

Programiranje 1

Poglavje 5: Tabele

Luka Fürst

Izziv

- program, ki
 - prebere število n in n števil
 - ko prebere vsa števila, jih izpiše v istem vrstnem redu
- z dosedanjim znanjem to ni izvedljivo
- potrebovali bi spremenljivo število spremenljivk
- naloga je enostavno rešljiva s **tabelo**

Tabela

- zaporedje spremenljivk istega tipa
- posamezne spremenljivke so dosegljive prek **indeksov**
- spremenljivke se imenujejo **elementi** tabele

indeksi →	0	1	2	3	4	5
elementi →	25	42	17	54	20	36

Tabela

- indeksi se pričnejo z 0
- `tabela[indeks]`: element na indeksu *indeks*
- `tabela.length`: dolžina tabele (število elementov)
- veljavni indeksi: 0, 1, ..., `tabela.length` – 1
- dostop z neveljavnim indeksom sproži izjemo tipa `ArrayIndexOutOfBoundsException`

Tabela

0	1	2	3	4	5
25	42	17	54	20	36

```
System.out.println(tabela[0]); // 25
System.out.println(tabela[3]); // 54
tabela[3] = -9;
System.out.println(tabela[3]); // -9
System.out.println(tabela.length); // 6
System.out.println(tabela[tabela.length - 1]); // 36

System.out.println(tabela[-1]); // ArrayIndexOutOfBoundsException
System.out.println(tabela[6]); // ArrayIndexOutOfBoundsException
```

Izdelava tabele

- $T[]$: tip tabele z elementi tipa T
- izdelava s seznamom elementov
 - $T[] \text{ tabela} = \{element_0, element_1, \dots\};$
- izdelava s privzetimi vrednostmi elementov
 - $T[] \text{ tabela} = \text{new } T[\text{dolžina}];$
- dolžino tabele določimo ob izdelavi in je kasneje ne moremo več spreminjati

Privzete vrednosti

tip	vrednost
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	0.0f
double	0.0
char	'\0'
boolean	false
referenčni tipi	null

Izdelava tabele

```
int[] popolna = {6, 28, 496, 8128};

boolean[] metiKovanca = new boolean[100];
    // {false, false, ..., false}

String[] odgovori = {"da", "ne", "morda"};

char[] samoglasniki = {'a', 'e', 'i', 'o', 'u'};

float[] rezultati = new float[3]; // {0.0f, 0.0f, 0.0f}

double[] dolzinaNic = new double[0]; // {}
```


Sprehod po indeksih tabele

- splošen vzorec

```
for (int i = 0; i < t.length; i++) {  
    // i: trenutni indeks  
    // t[i]: element tabele na indeksu i  
    ...  
}
```

- primer

```
char[] samoglasniki = {'a', 'e', 'i', 'o', 'u'};  
for (int i = 0; i < samoglasniki.length; i++) {  
    System.out.printf("element na indeksu %d: %c%n",  
                      i, samoglasniki[i]);  
}
```

Sprehod po elementih tabele

- posebna oblika zanke for (**for-each**)

```
for (T element: t) {  
    // v prvem obhodu: element = t[0]  
    // v drugem obhodu: element = t[1]  
    // ...  
    // v zadnjem obhodu: element = t[t.length - 1]  
    ...  
}
```

- primer

```
char[] samoglasniki = {'a', 'e', 'i', 'o', 'u'};  
for (char crka: samoglasniki) {  
    System.out.println(crka);  
}
```

Rešitev motivacijskega primera

- pripravimo si tabelo dolžine n
- i -to število preberemo v i -ti element tabele
- izpišemo elemente tabele

```
System.out.print("Koliko števil želite vnesti? ");
int n = sc.nextInt();
int[] stevila = new int[n];

for (int i = 0; i < n; i++) {
    System.out.print("Vnesite število: ");
    stevila[i] = sc.nextInt();
}

for (int stevilo: stevila) {
    System.out.println(stevilo);
}
```

Vsota elementov tabele

- sprehodimo se po elementih in jih prištevamo tekoči vsoti

```
public static int vsota(int[] t) {  
    int vsota = 0;  
    for (int element: t) {  
        vsota += element;  
    }  
    return vsota;  
}
```

Indeks največjega elementa

- sprehodimo se po indeksih in vzdržujemo indeks doslej največjega elementa

```
public static int indeksMaksimuma(int[] t) {  
    int iMax = 0;  
    for (int i = 0; i < t.length; i++) {  
        if (t[i] > t[iMax]) {  
            iMax = i;  
        }  
    }  
    return iMax;  
}
```

