

## 1. UVOD

### 1.1 Osnovni pojmovi

U računarstvu se susrećemo s dva osnovna pojma:

strukture podataka . . . “statički” aspekt nekog programa. Ono sa čime se radi.

algoritmi . . . “dinamički” aspekt programa. Ono šta se radi.

Strukture podataka i algoritmi su u neraskidivoj vezi: ne može se govoriti o jednom, a da se ne spomene drugo.

U okviru ovog predmeta proučavat ćemo baš tu vezu: posmatrat ćemo kako odabrana struktura podataka utječe na algoritme za rad s njom, te kako odabrani algoritam sugerira pogodnu strukturu za prikaz svojih podataka. Na taj način upoznat ćemo se s nizom važnih ideja i pojmova, koji čine osnove računarstva.

Uz “strukture podataka” i “algoritme”, koristit ćemo još nekoliko naizgled sličnih pojmova.

Objasnit ćemo njihovo značenje i međusobni odnos.

**Tip podataka** . . . skup vrijednosti koje neki podatak može poprimiti (npr. podatak tipa int može imati samo vrijednosti iz skupa cijelih brojeva koje se mogu predstaviti u računaru).

**Apstraktni tip podataka (a.t.p.)** . . . zadaje se navođenjem jednog ili više tipova podataka, te jedne ili više operacija (funkcija). Operandi i rezultati navedenih operacija su podaci navedenih tipova.

**Struktura podataka** . . . skupina varijabli u nekom programu i veza među tim varijablama.

**Algoritam** . . . konačni niz instrukcija, od kojih svaka ima jasno značenje i može biti izvršena u konačnom vremenu. Iste instrukcije mogu se izvršiti više puta, pod pretpostavkom da same instrukcije ukazuju na ponavljanje. Ipak, zahtijevamo da, za bilo koje vrijednosti ulaznih podataka, algoritam završava nakon konačnog broja ponavljanja.

**Implementacija apstraktnog tipa podataka** . . . konkretna realizacija dotičnog apstraktnog tipa podataka u nekom programu. Sastoji se od definicije za strukturu podataka (kojom se prikazuju podaci iz apstraktnog tipa podataka) te od potprograma (kojima se operacije iz apstraktnog tipa podataka ostvaruju pomoću odabranih algoritama). Za isti apstraktni tip podataka obično se može smisliti više različitih implementacija - one se razlikuju po tome što koriste različite strukture za prikaz podataka, te različite algoritme za izvršavanje operacija.

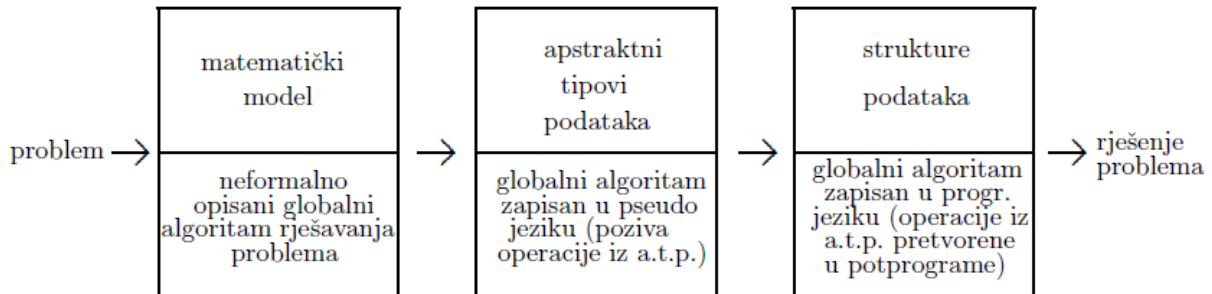
Kroz predmet ćemo se upoznati sa raznim apstraktnim tipovima podataka. Posmatrat će se razne implementacije za taj apstraktni tip podatka, te njihove prednosti i mane. Na taj način upoznat ćemo mnogo apstraktnih tipova podataka, te još više struktura podataka i algoritama. Također ćemo se upoznati sa općenitim metodama (strategijama) koje mogu služiti za oblikovanje složenijih algoritama. Znanje iz ovog predmeta trebalo bi omogućiti da

se bolje programira. Naime, razvoj programa (metodom postepenog profinjavanja) može se promatrati u skladu sa dijagramom na slici ispod.

