Sortiranje nizova
- elementarne metode sortiranja -

Strukture podataka i algoritmi 2

Sortiranje

- Sortiranje kolekcija podataka je jedna od fundamentalnih računarskih operacija
- Sortirati kolekciju podataka znači preurediti je tako da elementi budu u nekom poretku
  - Leksikografski poredak za stringove
  - Neopadajuće ili nerastuće kolekcije brojeva
  - Hronološki poredak za vremenske događaje
- Sortirati se mogu samo one kolekcije podataka čiji su svi elementi međusobno uporedivi

Sortiranje nizova

- Kolekcije podataka se mogu predstaviti nizovima
- Problem sortiranja niza
  - Preurediti elemente niza *A* veličine *m* tako da elementi niza budu u neopadajućem ili nerastućem poretku
    - $A[0] \leq A[1] \leq A[2] \leq \ldots \leq A[m - 1]$ (neopadajući poredak)
    - $A[0] \geq A[1] \geq A[2] \geq \ldots \geq A[m - 1]$ (nerastući poredak)
  - U nastavku ćemo podrazumevati da sortiranje prevodi niz u neopadajući poredak
  - Postupak koji niz prevodi u neopadajući poredak se trivijalno modifikuje u postupak koji niz prevodi u nerastući poredak

Java interfejsi

- Osnovni OO koncepti programskog jezika Java su klase i interfejsi
  - Java program je kolekcija klasa i interfejsa (koji se mogu grupisati po paketima)
- Java klase implementiraju Java interfejse, Java interfejsi su apstraktne specifikacije Java klasa
- Interfejs se sastoji od definicija konstanti i zaglavlja metoda
- Interfejs je ugovor: ako **ne-apstraktna** klasa *A* implementira interfejs *I* tada *A* mora definisati sva zaglavlja data u *I*
- Interfejsi će detaljnije biti pokriveni kursom OOP1

Primer...

```
interface EchoInterface {
	void echo(String t);
}

class SimpleEcho implements EchoInterface {
	public void echo(String t) {
		System.out.println(t);
	}
}

class AdvancedEcho implements EchoInterface {
	private int echoCounter = 0;

	public void echo(String t) {
		if (t == null)
			return;
		System.out.println(t);
		echoCounter++;
	}

	public int getEchoCounter() {
		return echoCounter;
	}
}
```

Genričke klase

- **Generička klasa:** klasa sa *type* parametrima, klasa koja sadrži nekonkretizovane tipove (klase)

```
public class Box<T> {
    private T content;

    public Box(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}
```

```
Box<Integer> b1 = new Box<Integer>(42);
Box<String> b2 = new Box<String>("Pera");
...
int b1Cont = b1.getContent();
String b2Cont = b2.getContent();


public class TwoCompartmentBox<X, Y> {
    private X compartment1;
    private Y compartment2;
    ...
}
```

- Generičke klase omogućuju opšte (generičke) implementacije algoritama i tipova podataka
  - *Code reuse* princip: *write once reuse anywhere*

- Interfejsi i metode takođe mogu biti generički

```
interface EchoI<T> {
     void echo(T obj);
}

class SimEcho<T> implements EchoI<T> {
     public void echo(T obj) {
          System.out.println(obj);
     }
}

// type parametri se navode pre tipa povratne
// vrednosti metoda
public <T, V> void simpleGenericMethod(T arg1, V arg2) {
     System.out.println(arg1 + ", " + arg2);
}

public static <T> int linSearch(T[] arr, T el) {
     for (int i = 0; i < arr.length; i++)
          if (arr[i].equals(el))
               return i;
     return -1;
}
```

Comparable interfejs

- Prirodan poredak objekata neke klase u PJ Java definišemo implementacijom ***Comparable*** interfejsa
  - Ako klasa *A* implementira *Comparable* interfejs tada postoji poredak objekata klase *A*
  - Za dva objekta *x* i *y* klase *A* možemo pitati da li su jednaki ili je jedan od njih veći/manji od drugog
- Sortirati se mogu samo one kolekcije podataka čiji su svi elementi međusobno uporedivi.
- Stoga u opštem slučaju sortiramo nizove objekata koji su instance klase koja implementira ***Comparable*** interfejs.

