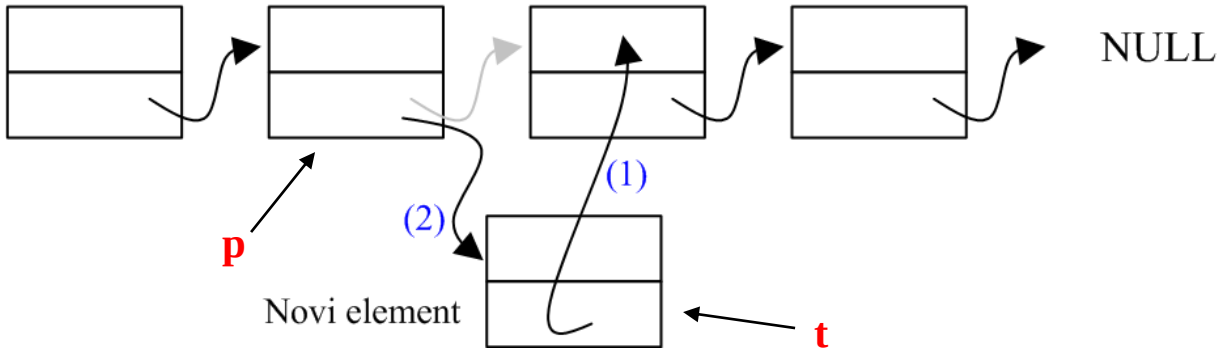
Programiranje I

Liste – Nastavak
Grafovi

Dodavanje elementa u sredinu liste

- Prvo grafički ilustrujmo što se dešava u ovom slučaju:



- Očigledno je da treba odrediti poziciju (član liste na koji pokazuje pokazivač p) iza koje treba ubaciti novi element, na koji pokazuje pokazivač t.
- Zatim, pokazivač next iz elementa na koji pokazuje t treba usmjeriti na element koji se nalazi nakon člana liste na koji pokazuje p (korak (1)).
- Konačno, pokazivač next iz elementa na koji pokazuje p treba preusmjeriti da pokaže na novododati element t (korak (2)).

Dodavanje elementa u sredinu liste

- Dio koda:
  t->next=p->next;
  p->next=t;
- Naravno, potrebno je alocirati element na koji pokazuje t, a da bi sve opcije zajedno lijepo uvježbali pokušajte (bez gledanja u zbirku) da napišete funkciju koja radi sljedeće:

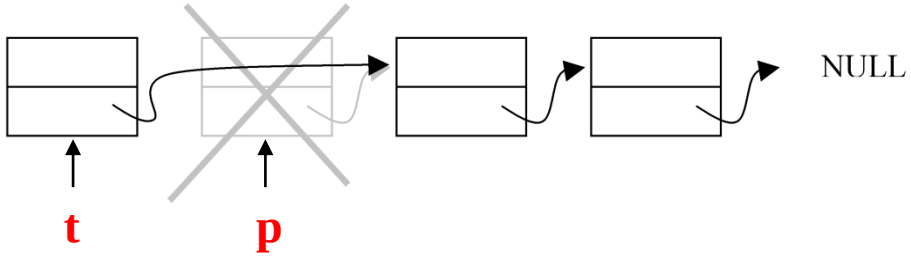
  Funkciji se predaje pokazivač na glavu sortirane liste. Elementi (cijeli brojevi) upisani u čvorovima liste su sortirani u rastući redosljed. Drugi argument funkcije je cijeli broj. Treba ga smjestiti u novi član liste koji će biti postavljen tako da se ne naruši sortiranost liste. Funkcija treba da vrati pokazivač na glavu liste.

Komentari o datom problemu

- Prva varijanta, koju vjerovatno ne očekujete, je da je predati element manji od onoga koji je upisan u glavu liste. U tom slučaju treba alocirati novi element, upisati u njega cijeli broj koji je argument funkcije i njegov pokazivač **next** treba usmjeriti na glavu liste. Tada ovaj novi element postaje glava liste i funkcija treba da vrati pokazivač na njega.
- Ako prva varijanta nije zadovoljena možemo da krenemo u rekurzivnu proceduru traženja prave pozicije za novi element liste. Sa srećom!