Inače, profinjavanje (engl. refinement) ili pročišćavanje (usavršavanje) je proces razvoja programa odozgo-naniže. Nerazrađeni koraci se tokom profinjavanja sve više preciziraju dok se na samom kraju ne dođe do sasvim preciznog opisa u obliku funkcionalnog programskog koda. U svakom koraku jedan zadatak razlaže se na sitnije zadatke.

Na primer, zadatak obradi\_podatke\_iz\_datoteke() može da se razloži na otvori\_datoteku(), procitaj\_podatke(), obradi\_podatke(), zapiši\_podatke(), zatvori\_datoteku(), itd.

Apstrahovanje i profinjavanje međusobno su suprotni procesi.



Slika 1. Postupak rješavanja problema.

U rješavanju problema postupkom postepenog profinjavanja koraci su:

- prvo postavljamo matematički model i algoritam u neformalnom jeziku koji koristi matematičke pojmove,
- zatim stvaramo apstraktni tip podataka, i algoritam zapisujemo u pseudo-jeziku koji koristi operacije iz ATP, te konačno
- apstraktni tip podataka i algoritam u pseudokodu prevodimo u strukture podataka u stvarnom programskom jeziku i algoritam zapisan u stvarnom programskom jeziku, što nam predstavlja rješenje problema.

Iz ovog dijagrama vidi se uloga apstraktnih tipova podataka, struktura podataka i algoritama u postupku rješavanja problema. Također se vidi da se programiranje u podjednakoj mjeri sastoji od razvijanja struktura podataka kao i od razvijanja algoritama.

## 2. UVOD U ALGORITME

### 2.1. Nastanak riječi algoritam

U svojoj knjizi arapski matematičar Muhamed ibn Musa al Horezmi (9. stoljeće) je opisao pravila za obavljanje aritmetičkih operacija nad brojevima zapisanim u dekadskom sistemu. Original te knjige na arapskom jeziku je izgubljen, ali postoji prijevod na latinski. U latinskom prijevodu ispred svakog pravila piše:

Dixit Algorizmi

što je u prijevodu: Algorizmi je govorio. Zadnji dio imena al Horezmi pretvoren je u Algorizmi. Postepeno 'Algorizmi je govorio' se pretvara u 'algoritam glasi'.

U početku su se pod pojmom algoritma podrazumijevala samo pravila za računanje brojevima. Danas se pod algoritmom podrazumijevaju pravila za obavljanje zadataka u različitim oblastima, a najčešće u računarstvu.

## 2.2. Pojam algoritma

Algoritam je logički niz diskretnih koraka koji opisuju rješenje nekog problema. Pri tome rješenje treba biti izvodljivo u konačnom vremenu.

Algoritmi imaju sljedeće karakteristike:

- obično imaju jedan ili više ulaznih objekata,
- imaju barem jedan izlazni objekat, koji nastaje kao rezultat rada algoritma,
- značenje svih naredbi od kojih se algoritam sastoji treba biti nedvosmisleno,
- algoritam se mora izvesti u konačnom vremenu,
- svaka naredba algoritma mora biti ostvarljiva računarskim instrukcijama u konačnom vremenu. Važno je istaknuti i to da algoritme karakterizira determinističko ponašanje. Pod pojmom determinističkog ponašanja podrazumijevamo da se algoritam pod istim uslovima ponaša uvijek na isti način.

## 2.3. Analiza i složenost algoritma

Obično postoji više algoritama pomoću kojih možemo riješiti neki problem. Zato je važno obrazložiti moguće kriterije po kojima se u konkretnim situacijama biraju najprikladniji algoritmi. Jedan od najvažnijih kriterija je efikasnost algoritma. Pod efikasnošću se obično podrazumijeva stepen potrošnje računarskih resursa potrebnih za izvođenje algoritma. U tu svrhu radimo analizu algoritma.

Pod analizom algoritma podrazumijevamo procjenu vremena izvršavanja (odnosno broj operacija) tog algoritma ili prostora koji će podaci zauzimati.