Comparable interfejs

- package java.lang;
  public interface Comparable<T> {
  int compareTo(T o);
  }

- Neka su *x* i *y* dva objekta klase koja implementira *Comparable* interfejs. Tada, **x.compareTo(y)** poredi *x* i *y* i vraća
  - Negativan broj ukoliko je *x* **pre** *y*
  - Nulu ukoliko su *x* i *y* **jednaki**
  - Pozitivan broj ukoliko je *x* **posle** *y*.

$$x.compareTo(y)\begin{cases} < 0, x < y \\ = 0, x = y \\ > 0, x > y \end{cases}$$

String implements Comparable<String>

```
String s1, s2;
int cmp = s1.compareTo(s2);
if (cmp < 0)
        S.O.P("Prvi string leksikografski manji");
else if (cmp == 0)
        S.O.P("Stringovi su jednaki");
else
        S.O.P("Prvi string leksikografski veći);
```

Primer klase koja implementira Comparable

```
public class Person implements Comparable<Person> {

	private String firstName, lastName;

	public Person(String firstName, String lastName) {
		this.firstName = firstName;
		this.lastName = lastName;
	}

	public int compareTo(Person o) {
		int cmpLastNames = lastName.compareTo(o.lastName);
		if (cmpLastNames != 0)
			return cmpLastNames;
		else
			return firstName.compareTo(o.firstName);
	}
}
```

Interfejs Comparable je generički

- Parametrizacija tipom daje upotrebljivost interfejsu koju ne treba “zloupotrebiti”

```
class Zabe { … }
class Babe implements Comparable<Zabe> {
    public int compareTo(Zabe o) {
        …
    }
}
```

Uglavnom imamo potrebu za

class A implements Comparable<A>

Comparator interfejs

- Implementacijom *Comparable* interfejsa definišemo prirodno uređenje objekata neke klase
- U praksi je često potrebno urediti objekte i po nekom drugom, neinherentnom poretku
  - Sortiranje stringova leksikografski (prirodno uređenje), ali stringove možemo sortirati i po dužini
  - Sortiranje osoba po JMBG-u (prirodno uređenje), ali i po imenu, godištu, visini, težini, primanjima, itd.
- Proizvoljan (eksterni) poredak objekata neke klase definišemo implementirajući ***Comparator*** interfejs

```
package java.util;
interface Comparator<T> {
    int compare(T o1, T o2);
}
```

$$c.compare(x,y)\begin{cases} <0, x<y \\ =0, x=y \\ >0, x>y \end{cases}$$

Referencijalni tipovi

- **Svaka promenljiva u Java programu ima tip**
- Klase su konkretni tipovi, dok su interfejsi apstraktni tipovi objekata (referencijalni tipovi)
- Ako je *c* objekat klase *C* koja implementira interfejs *I* tada je *C* tip objekta *c*, ali je i *I* tip objekta c.
- Drugim rečima, interfejs može biti tip promenljive
  - **I c = new C();**
  - Metod **void simple(I x)** možemo pozvati sa objektom bilo koje klase koja implementira interfejs I.

Sortiranje nizova

- Niz objekata klase T možemo sortirati
  - po prirodnom uređenju (koje postoji ako T implementira Comparable interfejs)
    static <T extends Comparable<T>> void sort(T[] arr)
  - po nekom eksternom kriterijumu koji je definisan proizvoljnim komparatorom za objekte klase T
    static <T> void sort(T[] arr, Comparator<T> cmp)
- Pretpostavićemo da nizovi objekata ne sadrže null vrednosti
  - Ograničenje koje se jednostavno prevazilazi a doprinosi jasnijoj vidljivosti suštine

Sortiranje grubom silom

- Niz A od n elemenata je sortiran (neopadajuće) akko
  za svako $0 \leq i < j < n$: $A[i] \leq A[j]$

- Za dva elementa niza A sa indeksima i i j, $i < j$, kažemo da su u inverziji ako je $A[i] > A[j]$.

