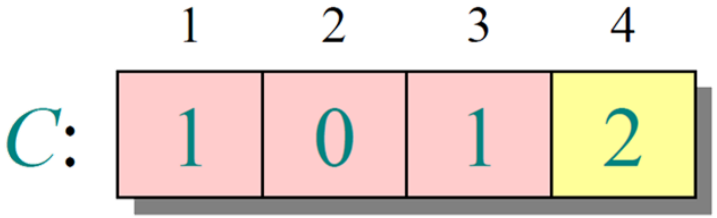
$$C[A[j]] = C[A[j]] + 1$$

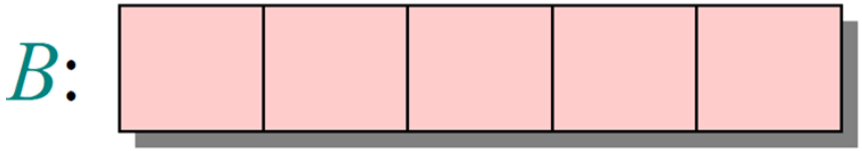
# … COUNTING SORT …



- Primer – druga petlja



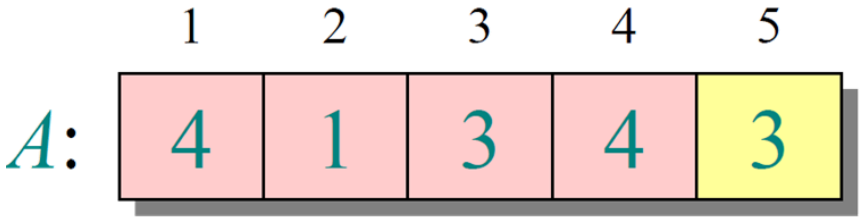



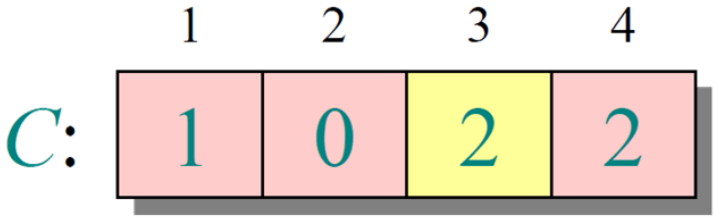

```
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
```

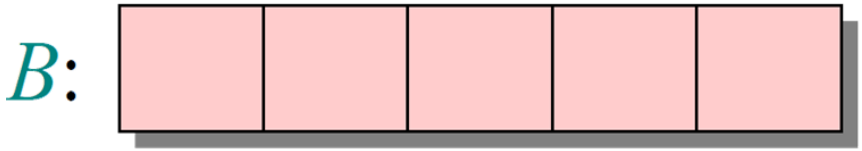
# … COUNTING SORT …



- Primer – druga petlja



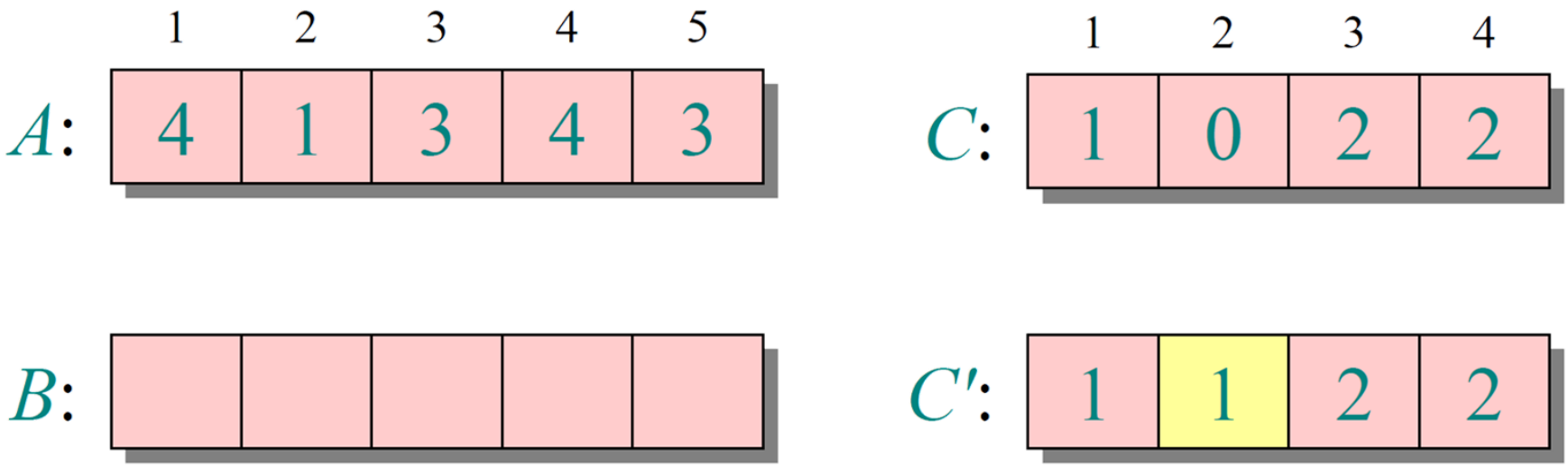




```
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
```