- v zanki lahko *i* inicializiramo tudi na 1

Frekvence ocen

- program, ki prebere število n in n šolskih ocen in izpiše število posameznih ocen

```
Vnesite število učencev: 9
Vnesite oceno: 3
Vnesite oceno: 4
Vnesite oceno: 3
Vnesite oceno: 1
Vnesite oceno: 4
Vnesite oceno: 5
Vnesite oceno: 1
Vnesite oceno: 4
Vnesite oceno: 3
1: 2
2: 0
3: 3
4: 3
5: 1
```

Prva rešitev

- števci st1, st2, st3, st4, st5
- nastavimo jih na 0
- ko preberemo oceno, povečamo ustrezní števec

```
int st1 = 0, st2 = 0, st3 = 0, st4 = 0, st5 = 0;
for (int i = 1; i <= stUcencev; i++) {
    System.out.print("Vnesite oceno: ");
    int ocena = sc.nextInt();
    switch (ocena) {
        case 1: st1++; break;
        case 2: st2++; break;
        case 3: st3++; break;
        case 4: st4++; break;
        case 5: st5++; break;
    }
}
System.out.printf("1: %d%n", st1);
System.out.printf("2: %d%n", st2);
System.out.printf("3: %d%n", st3);
System.out.printf("4: %d%n", st4);
System.out.printf("5: %d%n", st5);
```

Druga rešitev

- uvedemo tabelo števecv

```
int[] stevci = new int[5];
for (int i = 1; i <= stUcencev; i++) {
    System.out.print("Vnesite oceno: ");
    int ocena = sc.nextInt();
    switch (ocena) {
        case 1: stevci[0]++; break;
        case 2: stevci[1]++; break;
        case 3: stevci[2]++; break;
        case 4: stevci[3]++; break;
        case 5: stevci[4]++; break;
    }
}
for (int i = 0; i < stevci.length; i++) {
    System.out.printf("%d: %d%n", i + 1, stevci[i]);
}
```


Tretja rešitev

- ko preberemo oceno `ocena`, povečamo števec z indeksom `ocena - 1`

```
int[] stevci = new int[5];
for (int i = 1; i <= stUcencev; i++) {
    System.out.print("Vnesite oceno: ");
    int ocena = sc.nextInt();
    stevci[ocena - 1]++;
}
for (int i = 0; i < stevci.length; i++) {
    System.out.printf("%d: %d%n", i + 1, stevci[i]);
}
```

Iskanje elementa v urejeni tabeli

- `public static int poisci(int[] t, int x)`
- parametra
 - `t`: naraščajoče urejena tabela
 - `x`: iskani element
- rezultat
 - indeks elementa `x`, če se nahaja v tabeli
 - `-1`, če ga ni

Splošen postopek

- sprehodimo se po indeksih
- če naletimo na element, takoj vrnemo njegov indeks
- na koncu vrnemo -1

```
public static int poisci(int[] t, int x) {  
    for (int i = 0; i < t.length; i++) {  
        if (t[i] == x) {  
            return i;  
        }  
    }  
    return -1;  
}
```

Sprehod, ki upošteva urejenost

- potujemo po tabeli, dokler ne pridemo do konca oziroma do elementa, ki je večji ali enak x
- pogledamo, kje smo se ustavili

```
public static int poisci(int[] t, int x) {  
    int i = 0;  
    while (i < t.length && t[i] < x) {  
        i++;  
    }  
    return (i < t.length && t[i] == x) ? (i) : (-1);  
}
```

Dvojiško iskanje

- pogledamo element na sredini tabele
- če je enak iskanemu, smo ga našli
- če je večji od iskanega, lahko izločimo desno polovico tabele
- če je manjši od iskanega, lahko izločimo levo polovico tabele
- postopek ponovimo na tisti polovici tabele, ki je nismo izločili

Dvojiško iskanje števila 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	10	15	21	27	30	31	34	37	39	42	50	58	61	75
lm							s	dm						

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	10	15	21	27	30	31	34	37	39	42	50	58	61	75
lm								s	dm					

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	10	15	21	27	30	31	34	37	39	42	50	58	61	75
lm								s	dm					

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	10	15	21	27	30	31	34	37	39	42	50	58	61	75
										lm	dm	s		

Dvojiško iskanje števila 29

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	10	15	21	27	30	31	34	37	39	42	50	58	61	75
1m							s							dm

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	10	15	21	27	30	31	34	37	39	42	50	58	61	75
lm			s			dm								