- Sortiranje grubom silom (engl. *brute force*) se svodi na poređenje svaka dva elementa u nizu, pri čemu elemente u inverziji razmenjujemo

- Za niz od n elemenata, ukupno imamo $n(n-1)/2$ poređenja → vremenska složenost sortiranja grubom silom je $O(n^2)$

- Stoga je sortiranje, u najlošijem slučaju, operacija kvadratne složenosti veličine ulaznog niza

```java
public class BruteForceSort {

	public static <T extends Comparable<T>> void sort(T[] arr) {
		for (int j = 1; j < arr.length; j++) {
			for (int i = 0; i < j; i++) {
				if (arr[i].compareTo(arr[j]) > 0) {
					T tmp = arr[i];
					arr[i] = arr[j];
					arr[j] = tmp;
				}
			}
		}
	}

	public static <T> void sort(T[] arr, Comparator<T> cmp) {
		for (int j = 1; j < arr.length; j++) {
			for (int i = 0; i < j; i++) {
				if (cmp.compare(arr[i], arr[j]) > 0) {
					T tmp = arr[i];
					arr[i] = arr[j];
					arr[j] = tmp;
				}
			}
		}
	}
}
```

Primer korišćenja – sortiranje radnika

```
class Radnik implements Comparable<Radnik> {
	private String ime;
	private int plata;

	public Radnik(String ime, int plata) {
		this.ime = ime;
		this.plata = plata;
	}

	public String toString() {
		return ime + ", plata: " + plata;
	}

	public String getIme() { return ime;   }
	public int getPlata()  { return plata; }

	public int compareTo(Radnik drugiRadnik) {
		return ime.compareTo(drugiRadnik.ime);
	}
}

class KomparatorPoPlati implements Comparator<Radnik> {
	public int compare(Radnik r1, Radnik r2) {
		return r2.getPlata() - r1.getPlata();
	}
}
```

```java
public static void main(String[] args) {
	Radnik[] radnici = new Radnik[4];
	radnici[0] = new Radnik("Mika", 2000);
	radnici[1] = new Radnik("Tika", 1000);
	radnici[2] = new Radnik("Zika", 5000);
	radnici[3] = new Radnik("Pera", 3000);

	BruteForceSort.sort(radnici);
	for (int i = 0; i < radnici.length; i++) {
		System.out.println(radnici[i]);
	}

	System.out.println("Radnici sortirani po plati... ");
	BruteForceSort.sort(radnici, new KomparatorPoPlati());
	for (int i = 0; i < radnici.length; i++) {
		System.out.println(radnici[i]);
	}
}
```

```
Mika, plata: 2000
Pera, plata: 3000
Tika, plata: 1000
Zika, plata: 5000
Radnici sortirani po plati...
Zika, plata: 5000
Pera, plata: 3000
Mika, plata: 2000
Tika, plata: 1000
```

U nastavku...

- Videćemo i ostale algoritme sortiranja, ali...
- ... samo po prirodnom uređenju objekata
- Postupak sortiranja po prirodnom uređenju se trivijalno modifikuje u postupak sortiranja po proizvoljnom komparatoru
- a[i].compareTo(a[j]) → c.compare(a[i], a[j]), gde je c neki komparator

Bogosort

- Jedan od najnefikasnijih algoritama za sortiranje nizova.

- Randomizovani (engl. *randomized*) algoritam: oslanja se na upotrebu generatora pseudo-slučajnih brojeva
  - **Monte Karlo algoritmi:** kontrolisano vreme izvršavanja, rezultat izvršavanja je korektan sa nekom verovatnoćom
  - **Las Vegas algoritmi:** rezultat je uvek korektan, ali nemamo kontrolu nad vremenom izvršavanja

- Las Vegas sortiranje: dok niz ne postane sortiran napravi random permutaciju niza

```java
public class Bogosort {

	public static <T extends Comparable<T>> void sort(T[] arr) {
		while (!sorted(arr)) {
			shuffle(arr);
		}
	}

	private static <T extends Comparable<T>> boolean sorted(T[] arr) {
		for (int i = 0; i < arr.length - 1; i++) {
			if (arr[i].compareTo(arr[i + 1]) > 0)
				return false;
		}

		return true;
	}

	private static <T extends Comparable<T>> void shuffle(T[] arr) {
		for (int i = arr.length - 1; i > 0; i--) {
			int rndIndex = (int) (Math.random() * i);
			T tmp = arr[rndIndex];
			arr[rndIndex] = arr[i];
			arr[i] = tmp;
		}
	}
}
```

Elementarne metode sortiranja

- Elementarne metode sortiranja su unapređenja sortiranja grubom silom koja su kvadratne složenosti.