U nekim situacijama vremenska ili prostorna složenost programa nisu mnogo važne (ako se zadatak izvršava brzo, ne zahtijeva mnogo memorije, ima dovoljno raspoloživog vremena itd), ali u nekim je vrijedna ušteda svakog sekunda ili bajta. I u takvim situacijama dobro je najprije razviti najjednostavniji program koji obavlja dati zadatak, a onda ga modifikovati ako je potrebno da se uklopi u zadata vremenska ili prostorna ograničenja. Vremenska složenost algoritma određuje i njegovu praktičnu upotrebljivost tj. najveće ulazne vrijednosti za koje je moguće da će se algoritam izvršiti u nekom razumnom vremenu. Analogno važi i za prostornu složenost.

Iako kažemo da su najvažniji resursi vrijeme i memorijski prostor, imajući u vidu brzi razvoj tehnologije koja omogućuje sve veću dostupnost memorijskih komponenti po sve nižoj cijeni, onda se nameće zaključak da je u većini slučajeva vrijeme izvođenja algoritma zapravo najvažniji kriterij. Mi ćemo ovdje govoriti o vremenskoj složenosti algoritama.

Vrijeme izvođenja algoritama ovisi o sljedećem:

- konkretnoj hardversko-softverskoj platformi koja se koristi za izvršavanje algoritma,

- ulaznih podataka, te
- vremenske složenosti algoritma koja je karakteristika samog algoritma.

Vrijeme poistovjećujemo s brojem operacija koje odgovarajući program treba obaviti, i izražavamo kao funkciju oblika  $T(n)$ , gdje je  $n$  neka mjera za veličinu skupa ulaznih podataka. Naprimjer, ako promatramo algoritam koji sortira niz realnih brojeva, tada njegovo vrijeme izvršavanja izražavamo u obliku  $T(n)$  gdje je  $n$  dužina niza realnih brojeva. Slično, ako promatramo algoritam za invertovanje matrice, tada njegovo vrijeme izvršavanja izražavamo kao  $T(n)$ , gdje je  $n$  red matrice.

Međutim, često se dešava da  $T(n)$  ne zavisi samo od veličine skupa ulaznih podataka  $n$ , nego i od vrijednosti tih podataka. Npr. algoritam za sortiranje će možda brže sortirati niz brojeva koji je “skoro sortirani”, a sporije niz koji je “veoma izmiješan”. Tada  $T(n)$  definišemo kao vrijeme u najgorem slučaju, dakle kao najduže vrijeme izvršavanja po svim skupovima ulaznih podataka veličine  $n$ . Također se može posmatrati vrijeme u prosječnom slučaju  $T_{avg}(n)$ , koje se dobiva kao matematičko očekivanje vremena izvršavanja. Da bismo mogli govoriti o očekivanju, moramo pretpostaviti distribuciju za razne skupove ulaznih podataka veličine  $n$ . Obično se pretpostavlja uniformna distribucija, dakle smatra se da su svi skupovi ulaznih podataka jednako vjerovatni.

Funkciju  $T(n)$  nema smisla precizno određivati, dovoljno je utvrditi njen red veličine. Npr. vrijeme izvršavanja Gaussovog algoritma za invertovanje matrice je  $O(n^3)$ . Time smo zapravo rekli da  $T(n) \leq \text{const} \cdot n^3$ . Nismo precizno utvrdili koliko će sekundi trajati naš program. Jedino što znamo je sljedeće:

- ako se red matrice udvostruči, invertovanje bi moglo trajati i do 8 puta duže;
- ako se red matrice utrostruči, invertovanje traje 27 puta duže, itd.

Vrijeme izvršavanja programa izmjereno na jednom konkretnom računaru zavisi i od operativnog sistema pod kojim računar radi, od jezika i od kompajlera kojim je napravljen izvršni program za testiranje, itd.