# … COUNTING SORT …



- Primer – treća petlja



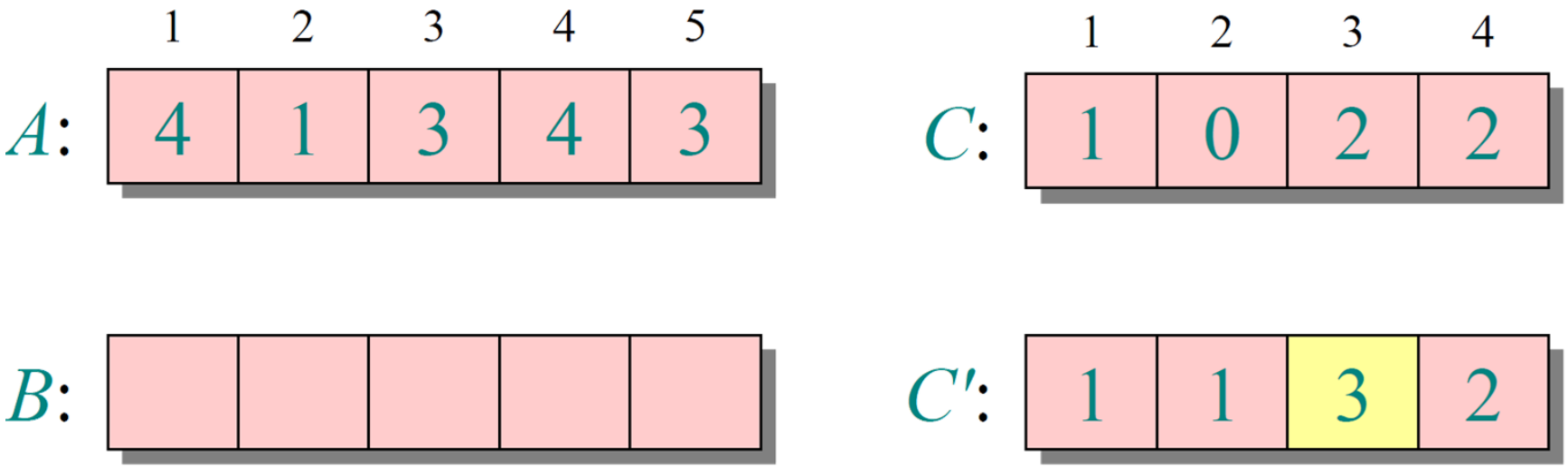

**for** $i = 2$ **to** $k$

$$C[i] = C[i] + C[i-1]$$

# … COUNTING SORT …



- Primer – treća petlja



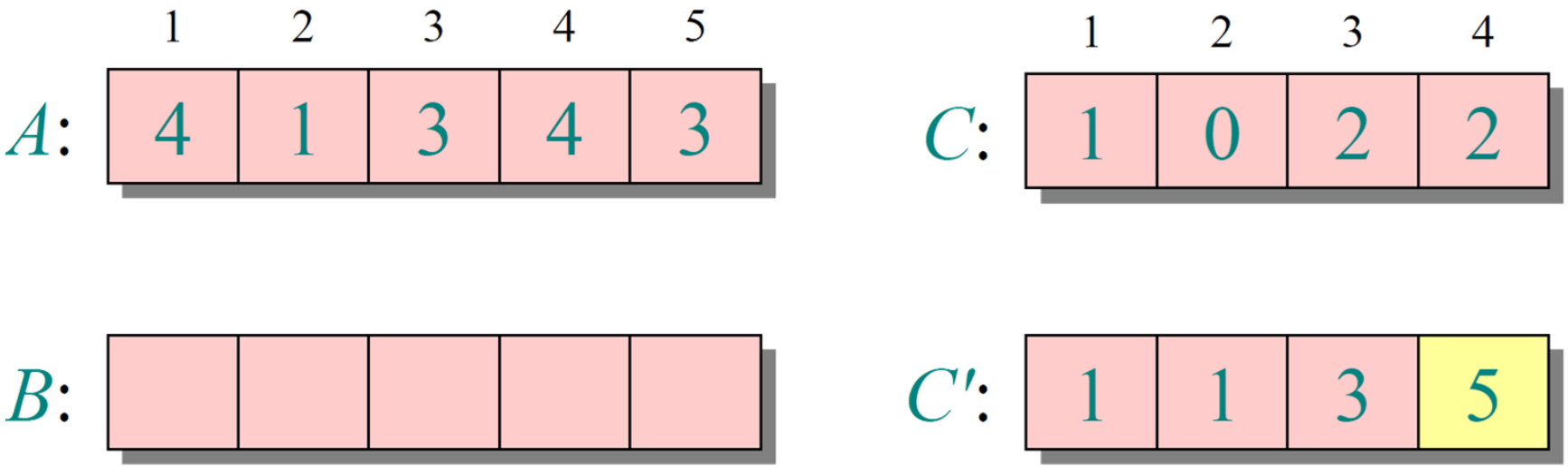

**for** $i = 2$ **to** $k$

$$C[i] = C[i] + C[i-1]$$

# … COUNTING SORT …



- Primer – treća petlja



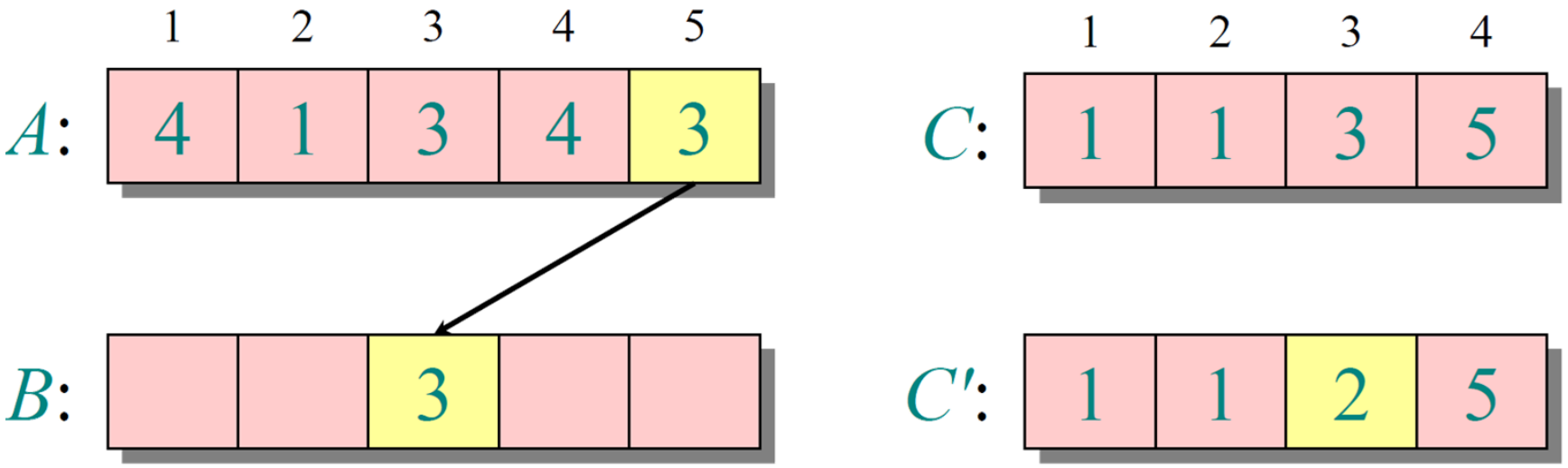

**for** $i = 2$ **to** $k$

$$C[i] = C[i] + C[i-1]$$

# … COUNTING SORT …



- Primer – četvrta petlja




**for** $j = A.length$ **downto** 1

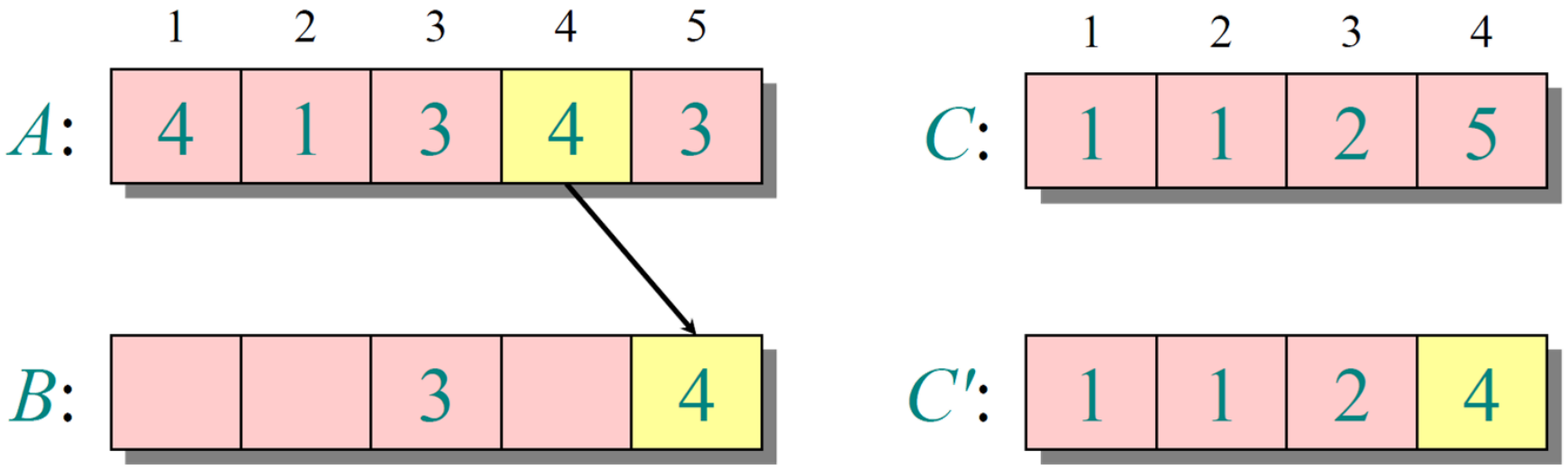
$B[C[A[j]]] = A[j]$

$C[A[j]] = C[A[j]] - 1$

# … COUNTING SORT …



- Primer – četvrta petlja




**for** $j = A.length$ **downto** 1

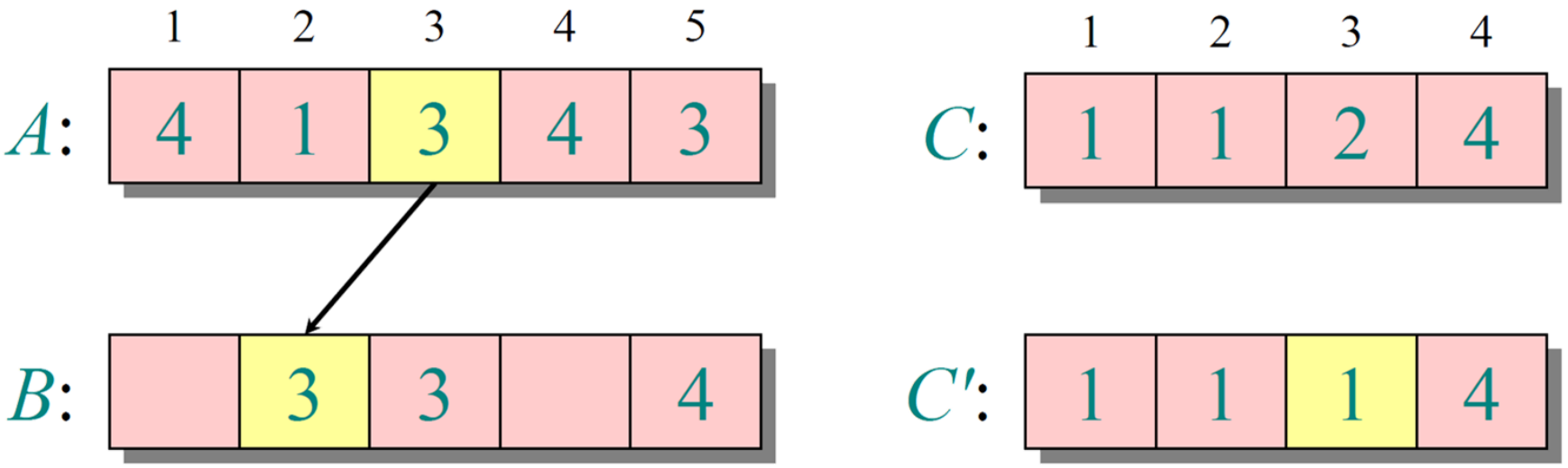
$$B[C[A[j]]] = A[j]$$

$$C[A[j]] = C[A[j]] - 1$$

# … COUNTING SORT …



- Primer – četvrta petlja