- **Ideja:** smanjivati broj parova koji su inverziji tako da niz uvek možemo podeliti na sortirani i nesortirani deo niza

- Sortiranje umetanjem (*insertion sort*), sortiranje izabiranjem (*selection sort*) i sortiranje razmenom (*bubble, exchange sort*)

Sortiranje umetanjem

- Ideja:
  - niz je sastavljen od sortiranog i nesortiranog dela pri čemu je sortirani deo sa leve strane
  - prvi element iz nesortiranog dela ubaci u sortirani deo na odgovarajuće mesto (sortirani deo niza se tako povećava za jedan element)
  - ponavljaj prethodni korak dok ceo niz ne postane sortiran
  - početak: sortirani deo je prvi element niza, ostatak niza je nesortirani deo

Sortiranje umetanjem

- Neka je deo niza A od indeksa 0 do *i* – 1 sortiran
  - Prvi element iz nesortiranog dela je na poziciji A[*i*]
- U svakom koraku povećavamo sortirani deo niza za jedan element
  - *i* varira u opsegu [1, A.length – 1] sa korakom 1
- Ako je A[*i*] ≥ A[*i* – 1] ne radimo ništa **(zašto?)**
- U suprotnom treba da nadjemo indeks *j* manji od *i* takav da je A[*j*] ≤ A[*i*]
  - A[*i*] tada treba da bude na poziciji *j* + 1
  - Elemente od A[*j* + 1] do A[*i* – 1] treba pomeriti za jedno mesto u desno da bi se napravilo mesto za *A*[*i*]
    - Element A[i – 1] će biti pomeren na A[*i*] stoga trebamo sačuvati referencu na A[*i*]
  - **Pomeranje elemenata vršimo dok tražimo indeks *j***

Sortiranje umetanjem

```
public static <T extends Comparable<T>> void insertionSort_v1(T[] arr) {
	for (int i = 1; i < arr.length; i++) {
		if (arr[i].compareTo(arr[i - 1]) < 0) {
			T current = arr[i];

			// trazimo najvece j takvo da je arr[j] <= current
			int j = i - 1;
			while (j >= 0 && arr[j].compareTo(current) > 0) {
				arr[j + 1] = arr[j];
				j--;
			}

			arr[j + 1] = current;
		}
	}
}
```

Sortiranje umetanjem

- Da li je provera arr[i].compareTo(arr[i – 1]) < 0 neophodna?

```
public static <T extends Comparable<T>> void insertionSort(T[] arr) {
	for (int i = 1; i < arr.length; i++) {
		T current = arr[i];

		// trazimo najvece j takvo da je arr[j] <= current
		int j = i - 1;
		while (j >= 0 && arr[j].compareTo(current) > 0) {
			arr[j + 1] = arr[j];
			j--;
		}

		arr[j + 1] = current;
	}
}
```

Sortiranje umetanjem

- Razmotrimo sledeću situaciju: svi elementi sortiranog dela niza su veći od prvog elementa iz nesortiranog dela niza

- Tada se petlja

```
while (j >= 0 && arr[j].compareTo(current) > 0) {
        arr[j + 1] = arr[j];
        j--;
}
```

završava kada *j* postane -1.

- U jezicima koji ne podržavaju **lenjo izračunavanje logičkih izraza** ovo može da bude problem.

Sortiranje umetanjem u odsustvu lenjog izračunavanja logičkih izraza

- Možemo modifikovati postojeći kod na nekoliko načina, ilustrovaćemo ideju sa graničnikom
- **Minimalni element niza postavimo na prvo mesto, pre nego što započnemo sortiranje**
- Tada
  - *j* = 0 kada su svi elementi iz sortiranog dela (osim graničnika) manji od prvog elementa iz nesortiranog dela
  - Nema potrebe proveravati da li je *j* >= 0

Sortiranje umetanjem, verzija 3

```
public static <T extends Comparable<T>> void insertionSort_v3(T[] arr) {
	int minIndex = 0;
	for (int i = 1; i < arr.length; i++) {
		if (arr[i].compareTo(arr[minIndex]) < 0) {
			minIndex = i;
		}
	}

	if (minIndex != 0) {
		T tmp = arr[minIndex];
		arr[minIndex] = arr[0];
		arr[0] = tmp;
	}

	for (int i = 2; i < arr.length; i++) {
		T current = arr[i];

		// trazimo najvece j takvo da je arr[j] <= current
		int j = i - 1;
		while (arr[j].compareTo(current) > 0) {
			arr[j + 1] = arr[j];
			j--;
		}

		arr[j + 1] = current;
	}
}
```