Ukoliko raspolažemo sa procjenom vremena izvršavanja pojedinih operacija na nekom konkretnom računaru, tada se vrijeme izvršavanja pojedinih dijelova kôda može približno tačno procijeniti. Npr., na računaru sa procesorom Intel Core i7-2670QM 2.2GHz koji radi pod operativnim sistemom Linux, operacija sabiranja dvije vrijednosti tipa int troši oko trećinu nanosekunde, a operacija množenja oko jednu nanosekundu. To znači da se za jednu sekundu na tom računaru može izvršiti oko milijardu množenja cijelih brojeva.

Postoje mnogi alati koji omogućuju analizu i unapređenje performansi programa. To su tzv. profajleri (engl. profiler). Oni daju podatke o tome koliko puta je koja funkcija pozvana tokom (nekog konkretnog) izvršavanja, koliko je utrošila vremena i slično. Ukoliko se razvijeni program ne izvršava željeno brzo, potrebno je unaprijediti neke njegove dijelove. Prvi kandidati za izmjenu su upravo oni dijelovi kôda koji troše najviše vremena.

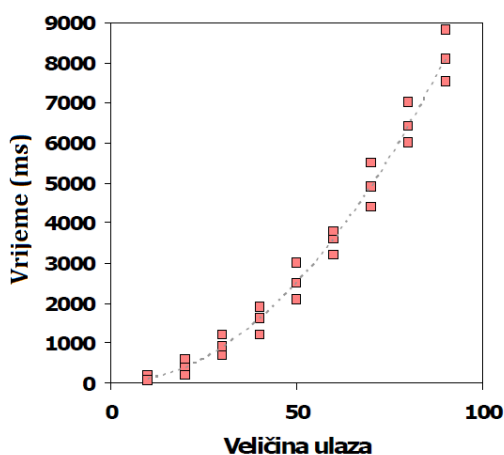
Generalno za analizu složenosti nekog algoritma možemo primijeniti:

- eksperimentalni pristup ili
- teoretski pristup.

Eksperimentalni pristup se sastoji u sljedećem:

- Napišemo program koji implementira algoritam kojeg analiziramo
- Izvršavamo program sa ulazima različite veličine
- Koristimo neke od funkcija, kao što je, na primjer, ugrađena `clock()` funkcija da dobijemo tačno vrijeme izvođenja programa
- Rezultate mjerenja prikazemo grafički.

Za ilustraciju na slici 2 je prikazan jedan mogući dijagram, gdje je za različite vrijednosti veličine ulaza grafički prikazano vrijeme izvođenja. Iz grafikona se može vidjeti da je u ovom primjeru mjerenje izvršeno tri puta za sljedeće veličine ulaza: 10,20, 30,40,50,60,70,80,90 i 100. Nadalje, za svako mjerenje izaberemo srednji rezultat i kroz te srednje vrijednosti povučemo liniju da bi dobili eksperimentalnu krivulju.



Slika 2. Vrijeme izvođenja

### Teoretska analiza

U teoretskoj analizi hardversko-sofversko okruženje u kojemu je implementiran neki algoritam u osnovi nije povezano sa samom suštinom algoritma, pa se za utvrđivanje vremenske složenosti i ne uzima u obzir. Naime, vrijeme izvršavanja algoritma možemo procjenjivati tako da ne promatramo stvarno vrijeme izvršavanja, nego uvodimo u razmatranje apstraktne elementarne operacije pri čemu svakoj pridružujemo neko jedinično vrijeme izvršavanja. Na taj način analizu algoritma odvajamo od stvarne implementacije nekog algoritma na nekoj konkretnoj hardversko-sofverskoj platformi. Nadalje, vrlo često nije potrebno uzimati u obzir sve operacije koje se izvršavaju u nekom algoritmu, nego je dovoljno u analizi uzeti samo one koje su najzahtjevnije sa aspekta korištenja vremenskih resursa.

Određivanje efikasnosti algoritama zapravo se svodi na izračunavanje vremenske složenosti samog algoritma u ovisnosti od veličine problema. Pod veličinom problema najčešće podrazumijevamo broj ulaznih podataka.