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	10	15	21	27	30	31	34	37	39	42	50	58	61	75
				lm	s	dm								

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	10	15	21	27	30	31	34	37	39	42	50	58	61	75

lm
dm
s

Dvojiško iskanje

```
public static int poisci(int[] t, int x) {  
    int lm = 0;           // leva meja  
    int dm = t.length - 1; // desna meja  
    while (lm <= dm) {  
        int s = (lm + dm) / 2;  
        if (t[s] == x) { // element smo našli!  
            return s;  
        }  
        if (t[s] < x) {  
            lm = s + 1;  
        } else {  
            dm = s - 1;  
        }  
    }  
    return -1; // elementa ni v tabeli  
}
```


Primerjava navadnega in dvojiškega iskanja

- največ koliko elementov moramo pregledati pri tabeli velikosti n ?
- navadno iskanje
 - pregledati moramo največ n elementov
- dvojiško iskanje
 - pri tabeli s 7 elementi moramo pregledati največ 3 elemente
 - pri tabeli s 15 elementi moramo pregledati največ 4 elemente
 - pri tabeli z 31 elementi moramo pregledati največ 5 elementov
 - pri tabeli z $2^k - 1$ elementi moramo pregledati največ k elementov
 - pri tabeli z n elementi moramo pregledati največ $\lceil \log_2(n + 1) \rceil$ elementov

Primerjava navadnega in dvojiškega iskanja

- dvojiško iskanje je bistveno hitrejše od navadnega

število elementov tabele	maks. št. pregledanih elementov	
	navadno iskanje	dvojiško iskanje
10	10	4
100	100	7
1000	1000	10
10^4	10^4	14
10^5	10^5	17
10^6	10^6	20
10^7	10^7	24
10^8	10^8	27
10^9	10^9	30

- žal pa deluje samo za urejene tabele ...

Urejanje tabele

- `public static int uredi(int[] t)`
- naraščajoče uredi tabelo t z n elementi
- eden od najpomembnejših problemov v računalništvu
- veliko različnih algoritmov
- urejanje z navadnim vstavljanjem
 - enostavno
 - stabilno
 - učinkovitejše od večine drugih enostavnih postopkov

Urejanje z navadnim vstavljanjem

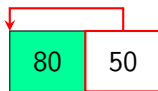
```
public static int uredi(int[] t) {  
    for (int i = 1; i < t.length; i++) {  
        // podtabela t[0..i-1] je že urejena  
        element t[i] vstavi na ustrezno mesto v podtabelo t[0..i-1]  
        // sedaj je urejena tudi podtabela t[0..i]  
    }  
}
```

Urejanje z navadnim vstavljanjem

80	50	75	30	45	60	95	20
----	----	----	----	----	----	----	----

80	50	75	30	45	60	95	20
----	----	----	----	----	----	----	----

80	50	75	30	45	60	95	20
----	----	----	----	----	----	----	----




50	80	75	30	45	60	95	20
----	----	----	----	----	----	----	----

Urejanje z navadnim vstavljanjem


50	80	75	30	45	60	95	20
----	----	----	----	----	----	----	----

50	80	75	30	45	60	95	20
----	----	----	----	----	----	----	----



50	75	80	30	45	60	95	20
----	----	----	----	----	----	----	----

50	75	80	30	45	60	95	20
----	----	----	----	----	----	----	----




30	50	75	80	45	60	95	20
----	----	----	----	----	----	----	----

Urejanje z navadnim vstavljanjem


30	50	75	80	45	60	95	20
----	----	----	----	----	----	----	----

30	50	75	80	45	60	95	20
----	----	----	----	----	----	----	----



30	45	50	75	80	60	95	20
----	----	----	----	----	----	----	----

30	45	50	75	80	60	95	20
----	----	----	----	----	----	----	----




30	45	50	60	75	80	95	20
----	----	----	----	----	----	----	----

Urejanje z navadnim vstavljanjem

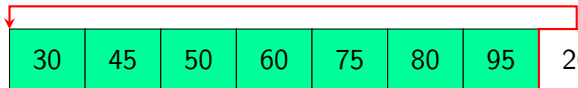
30	45	50	60	75	80	95	20
----	----	----	----	----	----	----	----