Sortiranje izabiranjem

- Ideja:
  - Niz je sastavljen od sortiranog i nesortiranog dela pri čemu je sortirani deo sa desne strane
  - Tada su svi elementi u nesortiranom delu manji od svih elemenata u sortiranom delu niza
  - Nađemo maksimum nesortiranog dela niza i postavimo ga na kraj nesortiranog dela niza

- Drugim rečima, za niz A veličine n
  - Nađemo maks. A[0 .. n – 1] i zamenimo ga sa A[n – 1]
  - Nađemo maks. A[0 .. n – 2] i zamenimo ga sa A[n – 2]
  - Nađemo maks. A[0 .. n – 3] i zamenimo ga sa A[n – 3]
  - …
  - Nađemo maks. A[0 .. 1] i zamenimo ga sa A[1]

Sortiranje izabiranjem

```
public static <T extends Comparable<T>> void selectionSort(T[] arr) {
	// na pocetku trazimo maksimum celog niza
	for (int i = arr.length - 1; i >= 1; i--) {

		// trazimo maksimum u podnizu [0 .. i]
		int maxIndex = 0;
		for (int j = 1; j <= i; j++) {
			if (arr[maxIndex].compareTo(arr[j]) < 0) {
				maxIndex = j;
			}
		}

		// maksimum ide na kraj podniza [0 .. i]
		if (maxIndex != i) {
			T tmp = arr[i];
			arr[i] = arr[maxIndex];
			arr[maxIndex] = tmp;
		}
	}
}
```

Stabilni i nestabilni sortovi

- Za postupak sortiranja kažemo da je stabilan ukoliko za svako *i* i *j, i* < *j,* takvo da je A[*i*] = A[*j*] u polaznom nizu, element A[*i*] u sortiranom nizu bude pre A[*j*].

- Drugim rečima relativni redosled identičnih elemenata biva očuvan kod stabilnog sortiranja.

- U suprotnom, sortiranje je nestabilno.

- Sortiranje umetanjem je stabilno, sortiranje izabiranjem nije (može se napraviti da bude)
  - Tražimo maksimalni element i postavljamo ga na kraj
    - 4 1* 3 1**
    - 1** 1* 3 4

Važnost stabilnih sortova

- Želimo da sortiramo niz objekata po dva kriterijuma (primarnom i sekundarnom).
- Npr. niz radnika po plati (primarni kriterijum), a onda po godištu za radnike sa istom platom (sekunardni kriterijum).
- Ukoliko je sort stabilan tada niz objekata možemo sortirati prvo po sekundarnom kriterijumu, a onda po primarnom kriterijumu.

Sortiranje izabiranjem (nastavak)

- U drugoj varijanti sortiranja izabiranjem imamo da je sortirani deo niza sa leve strane
  - Tada su svi elementi u sortiranom delu manji od svih elemenata u nesortiranom delu niza
  - Nađemo minimum nesortiranog dela niza i postavimo ga na početak nesortiranog dela niza

- Drugim rečima, za niz A veličine n
  - Nađemo min. A[0 .. n – 1] i zamenimo ga sa A[0]
  - Nađemo min. A[1 .. n – 1] i zamenimo ga sa A[1]
  - Nađemo min. A[2 .. n – 1] i zamenimo ga sa A[2]
  - …
  - Nađemo min. A[n – 2 .. n – 1] i zamenimo ga sa A[n – 2]

Sortiranje izabiranjem, verzija 2

```
public static <T extends Comparable<T>> void selectionSort_v2(T[] arr) {
	for (int i = 0; i < arr.length - 1; i++) {

		// trazimo minimum u podnizu [i .. arr.length - 1]
		int minIndex = i;
		for (int j = i + 1; j < arr.length; j++) {
			if (arr[j].compareTo(arr[minIndex]) < 0) {
				minIndex = j;
			}
		}

		// minimum ide na pocetak podniza [i .. arr.length - 1]
		if (minIndex != i) {
			T tmp = arr[i];
			arr[i] = arr[minIndex];
			arr[minIndex] = tmp;
		}
	}
}
```

Sortiranje razmenom

- Ako bi smo prošli kroz ceo niz od početka ka kraju i svaka dva susedna elementa koji su u inverziji razmenili tada bi maksimalni element “isplivao” na kraj niza