Karakteristike nekog algoritma mogu biti ovisne ne samo od broja ulaznih podataka, nego i o od njihovih konkretnih vrijednosti. Kao što je rečeno, razlikuju se najbolji, prosječni i najgori slučaj. Analiza najboljeg slučaja najčešće nije toliko interesantna. Ona je eventualno važna ako je vjerovatnost pojavljivanja takvih slučajeva relativno visoka. S druge strane, najčešće smo zainteresirani za analizu prosječnog i najgoreg slučaja. Čak možemo reći, da analiza prosječnog slučaja najbolje opisuje ključne karakteristike algoritma. Analiza najgoreg slučaja također ima veliku važnost zbog toga što se tom analizom dobivaju garantirane performanse algoritma jer nakon te analize dobivamo gornju granicu vremena izvođenja algoritma za bilo koji skup ulaznih podataka. Za ilustraciju prethodno rečenog uzmimo jedan jednostavan primjer. Neka je problem kojeg analiziramo pronalaženje nekog elementa u cjelobrojnem nizu. Najgori slučaj je kada element kojeg tražimo uopće nije u nizu jer je potrebno proći kroz čitav niz da bi se utvrdilo da element nije u nizu. Nadalje, iz ovog primjer možemo vidjeti da se najgori slučaj za neke probleme može relativno često ponavljati. Isto tako, postoje problemi i algoritmi kod kojih se najgori slučaj vrlo rijetko događa ili čak predstavlja samo teoretsku mogućnost. Na kraju ovog dijela spomenimo i to da postoje algoritmi kod kojih su najbolji, prosječni i najgori slučaj zapravo vrlo blizu jedan drugom, a ponekada su čak i isti.

### 2.3.1. Uobičajeni tipovi složenosti algoritama

Većina algoritama s kojima se susrećemo ima složenost proporcionalnu nekoj od sljedećih funkcija:

- 1** Označava konstantan broj operacija neovisno o veličini niza ulaznih podataka
- $\log n$**  Ovu složenost imamo *kod algoritma za traženje zadanog elementa u nekoj sortiranoj listi*. Algoritmi sa ovom složenošću su veoma prihvatljivi, jer se složenost povećava vrlo malo, npr. u slučaju udvostručenja broja ulaznih podataka, složenost je  $\log(2n) = \log n + \log 2$ .
- $n$**  Označava složenost koja je proporcionalna broju ulaznih podataka kod algoritama u kojima se određena količina posla mora obaviti nad svakim elementom ulaznog niza. Npr. *kad algoritam zahtijeva unos ili ispis  $n$  podataka*.
- $n \log n$**  Ovu složenost imamo u algoritmima koji za svaki ulazni podatak definišu podalgoritam rješiv u logaritamskom vremenu. Npr. *algoritam sortiranja* ima ovakvo ponašanje. Za umjerene vrijednosti  $n$  ovakav se algoritam ponaša skoro kao linearan, jer  $\log n$  sporo raste u odnosu prema veličini broja  $n$ . Ovakve algoritme nazivamo **loglinearnim**.
- $n^2$**  Kvadratnu ovisnost o ulaznim podacima ima algoritam koji je efikasan samo za umjerene vrijednosti broja  $n$ . Dvostruko veći niz ulaznih podataka, zahtijevat će četverostruko više vremena za rješavanje. Tipičan primjer algoritma sa ovom složenošću je *algoritam sa dvostrukom (ugnježđenom) petljom*.
- $n^\alpha$**  Razni algoritmi imat će ovu složenost za neku vrijednost od  $\alpha$ . Parametar  $\alpha$  ne mora biti prirodan broj. Kažemo da algoritmi imaju potencijalni rast. Za  $\alpha < 1$  algoritmi su brži od linearnog. Za  $\alpha > 1$  oni su sporiji od linearnog ali i od loglinernog, za iole veći broj  $n$ . Ako je  $n$  cijeli broj, govorimo da su ovi algoritmi sa **polinomskim** rastom.
- $\alpha^n$**  Pretpostavlja se da je  $\alpha > 1$ , jer složenost mora rasti sa volumenom  $n$ . Najčešće je  $\alpha = 2$ . Ovi algoritmi upotrebljivi su samo za početne vrijednosti broja  $n$ .