30	45	50	60	75	80	95	20
----	----	----	----	----	----	----	----



30	45	50	60	75	80	95	20
----	----	----	----	----	----	----	----

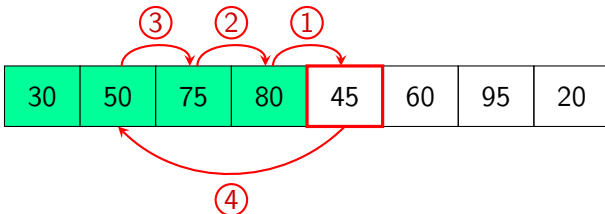
30	45	50	60	75	80	95	20
----	----	----	----	----	----	----	----



20	30	45	50	60	75	80	95
----	----	----	----	----	----	----	----

Urejanje z navadnim vstavljanjem

- vstavljanje izvirnega elementa na ustrezno mesto
 - potujemo od izvirnega elementa na levo, dokler ne prispemo do levega roba tabele oziroma do elementa, ki je manjši ali enak izvirnemu
 - vse elemente na poti premaknemo za eno mesto v desno
 - na »izpraznjeno« mesto postavimo izvirni element



Urejanje z navadnim vstavljanjem

```
public static void uredi(int[] t) {  
    for (int i = 1; i < t.length; i++) {  
        // vstavi element t[i] v podtabelo t[0..i-1]  
        int element = t[i];  
        int j = i - 1;  
        while (j >= 0 && t[j] > element) {  
            t[j + 1] = t[j];  
            j--;  
        }  
        t[j + 1] = element;  
    }  
}
```

Praštevila do n z Eratostenovim sitom

- eden od najučinkovitejših postopkov za iskanje praštevil
- pričnemo s $p = 2$
- označimo vse večkratnike števila p od $2p$ do n
- p nastavimo na prvo neoznačeno število, večje od trenutnega p
- označimo vse večkratnike števila p od $2p$ do n
- p nastavimo na prvo neoznačeno število, večje od trenutnega p
- označimo vse večkratnike števila p od $2p$ do n
- ...
- ponavljamo do $p \leq \sqrt{n}$
- neoznačena števila med 2 in n so praštevila

Praštevilna do n z Eratostenovim sitom

```
boolean[] sestavljeno = new boolean[n + 1];  
// cilj: sestavljeno[i] == true  $\iff$  število  $i$  je sestavljeno  
  
p = 2;  
while (p  $\leq \sqrt{n}$ ) {  
    za  $i \in \{2p, 3p, 4p, \dots\}$  nastavimo sestavljeno[i] = true  
    p = najmanjši indeks  $i > p$ , pri katerem velja !sestavljeno[i]  
}  
  
izpišemo vsa števila  $i \geq 2$ , pri katerih velja !sestavljeno[i]
```

Praštevila do n z Eratostenovim sitom

```
boolean[] sestavljeno = new boolean[n + 1];
int meja = (int) Math.round(Math.sqrt(n));

int p = 2;
while (p <= meja) {
    for (int i = 2 * p; i <= n; i += p) {
        sestavljeno[i] = true;
    }
    do {
        p++;
    } while (p <= meja && sestavljeno[p]);
}

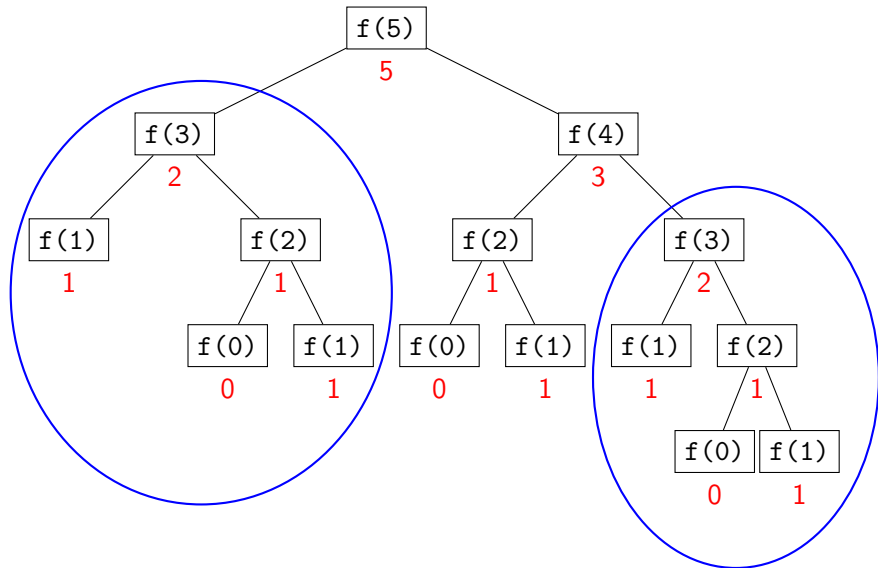
for (int i = 2; i <= n; i++) {
    if (!sestavljeno[i]) {
        System.out.println(i);
    }
}
```

Fibonacci (repriza)

- zaporedje $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-2} + F_{n-1}$
- 0, 1, 1, 2, 3, 5, 8, 13 ...
- osnovna rekurzivna rešitev

```
public static int f(int n) {  
    if (n <= 1) {  
        return n;  
    }  
    int pp = f(n - 2);  
    int p = f(n - 1);  
    return pp + p;  
}
```

Drevo rekurzivnih klicev za $n == 5$



Zakaj je rekurzivna rešitev neučinkovita?