**for** $j = A.length$ **downto** 1

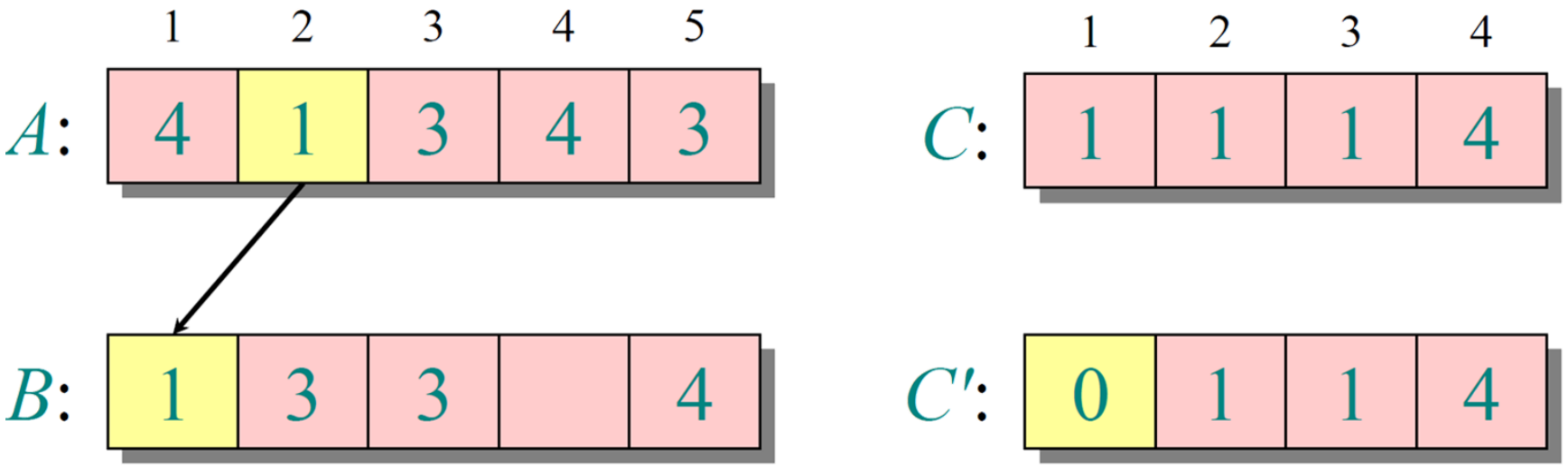
$B[C[A[j]]] = A[j]$

$C[A[j]] = C[A[j]] - 1$

# … COUNTING SORT …



- Primer – četvrta petlja




**for** $j = A.length$ **downto** 1

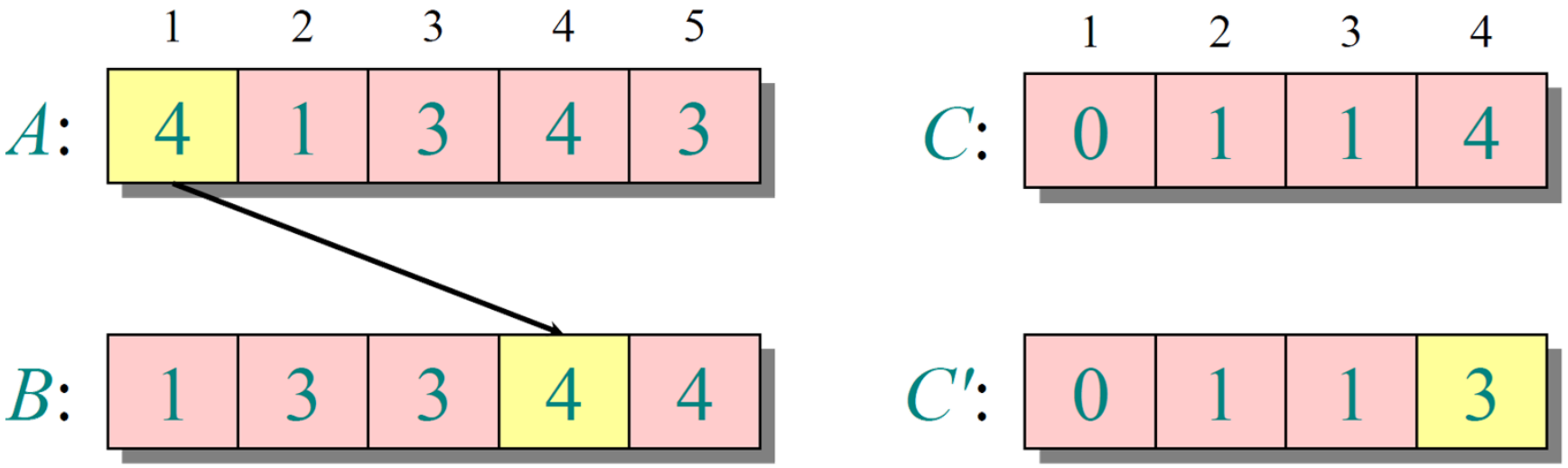
$B[C[A[j]]] = A[j]$

$C[A[j]] = C[A[j]] - 1$

# … COUNTING SORT



- Primer – četvrta petlja




**for** $j = A.length$ **downto** 1

$B[C[A[j]]] = A[j]$

$C[A[j]] = C[A[j]] - 1$

# COUNTING SORT ANALIZA …



- Kolika je kompleksnost counting sort algoritma?
- Izvršavanje svake petlje iznosi $\Theta(x)$, gde je $x$ broj izvršavanja petlje

```
                                COUNTING-SORT(A, B, k)
                    1   let C[0..k] be a new array
Θ(k)            2   for i = 0 to k
                3       C[i] = 0
Θ(n)            4   for j = 1 to A.length
                5       C[A[j]] = C[A[j]] + 1