Brisanje elementa iz sredine liste

- Grafički se ovaj problem može označiti na sljedeći način:



Procedura je relativno jednostavna. Treba pronaći element liste koji se briše. To se obavlja sa pozicije prethodnog elementa, na koji pokazuje pokazivač **t**. Znači, vrši se neka provjera stanja **t->next** pokazivača. Zatim se pokazivač **t->next** smjesti privremeno u pokazivač **p= t->next**, a pokazivač **t** se preusmjeri da pokaže na element koji je nakon **t->next**, tj. **t->next=t->next->next**. Na kraju treba dealocirati strukturu na koju pokazuje **p**: **free(p)**. Na ovaj način je "premošten" element koji se briše i da bi se on obrisao moramo uvesti pomoćni pokazivač na njega, a to je **p**.

Zadatak za vježbu

- Kreirati funkciju kojoj se proslijeđuje glava liste i cijeli broj upisan u elementu liste kojeg treba obrisati. Lista je sortirana u neopadajući redosljed. Funkcija treba da pored brisanja elementa vrati glavu liste. U slučaju da traženi element ne postoji u listi ne treba vršiti brisanje. Treba predvidjeti i situaciju da je glava liste traženi element, kao i situaciju da je glava liste ujedno i rep liste i da je to traženi element (kad je lista prazna i nakon operacije brisanja treba jednostavno vratiti NULL).

Brojanje elemenata liste

- Da bi ilustrovali rad sa rekurzijom primjenjen na liste, uradićemo dvije funkcije. Prva funkcija vrši prosto brojanje elemenata liste. Radi olakšice, pretpostavimo da lista ima barem jedan element. Ako je taj element ujedno i rep liste funkcija treba da vrati **1**, a ako nije treba da vrati 1+koliko_ima_elemenata_u ostatku_liste, a ostatak liste je opet lista koja počinje od narednog elementa.

```
int brojElemente(struct lista *glava){
   if(glava->next==NULL)
        return 1;
   else
        return 1+brojElemente(glava->next);}
```

- **Napomena**: Rekurzivna realizacija funkcije nije efikasna zbog velikog broj poziva funkcije. Iterativna realizacija je efikasnija.

Nadovezivanje listi

- Nadovezivanje (konkatenacija) listi je sljedeći problem kojeg želimo da ilustrujemo. Imamo dvije liste i na kraj prve želimo da nadovežemo drugu. To znači da rep prve liste treba da pokaže na glavu druge liste. Očigledno, nakon što odredimo rep prve liste, njegov pokazivač **next** treba preusmjeriti na glavu druge liste:

```
struct lista *nadovezi(struct lista *gl1,struct lista *gl2){
    struct lista *pom=gl1;
    while(pom->next!=NULL)
        pom=pom->next;
    pom->next=gl2;
    return gl1;
}
```

Uvodimo pomoćni pokazivač za kretanje kroz prvu listu.

Idemo na kraj prve liste.

Rep prve liste ukaži na glavu druge liste.

Vraćamo glavu novonastale liste.

Specijalni tipovi lista

- Prvi specijalni tip liste se naziva **stek** (podsjetite se gdje smo prvi put uveli ovaj pojam). Stek je specijalni tip liste kod kojeg se dodavanje novih elemenata i brisanje postojećih izvodi samo sa pozicije repa.
- Stek je kao takav FILO (First-In-Last-Out) memorija, odnosno član liste koji prvi uđe u listu posljednji iz nje izlazi, i obrnuto: posljednji uđe, prvi izađe.
- Operacije kod steka su:
  - **push** (dodavanje elementa),
  - **pop** (uklanjanje posljednje dodatog elementa) i
  - **peek** (čitanje posljednje dodatog elementa, bez modifikacije steka).
- Implementiraju se i operacije provjere da li je stek pun ili prazan.
- Osim preko listi, stekovi mogu realizovati i pomoću nizova.