- veliki deli drevesa se večkrat računajo
- problem je tem hujši, čim večji je n

Rešitev

- pripravimo si tabelo memo velikosti $n + 1$
 - tabela na začetku vsebuje same ničle
- ko izračunamo vrednost za nek k , jo shranimo v $\text{memo}[k]$
- pred računanjem vrednosti $f(k)$ preverimo, ali velja $\text{memo}[k] > 0$
- če velja, vrednost uporabimo, sicer jo izračunamo in shranimo
- **memoizacija** (ne memo-r-izacija!)

Rešitev

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int[] memo = new int[n + 1];
    System.out.println(f(n, memo));
}

public static int f(int n, int[] memo) {
    if (n <= 1) {
        return n;
    }
    if (memo[n] > 0) {
        return memo[n];
    }
    int pp = f(n - 2, memo);
    int p = f(n - 1, memo);
    memo[n] = pp + p;
    return memo[n];
}
```

Vprašanje

- Kaj izpiše sledeči izsek kode?

```
int[] a = {1, 2, 3};  
System.out.println(Arrays.toString(a));  
  
int[] b = a;  
b[0] = 42;  
System.out.println(Arrays.toString(b));  
System.out.println(Arrays.toString(a));
```

- `Arrays.toString(tabela)` vrne vsebino tabele v obliki niza [*element*₀, *element*₁, ...]
- razred `Arrays` se nahaja v paketu `java.util`

Vprašanje

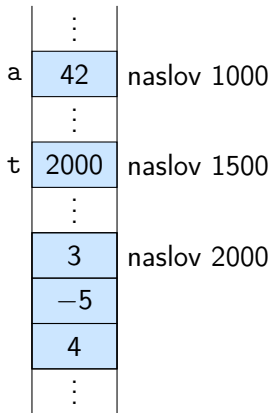
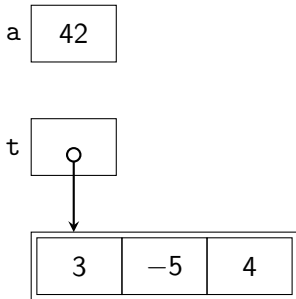
```
int[] a = {1, 2, 3};  
System.out.println(Arrays.toString(a));  
  
int[] b = a;  
b[0] = 42;  
System.out.println(Arrays.toString(b)); // [42, 2, 3]  
System.out.println(Arrays.toString(a)); // [42, 2, 3]
```

Primitivni in referenčni tipi

- **primitivni tipi**
 - byte, short, int, long, float, double, boolean, char
 - spremenljivka takega tipa vsebuje **vrednost**
- **referenčni tipi**
 - $T[]$, String, Scanner ...
 - spremenljivka takega tipa vsebuje **pomnilniški naslov** tabele oz. objekta
 - pomnilniški naslov = **kazalec** = **referenca**
 - podatek o lokaciji tabele/objekta v pomnilniku

Primitivni in referenčni tipi

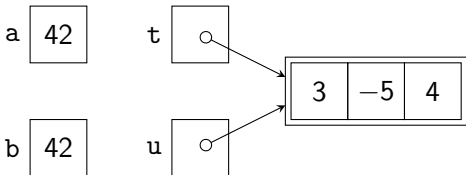
```
int a = 42;  
int[] t = {3, -5, 4};
```



Kopiranje spremenljivk

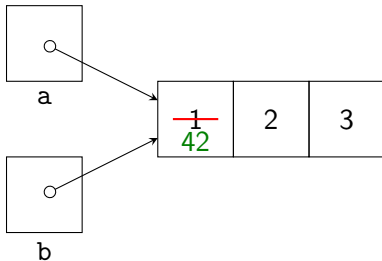
- kopira se **vrednost** spremenljivke
 - vrednost spremenljivke referenčnega tipa je kazalec na tabelo/objekt