                6   // C[i] now contains the number of elements equal to i.
Θ(k)            7   for i = 2 to k
                8       C[i] = C[i] + C[i − 1]
                9   // C[i] now contains the number of elements less than or equal to i.
Θ(n)           10   for j = A.length downto 1
               11       B[C[A[j]]] = A[j]
               12       C[A[j]] = C[A[j]] − 1
```

---

$$\Theta(n + k)$$

# … COUNTING SORT – ANALIZA

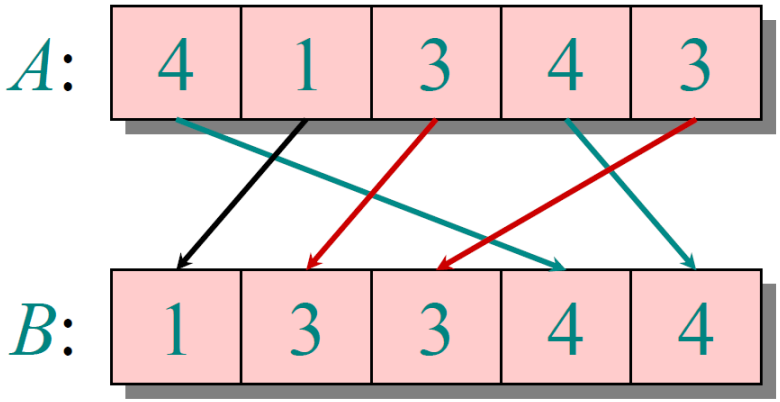


- Obično se Counting sort koristi kada je $\boldsymbol{k = O(n)}$, tako da je vreme izvršavanja algoritma $\Theta(\boldsymbol{n})$
- Ali zar nije najbolje vreme izvršavanja algoritma za sortiranje $\Omega(\boldsymbol{n} \log \boldsymbol{n})$?
- **Da, za algoritme koji koriste poređenje**
- Gde je ovde poređenje?
- Nema ga!

# COUNTING SORT – STABILNOST



- Vrlo važna osobina Counting sort algoritma je stabilnost
- Brojevi sa istom vrednošću se u izlaznom nizu pojavljuju u istom redosledu kao i u ulaznom




- Njegov osobina je važna za sledeću vrstu algoritma sortiranja: Radix sort

# RADIX SORT

# RADIX SORT…



- Korišćen od strane Herman Holeritovih mašina za sortiranje bušenih kartica tokom popisa u USA 1890. (mašine se mogu danas naći samo u muzejima)
- Kartice su imale 80 kolona po 12 mogućih mesta za udaranje rupice
- Ideja je bila da se za svaku karticu prouči odgovarajuća kolona i da se na osnovu toga gde je udarena rupica, kartica sortira u jednu od 12 predefinisanih grupa
- Sortiranje je radilo cifru po cifru, počev od cifre na najznačajnijem mestu
  - Kartice sa rupom na prvom mestu su na vrhu, pa zatim sve kartice sa rupicom na drugom mestu
- Za decimalne brojeve gleda se samo 10 kolona, preostale 2 se koriste za nenumeričke vrednost