  3 4 2 1

  3 4 2 1 → 3 2 4 1

  3 2 4 1 → 3 2 1 **4**

- Ako bi smo prošli kroz ceo niz od kraja ka početku i svaka dva susedna elementa koji su u inverziji razmenili tada bi minimalni element “isplivao” na početak niza

  3 4 1 2

  3 4 1 2 → 3 1 4 2

  3 1 4 2 → **1** 3 4 2

Sortiranje razmenom

- Verzija 1:
  - Neka je sortirani deo niza sa desne strane
  - Radimo isplivavanje maksimalnog elementa u nesortiranom delu niza
- Verzija 2:
  - Neka je sortirani deo niza sa leve strane
  - Radimo isplivavanje minimalnog elementa u nesortiranom delu niza

Sortiranje razmenom, verzija 1

- Neka je n dužina niza A i neka je
  - A[0 .. i] nesortirani deo niza
  - A[i + 1 .. n - 1] sortirani deo niza

- Radimo isplivavanje maksimalnog elementa u nesortiranom delu
  - Poredimo A[j] i A[j + 1] za j koje varira od 0 do i – 1

- Tada kraj nesortiranog dela niza (i) varira
  - od n – 1 (nesortirani deo niza je ceo niz)
  - do 1 (nesortirani deo niza čine prva dva elementa)

Sortiranje razmenom, verzija 1

```
public static <T extends Comparable<T>>
void exchangeSort(T[] arr) {
     // isplivavanje maksimuma
     // arr[0 .. i] je nesortirani deo niza, stoga
     // i varira od od arr.length - 1 do 1
     for (int i = arr.length - 1; i >= 1; i--) {
          for (int j = 0; j < i; j++) {
               if (arr[j].compareTo(arr[j + 1]) > 0) {
                    T tmp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = tmp;
               }
          }
     }
}
```

Sortiranje razmenom, verzija 2

- Neka je n dužina niza A i neka je
  - A[0 .. i - 1] sortirani deo niza
  - A[i .. n - 1] nesortirani deo niza
- Radimo isplivavanje minimalnog elementa u nesortiranom delu niza
  - Poredimo A[j] i A[j - 1] za j koje varira od n - 1 do i + 1
- Tada kraj početak dela niza (i) varira
  - od 0 (nesortirani deo niza je ceo niz)
  - do n - 2 (nesortirani deo niza čine poslednja dva elementa)

Sortiranje razmenom, verzija 2

```
public static <T extends Comparable<T>>
void exchangeSort_v2(T[] arr) {
     // isplivavanje minimuma
     // arr[i .. n - 1] je nesortirani deo niza, stoga
     // i varira od 0 do arr.length - 2
     for (int i = 0; i < arr.length - 1; i++) {
          for (int j = arr.length - 1; j > i; j--) {
               if (arr[j].compareTo(arr[j - 1]) < 0) {
                    T tmp = arr[j];
                    arr[j] = arr[j - 1];
                    arr[j - 1] = tmp;
               }
          }
     }
}
```

Unapređenje sortiranja razmenom

- Razmotrimo sledeći niz: 10 1 2 3 4 5 6 7 8 9
  - U prvoj iteraciji algoritma 10 isplivava na kraj
  - U ostalim iteracijama nema razmena elemenata niza

- Drugim rečima, sortiranje razmenom možemo prekinuti nakon iteracije u kojoj nije bilo razmene susednih elemenata (važi za obe verzije)

Sortiranje razmenom, verzija 3 (unapređenje verzije 1)

```
public static <T extends Comparable<T>>
void exchangeSort_v3(T[] arr) {
	// isplivavanje maksimuma
	// arr[0 .. i] je nesortirani deo niza, stoga
	// i varira od od arr.length - 1 do 1
	for (int i = arr.length - 1; i >= 1; i--) {
		boolean exchangeOccured = false;
		for (int j = 0; j < i; j++) {
			if (arr[j].compareTo(arr[j + 1]) > 0) {
				T tmp = arr[j];
				arr[j] = arr[j + 1];
				arr[j + 1] = tmp;
				exchangeOccured  = true;
			}
		}

		if (!exchangeOccured)
			break;
	}
}
```

Šelovo (Shell) sortiranje

- Postupak sortiranja koji je predložio Donalld Shell 1959. godine

- Osnovna ideja:
  - Ukupan broj parova elemenata u inverziji se može posmatrati kao mera neuređenosti niza.
  - Kod elementarnih metoda sortiranja, pri svakom poređenju susednih elemenata, ukupan broj inverzija se smanjuje za jedan.
  - Razmena nesusednih elemenata u nizu može smanjiti više od jedne inverzije.