```
int a = 42;  
int b = a;  
  
int[] t = {3, -5, 4};  
int[] u = t;
```



Razlaga motivacijskega primera

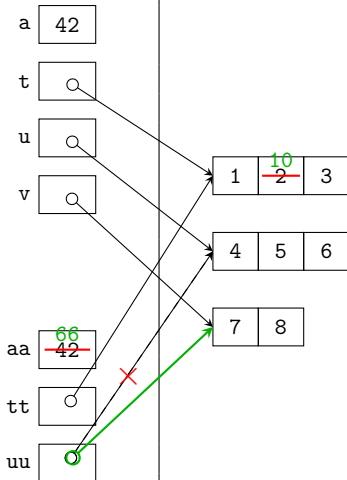
```
int[] a = {1, 2, 3};  
int[] b = a;  
b[0] = 42;
```



Prenašanje parametrov = kopiranje vrednosti

```
public static void main(String[] args) {  
    int a = 42;  
    int[] t = {1, 2, 3};  
    int[] u = {4, 5, 6};  
    int[] v = f(a, t, u);  
    // a: 42  
    // t: [1, 10, 3]  
    // u: [4, 5, 6]  
    // v: [7, 8]  
}
```

```
public static int[] f(  
    int aa, int[] tt, int[] uu) {  
    aa = 66;  
    tt[1] = 10;  
    uu = new int[]{7, 8};  
    return uu;  
}
```



Dvodimenzionalne tabele

- v javi ne obstajajo!
- obstajajo samo **tabele kazalcev na tabele**
- »dvodimenzionalne tabele«: običajno, a netočno poimenovanje

Izdelava dvodimenzionalne tabele

- izdelava tabele z elementi tipa T , ki ima m vrstic in n stolpcev
 - `T[] [] t = new T[m][n];`
 - vsi elementi imajo privzete vrednosti (npr. 0 za $T = \text{int}$)
- izdelava tabele z določenimi elementi
 - `int[] [] t = {
 {e00, e01, ..., e0,n-1},
 {e10, e11, ..., e1,n-1},
 ...
 {em-1,0, em-1,1, ..., em-1,n-1}
};`

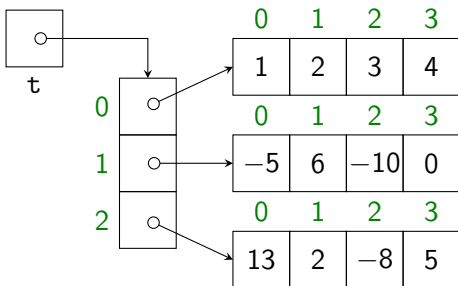
Zgradba dvodimenzionalne tabele

```
int[] [] t = {  
    {1, 2, 3, 4},  
    {-5, 6, -10, 0},  
    {13, 2, -8, 5}  
};
```

konceptualni vidik

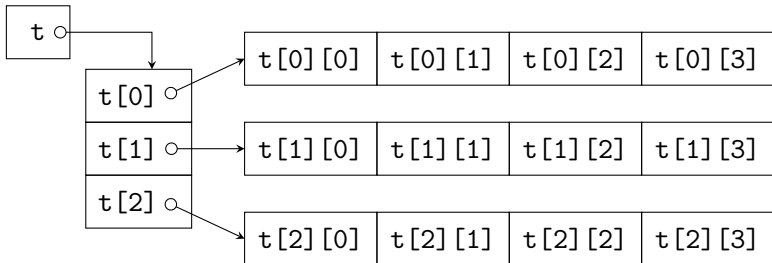
	0	1	2	3
0	1	2	3	4
1	-5	6	-10	0
2	13	2	-8	5

realnejša slika



Dostop do elementov tabele

- $t[i][j]$
 - element z indeksom j v vrstici z indeksom i
 - »element v vrstici i in stolpcu j « (stolpci ne obstajajo!)
 - »element (i, j) «
- $t[i]$
 - kazalec na vrstico z indeksom i



Nepravokotne tabele

- vrstice so lahko različno dolge

```
int[] [] t = {  
    {1, 2, 3},  
    {},  
    {4, 5, 6, 7, 8}  
};
```

- lahko najprej izdelamo samo tabelo kazalcev na vrstice, nato pa še posamezne vrstice

```
int[] [] t = new int[10][];  
for (int i = 0; i < 10; i++) {  
    t[i] = new int[i + 1];  
}
```

Nepravokotne tabele

- stavek