Specijalni tipovi lista

- Drugi specijalni tip liste je **red** (eng. queue). Kod reda se podaci dodaju na rep liste, a brišu sa glave.
- Kao takav, red je FIFO (First-In-First-Out) memorija, odnosno elementi koji prvi ulaze u listu iz nje se kao prvi i brišu, i obrnuto: oni koji posljednji uđu, posljednji izađu.
- Tri osnovne operacije kod reda su:
  - **enqueue** (dodavanje elementa na početak reda),
  - **dequeue** (uklanjanje elemenata sa kraja reda) i
  - **peek** (čitanje elementa sa kraja reda, bez njegovog uklanjanja).
- Ovdje takođe možemo implementirati operacije provjere da li je red pun ili prazan.
- Rad sa stekom i redom je jednostavniji nego rad sa jednostruko povezanom listom koja dozvoljava da se upis i brisanje vrše na svim pozicijama.

Dvostruko povezana lista

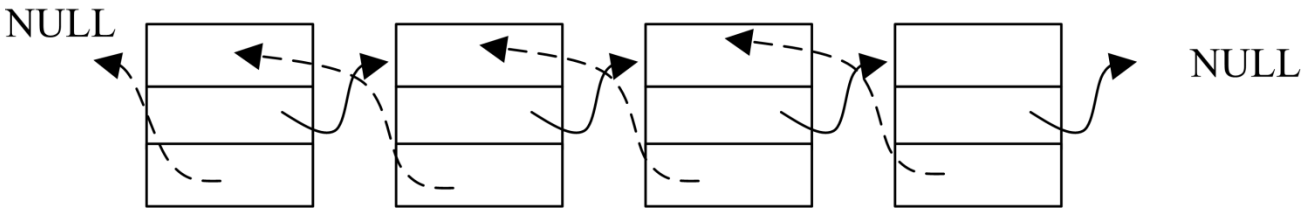
- Kvalitativno drugačiji tip liste je dvostruko povezana lista. Kod ove liste čvorovi pamte, pored narednog, i prethodni čvor.
- Dakle, dvostruko povezana lista se može deklarisati kao:

```
struct lista2pov {
    ... /*podaci koji čine listu*/
    struct lista2pov *next;
    struct lista2pov *prev;
};
```

Pokazivači na naredni i prethodni čvor liste.

Vizuelizacija dvostruko povezane liste

- Uobičajena grafička predstava dvostruko povezane liste je:

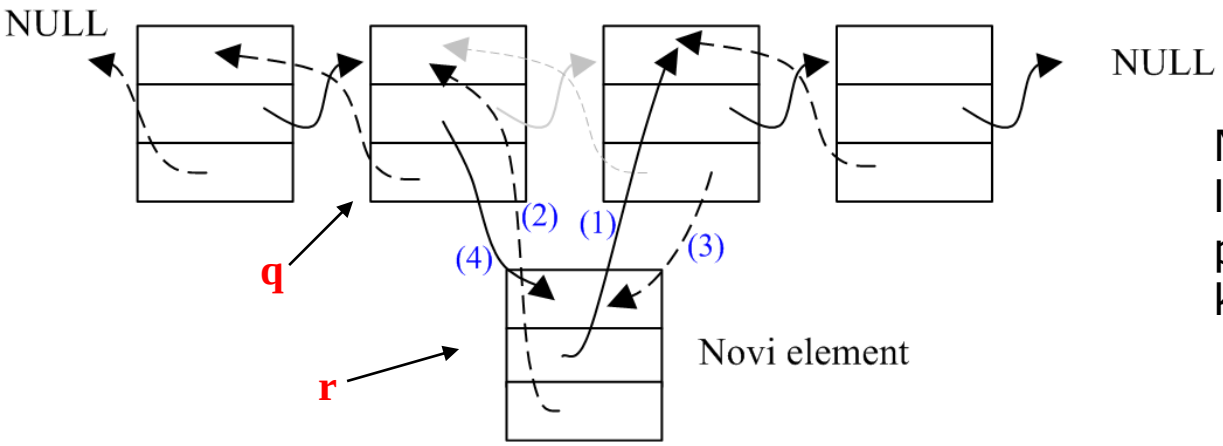


Punom linijom su označeni pokazivači na naredne elemente u listi, dok su isprekidanom označeni pokazivači na prethodne elemente. Ovaj tip liste ima dodatnu funkcionalnost, odnosno dozvoljava kretanje i unazad i unaprijed po listi.

Međutim, rad sa ovakvim tipom liste zahtjeva više pažnje, jer se mora voditi računa o dva pokazivača.

Dodavanje elementa u dvost. listu

- Grafička vizuelizacija dodavanja elementa u dvostruko povezanu listu:



Neka je r pokazivač na element liste koji se dodaje, dok je q pokazivač na element liste iza kojeg se novi element dodaje.

```
r->next=q->next;     (korak (1))
r->prev=q;           (korak (2))
q->next->prev=r;     (korak (3))
q->next=r;           (korak (4))
```

Jednostavno pravilo kojega se treba držati je da prvo novi element uspostavi veze sa ostalim čvorovima u listi, pa da se tek onda prekidaju i preusmjeravaju stare veze.

Dvostruke liste – Za vježbu

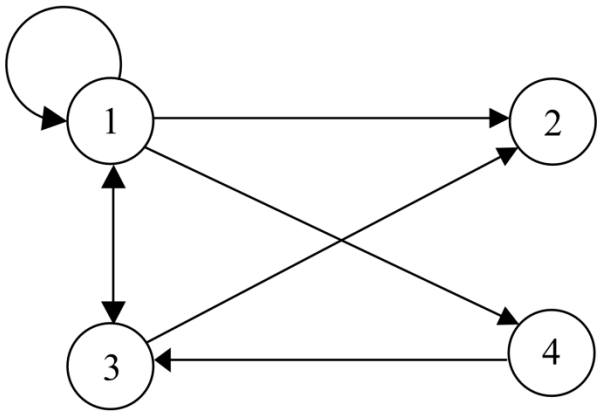
- Kod dvostruko povezane liste uopšte nije potrebno pamtiti glavu jer se možemo po listi kretati u oba pravca, i ka kraju i ka početku.
- U zbirci je dat zadatak kod kojeg se element dodaje u dvostruko povezanu listu, a funkciji se prosljeđuje glava liste.
- Prepravite dati zadatak tako da se funkciji može proslijediti pokazivač na bilo koji čvor u listi.
- Napisati i funkciju koja briše čvor iz sortirane dvostruko povezane liste, a da se funkciji prosljeđuje pokazivač na proizvoljni čvor liste.

Grafovi

- Liste su izuzetno napredni podaci. Pored pomenutih postoje i drugi tipovi lista koji, iz razloga složenosti i vremena kojeg u kursu imamo, neće biti razrađivani.
- Graf je još jedan veoma napredan tip podataka.
- Graf je skup čvorova u kojem svaki čvor može da pokaže na proizvoljan broj drugih čvorova.
- Precizna matematička definicija grafa je:
  - Graf **G=(V,E)** je uređeni par koji se sastoji od konačnog nepraznog skupa čvorova **V** i skupa ivica **E**. Ako je ivica uređeni par čvorova **(v,w)** iz skupa **V** gdje je **v** početak, a **w** kraj ivice tada govorimo o **usmjerenom grafu**, dok je u suprotnom riječ o **neusmjerenom grafu**.

Vizuelizacija grafa

- Standardna metodologija za vizuelizaciju grafa je data na slici.