# … RADIX SORT…

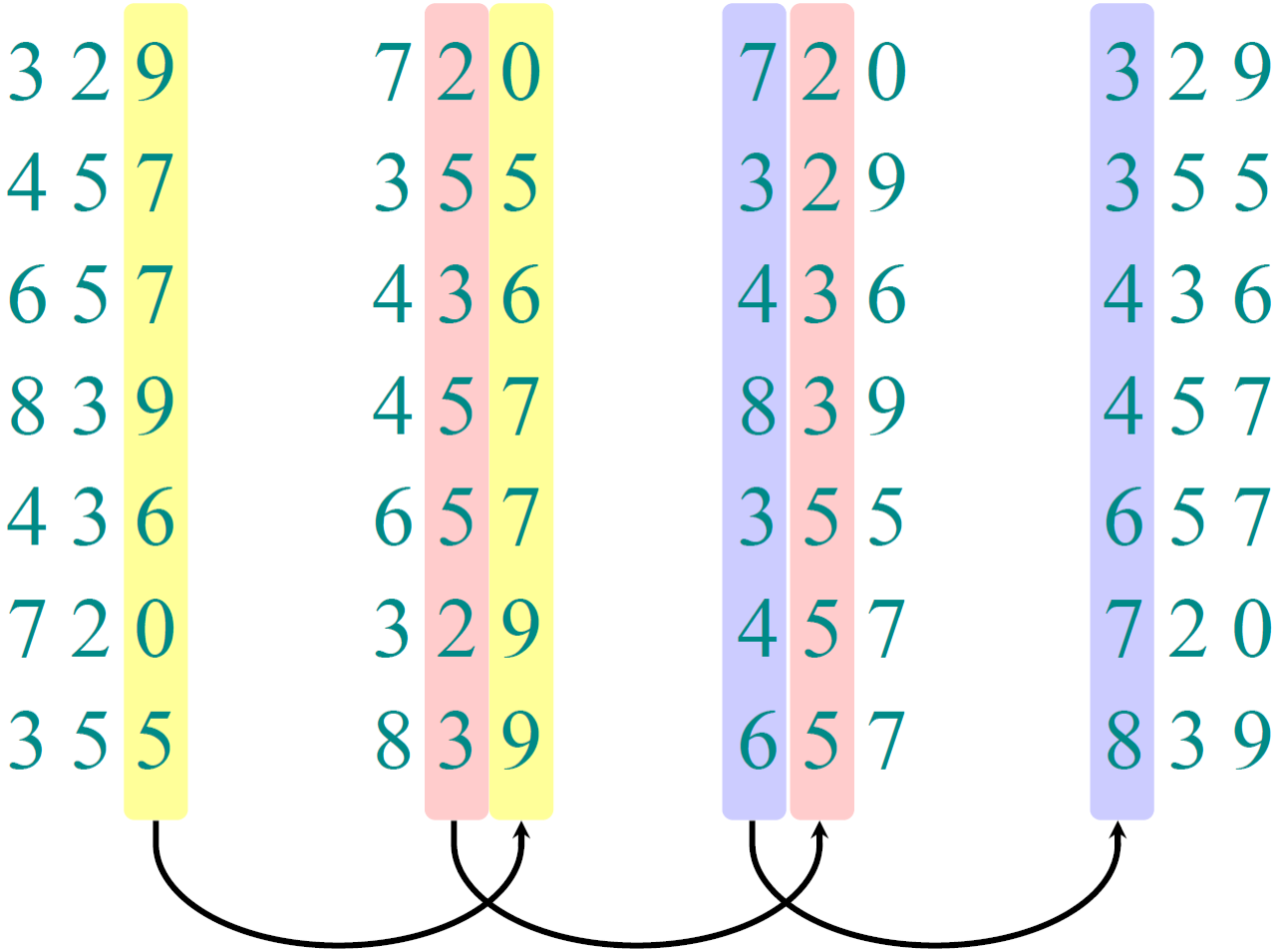


- Intuitivno  je da se krene sa sortiranjem preko cifre na najvišoj poziciji, pa da se posle sortira po svakoj od preostalih cifara
- Na žalost, pošto dok se sortira jedna grupa po jednoj cifri, čak 9 grupa ostaje ne iskorišćeno
- Ova procedura onda koristi jako puno među-grupa o kojima treba voditi računa
- Radix sort rešava problem sortiranja, tako što, kontra-intuitivno, kreće sortiranje od najniže cifre, pa onda spaja grupu ***0***, sa grupom ***1***, i svim preostalim u jednu grupu
- Postupak se nastavlja dok se ne sortira po svim ciframa
- Tako da je potrebno ***d*** prolaza kroz ceo niz kartica, gde je ***d*** broj cifara
- Da bi Radix algoritam radio potrebno je da algoritam za sortiranje po cifri bude stabilan

# … RADIX SORT…



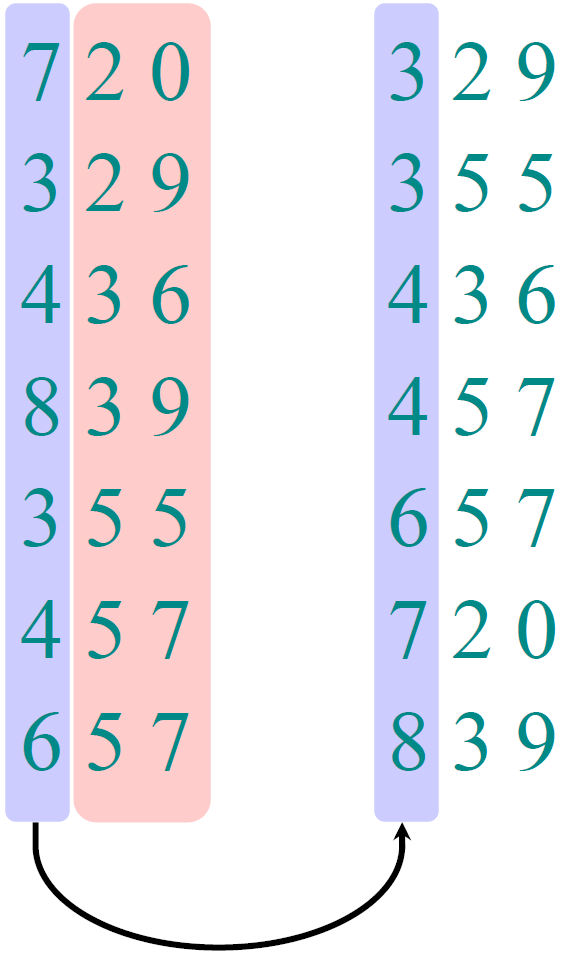
- Primer na grupi od 7 brojeva sa 3 cifre