```
int[] [] t = new int[n] [];
```

izdela tabelo n kazalcev

- vsi kazalci imajo privzeto vrednost
- ta vrednost je `null`
 - neobstoječ pomnilniški naslov
 - kazalec, ki ne kaže nikamor

Sprehod po tabeli (z indeksi)

- `t.length`: število vrstic tabele
- `t[i].length`: dolžina vrstice z indeksom `i`

```
for (int i = 0; i < t.length; i++) {  
    for (int j = 0; j < t[i].length; j++) {  
        System.out.print(t[i][j] + " ");  
    }  
    System.out.println();  
}
```


Sprehod po tabeli (z zanko *for-each*)

- elementi tabele kazalcev so tipa $T[]$
- elementi posamezne vrstice so tipa T

```
for (int[] vrstica: t) {  
    for (int element: vrstica) {  
        System.out.print(element + " ");  
    }  
    System.out.println();  
}
```

Vsota po vrsticah

- `public static int[] vsotaPoVrsticah(int[] [] t)`
 - vrne tabelo s toliko elementi, kot je vrstic tabele `t`
 - i -ti element izhodne tabele vsebuje vsoto i -te vrstice tabele `t`

	0	1	2	3	4	
0	2	3	-1	-4	0	→ 0
1	7	-2	-4	3	8	→ 12
2	-6	9	-3	5	-7	→ -2
3	3	-1	-7	6	4	→ 5

Vsota po vrsticah

```
public static int[] vsotaPoVrsticah(int[][] t) {  
    int[] rezultat = new int[t.length];  
    for (int i = 0; i < t.length; i++) {  
        // izračunaj vsoto vrstice z indeksom i (tabele t[i]) ...  
        int vsota = 0;  
        for (int j = 0; j < t[i].length; j++) {  
            vsota += t[i][j];  
        }  
        // ... in jo shrani v rezultat[i]  
        rezultat[i] = vsota;  
    }  
    return rezultat;  
}
```

Indeks maksimuma po stolpcih

- predpostavimo, da imajo vse vrstice enako število elementov
- stolpec z indeksom i
 - elementi $t[0][i]$, $t[1][i]$, ..., $t[t.length - 1][i]$
- `public static int[] imaxPoStolpcih(int[] [] t)`
 - vrne tabelo s toliko elementi, kot je stolpcev tabele t
 - i -ti element izhodne tabele vsebuje indeks maksimuma i -tega stolpca tabele t

Indeks maksimuma po stolpcih

	0	1	2	3	4
0	2	3	-1	-4	0
1	7	-2	-4	3	8
2	-6	9	-3	5	-7
3	3	-1	-7	6	4

↓	↓	↓	↓	↓
1	2	0	3	1

Indeks maksimuma po stolpcih

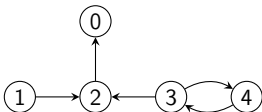
```
public static int[] imaxPoStolpcih(int[][] t) {  
    int stStolpcev = t[0].length;    // upoštevamo predpostavko  
    int[] rezultat = new int[stStolpcev];  
  