Predmetni graf ima četiri čvora. Ovo je usmjeren graf. Kod usmjerenog grafa koriste se strelice da prikažu početni i krajnji čvor ivice (krajnji čvor je onaj na koji pokazuje strelica).

Kod usmjerenog grafa dozvoljena je ivica **(v,v)**, dok kod neusmjerenog nije dozvoljeno da isti čvor bude i početak i kraj ivice.

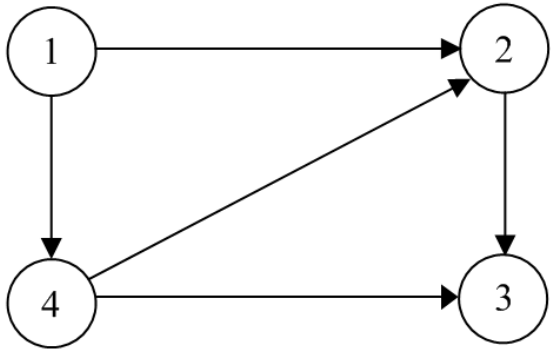
Graf - Definicije

- Ako postoji ivica **(v,w)** u grafu kaže se da je čvor **w** susjed čvora **v** (ovo ne implicira da je **v** susjed čvora **w**).
- Broj čvorova koji su susjedi čvora **v** naziva se izlazni stepen čvora.
- Skup ivica $(v_1,v_2),(v_2,v_3),\ldots,(v_{n-1},v_n)$ naziva se put od čvora v_1 do čvora v_n.
- Dati put je dužine **n-1**.
- Jednostavna putanja je ona kod koje se nijedan čvor osim prvog i posljednjeg ne ponavlja (početak i kraj jednostavne putanje može biti u istom čvoru, ali ne mora).
- Ciklus je jednostavna putanja dužine makar **3** koja počinje i završava se u istom čvoru.

Predstavljanje grafa

- Postavlja se pitanje kako se grafovi memorijski reprezentuju. Čvorovi grafa predstavljaju neke podatke strukturnog tipa.
- Postoje tri načina da se memorišu veze između čvorova.
- Prvi način je preko **matrice susjedstva**. Ako je N_V broj čvorova u grafu matrica, susjedstva je matrica dimenzija $N_V \times N_V$, sa vrijednošću **1** upisanom na poziciji **[I][J]** ako je čvor **J** susjed čvoru **I**. Ako čvor **J** nije susjed čvoru **I** onda se na odgovarajuće mjesto u matrici susjedstva upisuje **0**.

Matrica susjedstva



Graf

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

Matrica susjedstva

- Dakle, pored predstave čvora dodaje se veoma jednostavna matrica susjedstva. Na osnovu matrice susjedstva se lako može provjeriti da li je u pitanju usmjereni ili neusmjereni graf. Kako? Takođe, veoma jednostavno se određuje izlazni stepen svakog čvora. Kako? I mnoštvo drugih informacija o grafu ili obrada grafa se može izvršiti na veoma jednostavan način preko matrice susjedstva.