# … RADIX SORT…



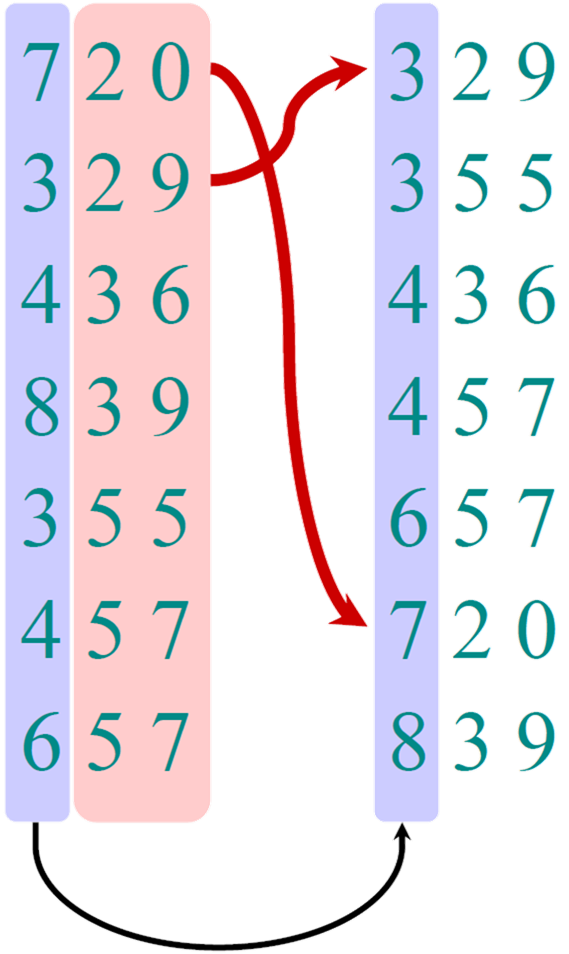
- Kako algoritam radi
  - Pretpostaviti da su brojevi sortirani po ***t – 1*** cifara na nižim pozicijama
  - Sortirati po cifri ***t***



# … RADIX SORT…



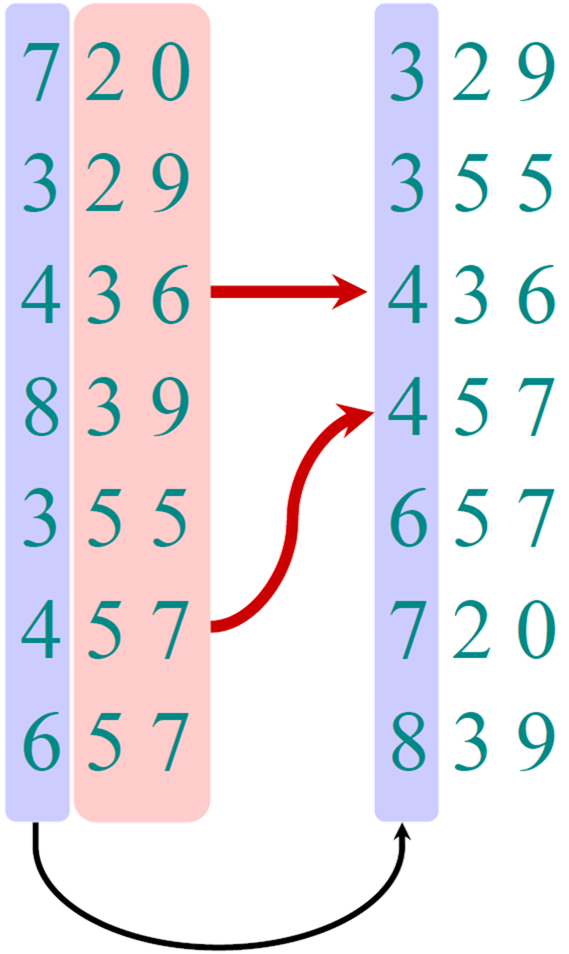
- Kako algoritam radi
  - Pretpostaviti da su brojevi sortirani po ***t – 1*** cifara na nižim pozicijama
  - Sortirati po cifri ***t***
    - Dva broja koji se razlikuju po cifri ***t*** će se pravilno sortirati



# … RADIX SORT…



- Kako algoritam radi
  - Pretpostaviti da su brojevi sortirani po ***t – 1*** cifara na nižim pozicijama
  - Sortirati po cifri ***t***
    - Dva broja koji se razlikuju po cifri ***t*** će se pravilno sortirati
    - Dva broja koja su jednaka u cifri ***t*** će se smestiti u odgovarajućem redosledu (kao u ulaznom nizu), jer stabilni algoritam održava taj redosled