Šelovo sortiranje

- “Podelimo” niz A dužine *n* na podnizove A_0, A_1..., A_{k-1} za neko k:
  - A_0 = (0, k, 2k, 3k, …)
  - A_1 = (1, k + 1, 2k + 1, 3k + 1, …)
  - A_2 = (2, k + 2, 2k + 2, 3k + 2, …)
  - …
  - A_{k-1} = (k - 1, 2k - 1, 3k - 1, 4k - 1, …)
- Drugim rečima, podniz A_j sadrži one indekse koji pri deljenju sa k daju ostatak j
- Sortiramo podnizove A_0, A_1..., A_{k-1} umetanjem
- Ponavljamo postupak za k koje se smanjuje od neke početne vrednosti do 1 sa nekim korakom
  - Kada je k = 1 imamo sortiranje umetanjem nad celim nizom.

Ilustracija Šelovog sortiranja

- Početni niz: A = (8, 6, 3, 9, 2, 7, 1)
- k: 4, 2, 1

- Za k = 4
  - A_0 = (8, 2) → (2, 8)
  - A_1 = (6, 7) → (6, 7)
  - A_2 = (3, 1) → (1, 3)
  - A_3 = (9) → (9)
  - Na kraju koraka imamo A = (2, 6, 1, 9, 8, 7, 3)

Ilustracija Šelovog sortiranja (nastavak)

- A = (2, 6, 1, 9, 8, 7, 3), k = 2
  - A_0 = (2, 1, 8, 3) → (1, 2, 3, 8)
  - A_1 = (6, 9, 7) → (6, 7, 9)
  - Na kraju koraka imamo A = (1, 6, 2, 7, 3, 9, 8)

- A = (1, 6, 2, 7, 3, 9, 8), k = 1
  - Sortiranje umetanjem na celom nizu koje A prevodi u (1, 2, 3, 6, 7, 8, 9)

Šelovo sortiranje

- Postavlja se pitanje kako varirati k?
  - U orginalnom Šelovom radu k varira na sledeći način: n/2, n/4, ..., $n/2^k$, ... ,1, gde je n veličina niza
  - *Shell sort* se značajno ubrzava kada k varira tako da bude stepen broja 2 uvećan ili umanjen za 1
  - **Tokuda, 1992:** Shell sort daje najbolje performanse kada se k se suksesivno smanjuje 2.25 puta
  - **Ciura, 2001:** Shell sort daje najbolje performnse kada k uzima redom vrednosti [701, 301, 132, 57, 23, 10, 4, 1]

Shell sort, Ciura's gap sequence

```
private static int[] gaps = {701, 301, 132, 57, 23, 10, 4, 1};

public static <T extends Comparable<T>> void sort(T[] arr) {
   for (int k : gaps) {
      if (k > arr.length)
         continue;

      // sortiraj svaki od k podnizova
      for (int i = 0; i < k; i++) {
         // sortiramo podniz (i, i + k, i + 2k, i + 3k, ...) umetanjem
         for (int j = i + k; j < arr.length; j += k) {
            T current = arr[j];
            int prevIndex = j - k;
            while (prevIndex >= i && arr[prevIndex].compareTo(current) > 0) {
               arr[prevIndex + k] = arr[prevIndex];
               prevIndex -= k;
            }
            arr[prevIndex + k] = current;
         }
      }
   }
}
```