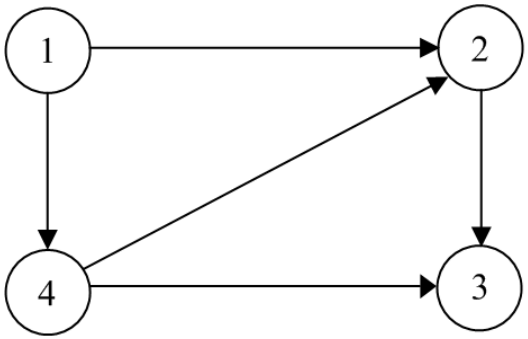
Mana matrice susjedstva

- Osnovni nedostatak matrice susjedstva je memorijska zahtjevnost. Naime, u grafu može postojati veoma veliki broj čvorova sa veoma malim brojem veza između njih.
- Recimo, tipična situacija je **N_V=1000**. Tada matrica susjedstva ima milion elemenata, a može da se dogodi da je samo nekoliko hiljada nenultih (samo nekoliko hiljada ivica).
- Postoje neke strategije koje omogućavaju da se i dalje koristi matrica susjedstva:
  - prva strategija je preko polja bitova, kada se pamćenje jedne jedinice ili nule vrši preko 1 bita, a ne recimo 2 bajta kao za cijeli broj.
  - alternativa je da se umjesto matrice pamte uređeni parovi koji predstavljaju poziciju nenultih elemenata (za graf sa prethodnog slajda bi to bilo: (1,2), (1,4), (2,3), (4,2) i (4,3)). Ovakve matrice se, inače, nazivaju **rijetkim** (u engl. terminologiji **sparse**).

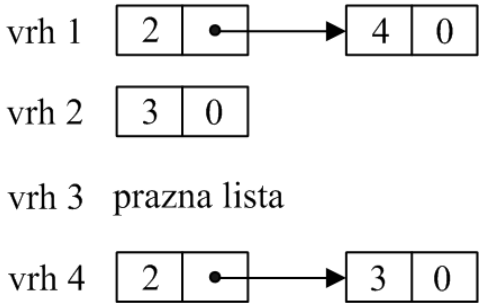
Alternativni načini predstavljanja grafa

- Ako se ne koristi matrica susjedstva u osnovnoj formi onda se gubi osnovna prednost ovog načina predstavljanja, a to je jednostavnost.
- Stoga su uvedeni drugi načini za memorisanje veza koje postoje između čvorova u grafu.
- Jedan od načina je preko **listi susjedstva**.
- Za svaki čvor grafa formira se lista čiji su elementi njemu susjedni čvorovi.

Liste susjedstva - Primjer



Graf

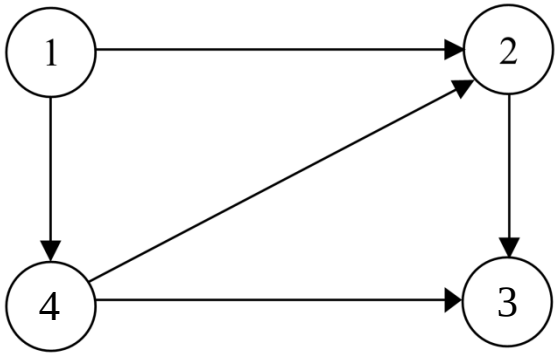


Lista susjedstva

- Iz lista susjedstva lako zaključujemo da čvor (vrh) **1** ima susjedne čvorove **2** i **4**, dok čvor **3** nema nijedan susjedni čvor.
- Ukupan broj elemenata u listama susjedstva je jednak N_V+N_E, gdje je N_E broj ivica u grafu.
- Ovo je, očigledno, mnogo efikasnije za memorisanje kod grafova sa relativno velikim brojem čvorova i relativno malim brojem ivica.

Predstavljanje grafa preko nizova

- Popularan način (mada možda u početku zbunjujući) je preko dva niza. Ti nizovi se nazivaju **END** i **NEXT** niz.
- Objasnimo na primjeru već uvedenog grafa:



Graf

		END	NEXT
vrhovi	1		5
	2		7
	3		0
	4		8
ivice	5	2	6
	6	4	0
	7	3	0
	8	2	9
	9	3	0

Nizovi END i NEXT

Predstavljanje grafa preko nizova

- Prvih N_V upisa se odnosi na čvorove grafa, redom od prvog do čvora N_V.
- Narednih N_E upisa se odnosi na ivice grafa.
- Prvih N_V upisa u nizu **END** su prazni.
- U nizu **NEXT** za čvor 1 upisana je vrijednost 5. To znači da se u petom redu nalaze podaci o prvoj ivici iz ovog čvora. Vidimo da je to ivica do čvora 2, a također vidimo i da se u redu 6 nalaze podaci o narednoj ivici koja kreće iz čvora 1.
- U redu 6 piše da je naredna ivica do čvora 4, ali i da nema više ivica iz čvora 1, što je određeno nulom u šestom redu niza **NEXT**.
- Tumačite sami ostala tri reda nizova **END** i **NEXT**.