# … RADIX SORT…



- Suštinski se Radix sort algoritam onda može svesti na sledeće

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

- ***n*** brojeva sa ***d*** brojem cifara koje mogu imati do ***k*** mogućih vrednosti, Radix sortiranje tačno sortira ove brojeve za $\Theta(\boldsymbol{d}(\boldsymbol{n}+\boldsymbol{k}))$ vremena
  - Dokaz ovoga zavisi od stabilnog sortiranja koje se koristi kao pomoćni algoritam
  - Ako su cifre u rasponu od ***0*** do ***k – 1*** , i ako ***k*** nije suviše veliko, onda je Counting sort algoritam pravi izbor
  - Svaki prolaz kroz ***n*** brojeva od ***d***-cifara traje $\Theta(\boldsymbol{n}+\boldsymbol{k})$
  - ***d*** prolaza onda traje $\Theta(\boldsymbol{d}(\boldsymbol{n}+\boldsymbol{k}))$

# … RADIX SORT…



- Kada je $d$ konstantno i kada je $k = O(n)$ Radix sortiranje se izvršava u linearnom vremenu
- Postoji određena fleksibilnost u tome kako podeliti brojeve po kojima se sortira u cifre
  - Ako je stabilni algoritam za sortiranja Counting sort
  - I neka se sortira $n$ brojeva predstavljenih sa $b$ bita
  - Svaki broj se može predstaviti kao da ima $b/r$ cifara u bazi $2^r$
  - Svaka cifra je zapravo ceo broj u rasponu $0$ do $2^r - 1$
  - U tom slučaju Counting sort traje: $k = 2^r - 1$
  - Na primer, 32-bitni broj se može posmatrati kao broj od četiri cifre, gde je svaka cifra veličine 8 bita
  - Onda je $b = 32$, $r = 8$, $k = 2^r - 1 = 255$ i $d = \frac{b}{r} = 4$
  - Svaki prolaz Counting sort algoritma traje: $\Theta(n + k) = \Theta(n + 2^r)$, ako ima $d$ prolaza, onda je ukupno vreme $\Theta(d(n + 2^r)) = \Theta(\left(\frac{b}{r}\right)(n + 2^r))$

# … RADIX SORT…



- Ako Radix sort sortira ulazni niz za $\boldsymbol{T(n,b) = \Theta\left(\left(\frac{b}{r}\right)(n + 2^r)\right)}$, koliko prolaza treba da bude, pa da sortiranje bude u linearnom vremenu?
- Očigledno treba izabrati $\boldsymbol{r}$ tako da $\boldsymbol{T(n,b)}$ bude minimalno:
  - Ako povećamo $\boldsymbol{r}$, to znači manji broj prolaza, ali $\boldsymbol{r \gg \log n}$, vreme sortiranja raste eksponencijalno
  - Ne želimo da $\boldsymbol{2^r \gg n}$, ali možemo birati asimptotski dovoljno veliko $\boldsymbol{r}$ , tako da i dalje zadovoljava da nije ispunjeno $\boldsymbol{2^r \gg n}$
  - Tako da, ako je $\boldsymbol{b < \lfloor \log n \rfloor}$, tada je za svako $\boldsymbol{r \leq b}$:

    $$\boldsymbol{\left(\frac{b}{r}\right)(n + 2^r) = (n + 2^r) = \Theta(n)}$$

  - Optimalan slučaj je naravno situacija kada je $\boldsymbol{r = b}$, jer je tada

    $$\boldsymbol{\left(\frac{b}{b}\right)(n + 2^b) = (n + 2^b) = \Theta(n)}$$

# … RADIX SORT…



- Ako Radix sort sortira ulazni niz za $\boldsymbol{T(n,b) = \Theta\left(\left(\frac{b}{r}\right)(n + 2^r)\right)}$, šta ako je $\boldsymbol{b \geq \lfloor \log n \rfloor}$?
- Optimalno je izabrati da je $\boldsymbol{r = \lfloor \log n \rfloor}$, jer daje nabolje vreme u konstantom faktoru
  - Ako je $\boldsymbol{r = \lfloor \log n \rfloor}$, onda je $\boldsymbol{\Theta\left(\left(\frac{b}{r}\right)(n + 2^r)\right) = \Theta\left(\frac{bn}{\log n}\right)}$
  - Očigledno je da, ako $\boldsymbol{r}$ raste preko $\lfloor \log n \rfloor$, $\boldsymbol{2^r}$ raste daleko brže nego $\boldsymbol{r}$ i onda je vreme izvršavanja algoritma $\boldsymbol{\Omega\left(\frac{bn}{\log n}\right)}$
  - Ako nasuprot tome, $\boldsymbol{r}$ ide ispod $\lfloor \log n \rfloor$, onda $\boldsymbol{\frac{b}{r}}$ raste i $\boldsymbol{n + 2^r}$ ostaje oko $\boldsymbol{\Theta(n)}$

# … RADIX SORT



- Da li ima smisla koristiti Radix sort algoritam?
- Da kada je ulaz veliki i kada je jednostavan za kodiranje i održavanje
- Na primer (za 32-bitne brojeve)
  - Radix ostvaruje najviše 3 prolaza na ulazu **≥ *2000*** brojeva
  - Merge sort i quick sort odrade minimum $\lceil \log \mathbf{2000} \rceil = \mathbf{11}$ prolaza
- Mana mu je što za razliku od quick sortiranja malo koristi lokalne reference, te ne zavisi puno od hardvera, tj. quick sort algoritam se može podesiti za konkretnu hardversku implementaciju (radi daleko bolje na savremenim arhitekturama računara)