    for (int j = 0; j < stStolpcev; j++) {  
        // izračunaj indeks maksimuma v stolpcu z indeksom j ...  
        int iMax = 0;  
        for (int i = 1; i < t.length; i++) {  
            if (t[i][j] > t[iMax][j]) {  
                iMax = i;  
            }  
        }  
        // ... in ga shrani v rezultat[j]  
        rezultat[j] = iMax;  
    }  
    return rezultat;  
}
```

Dosegljivost vozlišč v grafu

- vhod
 - število vozlišč (n)
 - množica usmerjenih povezav $i \rightarrow j$ ($i, j \in [0, n - 1]$)
- izhod
 - množica parov (i, j) , tako da je iz vozlišča i mogoče doseči vozlišče j
- problem tranzitivne ovojnice

Predstavitev podatkov

- množico povezav lahko predstavimo s tabelo graf tipa `boolean[] []` in velikosti $n \times n$



	0	1	2	3	4
0	F	F	F	F	F
1	F	F	T	F	F
2	T	F	F	F	F
3	F	F	T	F	T
4	F	F	F	T	F

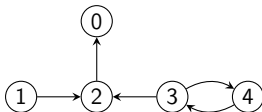
→

	0	1	2	3	4
0	F	F	F	F	F
1	T	F	T	F	F
2	T	F	F	F	F
3	T	F	T	T	T
4	T	F	T	T	T

Postopek

- inicializacija
 - tabelo graf skopiramo v tabelo dosegljivost
- glavni postopek
 - zanka, v kateri posodabljam tabelo dosegljivost
 - inicializacija upošteva samo sosedo vozlišč
 - prvi obhod zanke upošteva tudi sosedo sosedov
 - drugi obhod upošteva tudi sosedo sosedov sosedov
 - ...
 - $(n - 1)$ -vi obhod upošteva celoten graf

Postopek



inicializacija

	0	1	2	3	4
0	F	F	F	F	F
1	F	F	T	F	F
2	T	F	F	F	F
3	F	F	T	F	T
4	F	F	F	T	F

po 1. obhodu

	0	1	2	3	4
0	F	F	F	F	F
1	T	F	T	F	F
2	T	F	F	F	F
3	T	F	T	T	T
4	F	F	T	T	T

po 2. obhodu

	0	1	2	3	4
0	F	F	F	F	F
1	T	F	T	F	F
2	T	F	F	F	F
3	T	F	T	T	T
4	T	F	T	T	T

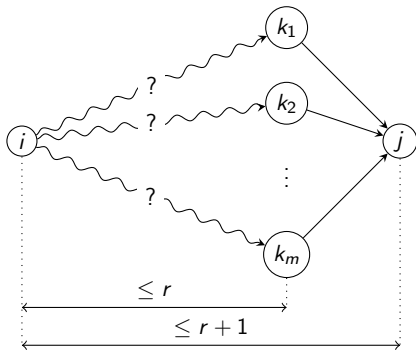
po 3. in 4. obhodu

	0	1	2	3	4
0	F	F	F	F	F
1	T	F	T	F	F
2	T	F	F	F	F
3	T	F	T	T	T
4	T	F	T	T	T

Posodabljanje tabele dosegljivost

- v r -tem obhodu glavne zanke
 - na podlagi tabele dosegljivost, v kateri so zajete razdalje do vključno r , zgradimo tabelo novaDosegljivost, v kateri so zajete razdalje do vključno $r + 1$
 - $\text{dosegljivost}[i][j] == \text{true} \iff$
od i do j je mogoče priti v največ r korakih
 - $\text{novaDosegljivost}[i][j] == \text{true} \iff$
od i do j je mogoče priti v največ $r + 1$ korakih

Posodabljanje tabele dosegljivost



```
novaDosegljivost[i][j] = dosegljivost[i][j] ||  
    dosegljivost[i][k1] && graf[k1][j] ||  
    dosegljivost[i][k2] && graf[k2][j] ||  
    ...  
    dosegljivost[i][km] && graf[km][j]
```

Implementacija

```
// Izdela in vrne matriko dosegljivosti za podani graf.

public static boolean[][] izracunaj(boolean[][] graf) {
    int stVozlisc = graf.length;

    // inicializacija
    boolean[][] dosegljivost = new boolean[stVozlisc][stVozlisc];
    for (int i = 0; i < stVozlisc; i++) {
        for (int j = 0; j < stVozlisc; j++) {
            dosegljivost[i][j] = graf[i][j];
        }
    }

    // glavna zanka
    for (int r = 1; r < stVozlisc; r++) {
        dosegljivost = posodobi(dosegljivost, graf);
    }
    return dosegljivost;
}
```

Implementacija

```
// Na podlagi matrike dosegljivosti, ki zajema razdalje do  $r$ ,  
// izdela in vrne matriko dosegljivosti, ki zajema razdalje do  $r + 1$ .  
  
public static boolean[][] posodobi(boolean[][] dosegljivost,  
                                   boolean[][] graf) {  
    int stVozlisc = graf.length;  
    boolean[][] novaDosegljivost = new boolean[stVozlisc][stVozlisc];  
    for (int i = 0; i < stVozlisc; i++) {  
        for (int j = 0; j < stVozlisc; j++) {  
            novaDosegljivost[i][j] = obstaja(dosegljivost, graf, i, j);  
        }  
    }  
    return novaDosegljivost;  
}
```

Implementacija

```
// Vrne true natanko tedaj, ko lahko na podlagi doslej zbranih
// podatkov zaključimo, da je vozlišče j dosegljivo iz vozlišča i.

public static boolean obstaja(boolean[] [] dosegljivost,
                             boolean[] [] graf, int i, int j) {
    if (dosegljivost[i][j]) {
        // smo že ugotovili, da je vozlišče j dosegljivo iz vozlišča i
        return true;
    }

    // preveri, ali obstaja vozlišče k, ki je dosegljivo iz vozlišča i
    // in hkrati povezano z vozliščem k
    int stVozlisc = graf.length;
    for (int k = 0; k < stVozlisc; k++) {
        if (dosegljivost[i][k] && graf[k][j]) {
            return true;
        }
    }
    return false;
}
```