Za vježbu

- Svi zadaci u zbirci zadataka su urađeni za najjednostavniji slučaj - matrice susjedstva.
- Za vježbu uradite neke od primjera za slučaj grafa reprezentovanog preko lista susjedstva i preko nizova.

Problem najkraćeg puta

- Grafovi su izuzetno upotrebljivi u brojnim primjenama:
  - algoritmi optimizacije;
  - planiranje saobraćaja;
  - planiranje poslovanja;
  - električna kola, itd.
- Obično treba dobro poznavati programiranje, problem koji treba riješiti i matematičku teoriju grafova. To znanje se brzo isplati izuzetno jednostavnim rješenjima.
- Da bi ilustrovali snagu grafova posmatraćemo poznati problem najkraćeg puta.

Problem najkraćeg puta

- Pretpostavimo da u grafu imamo N čvorova. Pretpostavimo da poznajemo cijene (težine) ivica između susjednih čvorova. Cijena je nenegativna veličina dodijeljena ivici. U slučaju da nema ivice između dva čvora cijena je beskonačna. Matrica sa cijenama ivica podsjeća na matricu susjedstva, ali umjesto binarnih vrijednosti upisane su vrijednosti u intervalu $[0,\infty)$.
- Problem je odrediti najjeftiniju putanju između svih čvorova I i J u grafu koja može prolaziti preko proizvoljno mnogo ivica.

Problem najkraćeg puta

- Problem najkraćeg puta se može riješiti kroz sljedeće algoritamske korake.
  - **Korak 1.** Zada se broj čvorova **N**.
  - **Korak 2.** Zada se cijena puta, tj. ivica, između svih čvorova: **l[i][j]** za $i \in [0,N)$ i $j \in [0,N)$ **(uglasta zagrada označava uključenu granicu, a obična granicu koja nije uključena).**
  - **Korak 3.** Postavi se početna iteracija: **cs[i][j]=l[i][j]** za **$i \neq j$** i **cs[i][j]=0** za **i=j**, jer je najjeftinije ne kretati se iz jednog čvora u samog sebe.
  - **Korak 4.** Ciklus po **k** u granicama **[0,N)** u kojem se računa:

cn[i][j]=min{cs[i][j],cs[i][k]+cs[k][j]}

Ključni korak algoritma!

Problem najkraćeg puta

- **Korak 5.** Ažuriranje stare vrijednosti matrice težina puta **cs[i][j]=cn[i][j]** čime se ciklus priprema za novu iteraciju. Povratak na **korak 4**.

- Što predstavlja ključni korak algoritma:
  cn[i][j]=min{cs[i][j],cs[i][k]+cs[k][j]} ?

- Ovaj korak bira jeftiniju od dvije moguće putanje od čvora i do čvora j:
  > direktnu od i ka j, ili
  > indirektnu od i ka j preko k (od i ka k, pa od k ka j).

- Nakon prve iteracije u petlji po k (k=0) matrica cn sadrži najjeftinije putanje između bilo koja dva čvora i i j koje su se mogle napraviti preko čvora 0, nakon toga za k=1 dobijamo matricu najjeftinijih putanja između bilo koja dva čvora koja se mogla napraviti preko čvorova 0 i 1.

Dijkstra algoritam

- Nakon posljednjeg koraka u algoritmu dobijamo najkraći put između svih čvorova u grafu `i` i `j`, a preko svih čvorova u grafu od `0` do `N-1`.
- Nestrpljivi studenti mogu da pokušaju da realizuju predmetni algoritam sami **(bez gledanja u materijale)**, dok oni strpljivi **(ovdje strpljivost nije vrlina)** mogu da sačekaju naredno predavanje.
- Predmetni algoritam se naziva **Dijkstra algoritam** **(mada je to dosta nekorektno prema drugim ljudima koji su učestvovali u njegovom razvoju).**