### 2.3.2. $O$ notacija i red složenosti algoritma

Kao što je već rečeno, vrijeme izvršavanja programa može biti procijenjeno ili izmjereno za neke konkretne veličine ulazne vrijednosti i neko konkretno izvršavanje. No, vrijeme izvršavanja programa može biti opisano opštije, u obliku funkcije koja zavisi od ulaznih argumenata.

Često se algoritmi ne izvršavaju isto za sve ulaze istih veličina, pa je potrebno naći način za opisivanje i poređenje efikasnosti različitih algoritama.

Analiza najgoreg slučaja zasniva procjenu složenosti algoritma na najgorem slučaju (na slučaju za koji se algoritam najduže izvršava — u analizi vremenske složenosti, ili na slučaju za koji algoritam koristi najviše memorije — u analizi prostorne složenosti). Ta procjena može da bude varljiva, ali predstavlja dobar opšti način za poređenje efikasnosti algoritama. U nekim situacijama moguće je izračunati prosječno vrijeme izvršavanja algoritma, ali i takva procjena bi često mogla da bude varljiva. Analiziranje najboljeg slučaja, naravno, nema smisla. U nastavku će, ako nije rečeno drugačije, biti podrazumijevana analiza najgoreg slučaja.

Neka je funkcija  $f(n)$  jednaka broju instrukcija koje zadati algoritam izvrši za ulaz veličine  $n$ . Tabela 1 prikazuje potrebno vrijeme izvršavanja algoritma ako se pretpostavi da jedna instrukcija traje jednu nanosekundu, tj. 0.001 mikrosekundi ( $\mu s$ ).

| $n \setminus f(n)$ | $\log n$      | $n$          | $n \log n$    | $n^2$       | $2^n$                    | $n!$                       |
|--------------------|---------------|--------------|---------------|-------------|--------------------------|----------------------------|
| 10                 | 0.003 $\mu s$ | 0.01 $\mu s$ | 0.033 $\mu s$ | 0.1 $\mu s$ | 1 $\mu s$                | 3.63 $ms$                  |
| 20                 | 0.004 $\mu s$ | 0.02 $\mu s$ | 0.086 $\mu s$ | 0.4 $\mu s$ | 1 $ms$                   | 77.1 $god$                 |
| 30                 | 0.005 $\mu s$ | 0.03 $\mu s$ | 0.147 $\mu s$ | 0.9 $\mu s$ | 1 $s$                    | $8.4 \times 10^{15}$ $god$ |
| 40                 | 0.005 $\mu s$ | 0.04 $\mu s$ | 0.213 $\mu s$ | 1.6 $\mu s$ | 18.3 $min$               |                            |
| 50                 | 0.006 $\mu s$ | 0.05 $\mu s$ | 0.282 $\mu s$ | 2.5 $\mu s$ | 13 $dan$                 |                            |
| 100                | 0.007 $\mu s$ | 0.1 $\mu s$  | 0.644 $\mu s$ | 10 $\mu s$  | $4 \times 10^{13}$ $god$ |                            |
| 1,000              | 0.010 $\mu s$ | 1.0 $\mu s$  | 9.966 $\mu s$ | 1 $ms$      |                          |                            |
| 10,000             | 0.013 $\mu s$ | 10 $\mu s$   | 130 $\mu s$   | 100 $ms$    |                          |                            |
| 100,000            | 0.017 $\mu s$ | 0.10 $\mu s$ | 1.67 $ms$     | 10 $s$      |                          |                            |
| 1,000,000          | 0.020 $\mu s$ | 1 $ms$       | 19.93 $ms$    | 16.7 $min$  |                          |                            |
| 10,000,000         | 0.023 $\mu s$ | 0.01 $s$     | 0.23 $s$      | 1.16 $dan$  |                          |                            |
| 100,000,000        | 0.027 $\mu s$ | 0.10 $s$     | 2.66 $s$      | 115.7 $dan$ |                          |                            |
| 1,000,000,000      | 0.030 $\mu s$ | 1 $s$        | 29.9 $s$      | 31.7 $god$  |                          |                            |

Tabela 1: Ilustracija vremena izvršavanja

Vodeći član u funkciji  $f(n)$  određuje potrebno vrijeme izvršavanja. Tako, na primjer, ako je broj instrukcija  $n^2+2n$ , onda za ulaz veličine 1000000, član  $n^2$  odnosi 16.7 minuta dok član  $2n$  odnosi samo dodatne dvije milisekunde.