Combsort

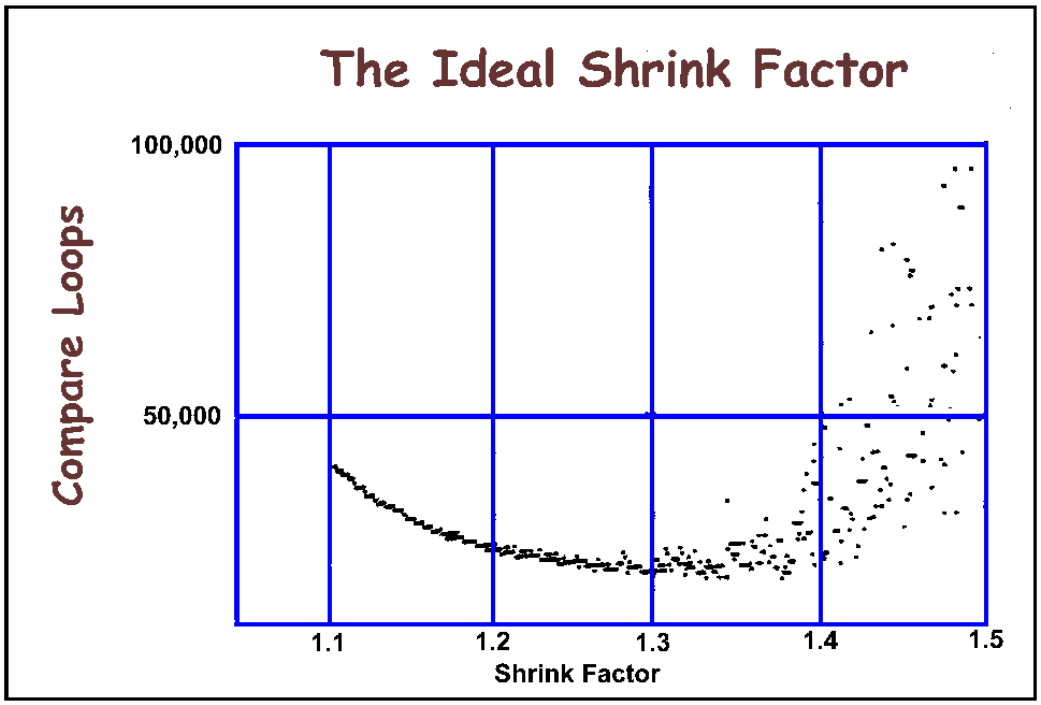
- Combsort se, slično Šelovom sortiranju, zasniva na razmeni nesusednih elemenata u nizu.
- Za razliku od Šelovog sortiranja, Combsort je unapređenje sortiranja razmenom (bubble sort).
- Posmatrajmo sortiranje razmenom kod kojeg isplivava maksimum na nizu A = (10, 9, 5, 6, 2, 1)
  - (9, 5, 6, 2, 1, 10)
  - (5, 6, 2, 1, 9, 10)
  - (5, 2, 1, 6, 9, 10)
  - (2, 1, 5, 6, 9, 10)
  - (1, 2, 5, 6, 9, 10)
- **Primećujemo da mali elementi sa kraja niza “sporo prilaze” početku niza, dok veliki elementi sa početka niza “brzo prilaze” kraju niza**

Combsort

- Autori CombSorta male elemente pri kraju niza zovu **kornjačama** (a velike elemente sa početka **zečevima**).
- Ideja CombSorta je eliminisati efekat kornjača razmenom nesusednih elemenata u nizu.
- CombSort algoritam:
  - **Comb-prolaz:** za svaki element niza proverimo da li je u inverziji sa elementom koji je od njega udaljen *k* pozicija u desno. Ako jeste razmenimo ih.
  - Ponavljamo **Comb-prolaz** dok niz ne postane sortiran smanjujući *k* od neke početne vrednosti do 1.
- Niz je sortiran kada je $k = 1$ i nije bilo razmena elemenata.
  - Drugim rečima, **Comb-prolaz** se za k = 1 ponavlja dokle god postoje elementi u inverziji što je suštinski bubble sort.

Combsort

- Postavlja se pitanje kako varirati *k*?
- Autori CombSorta su eksperimentalno utvrdili da CombSort daje najbolje performanse kada se
  - *k* sukcesivno smanjuje 1.3 puta
  - *k* = 9 i *k* = 10 preskočimo i koristimo k = 11



```java
private static int nextGap(int k) {
	k /= 1.3;
	if (k == 9 || k == 10) k = 11;
	else if (k < 1)        k = 1;
	return k;
}

public static <T extends Comparable<T>> void sort(T[] arr) {
	boolean sorted = false;
	int k = arr.length;

	do {
		k = nextGap(k);

		boolean exchOccured = false;
		for (int i = 0; i < arr.length - k; i++) {
			if (arr[i].compareTo(arr[i + k]) > 0) {
				T tmp = arr[i];
				arr[i] = arr[i + k];
				arr[i + k] = tmp;
				exchOccured = true;
			}
		}

		sorted = k == 1 && !exchOccured;
	} while (!sorted);
}
```