Vremenska (a i prostorna) složenost je, dakle, skoro potpuno određena „vodećim“ (ili „dominantnim“) članom u izrazu koji određuje broj potrebnih instrukcija. Na upotrebljivost algoritma ne utiču mnogo ni multiplikativni ni aditivni konstantni faktori u broju potrebnih instrukcija, koliko asimptotsko ponašanje broja instrukcija u zavisnosti od veličine ulaza. Ovakav pojam složenosti uvodi se sljedećom definicijom.

**Definicija 2.1.** Ako postoje pozitivna realna konstanta  $c$  i prirodan broj  $n_0$  iz  $N$  takvi da za funkcije  $f$  i  $g$  nad prirodnim brojevima važi

$f(n) \leq c \cdot g(n)$  za sve vrijednosti  $n$  veće od  $n_0$  onda pišemo

$$f(n) = O(g(n))$$

i čitamo „**f je veliko ,o' od g**“ (tzv. Big-O notacija).

Naglasimo da  $O$  ne označava neku konkretnu funkciju, već klasu funkcija i uobičajeni zapis  $f(n) = O(g(n))$

zapravo znači  $f(n) \in O(g(n))$ .

Lako se pokazuje da aditivne i multiplikativne konstante ne utiču na klasu kojoj funkcija pripada (na primjer, u izrazu  $5n^2 + 1$ , za konstantu 1 kažemo da je aditivna, a za konstantu 5 da je multiplikativna). Zato se može reći da je ovaj algoritam složenosti  $O(n^2)$ . Obično se ne govori da pripada, na primjer, klasi  $O(5n^2 + 1)$ , jer aditivna konstanta 1 i multiplikativna 5 nisu od suštinske važnosti. Zaista, ako jedan algoritam zahtijeva  $5n^2 + 1$  instrukcija, a drugi  $n^2$ , i ako se prvi algoritam izvršava na računaru koji je šest puta brži od drugog, on će biti brže izvršen za svaku veličinu ulaza. No, ako jedan algoritam zahtijeva  $n^2$  instrukcija, a drugi  $n$ , ne postoji računar na kojem će prvi algoritam da se izvršava brže od drugog za svaku veličinu ulaza. Ovdje se radi o algoritmima različitih klasa složenosti.

Primjer 1. Može se dokazati da važi:

- $n^2 = O(n^2)$

Tvrdnja važi jer  $c = 1$  (ali i za veće vrijednosti  $c$ ), važi  $n^2 \leq c \cdot n^2$  za sve vrijednosti  $n$  veće od 0.

- $n^2 + 10 = O(n^2)$

Za  $c = 2$ , važi  $n^2 + 10 \leq c \cdot n^2$  za sve vrijednosti  $n$  veće od 3 (jer za takve vrijednosti  $n$  važi  $n^2 \leq n^2 + 10 \leq 2n^2$ ).

- $5n^2 + 10 = O(n^2)$

Za  $c = 6$ , važi  $5n^2 + 10 \leq c \cdot n^2$  za sve vrijednosti  $n$  veće od 3 (jer za takve vrijednosti  $n$  važi  $5n^2 \leq 5n^2 + 10 \leq 6n^2$ ).

- $7n^2 + 8n + 9 = O(n^2)$

Za  $c = 16$ , važi  $7n^2 + 8n + 9 \leq c \cdot n^2$  za sve vrijednosti  $n$  veće od 2 (jer za takve vrijednosti  $n$  važi  $7n^2 \leq 7n^2 + 8n + 9 \leq 16n^2$ ).

- $n^2 = O(n^3)$

Za  $c = 1$ , važi  $n^2 \leq c \cdot n^3$  za sve vrijednosti  $n$  veće od 0.

- $7 \cdot 2^n + 8 = O(2^n)$

Za  $c = 8$ , važi  $7 \cdot 2^n + 8 \leq c \cdot 2^n$  za sve vrijednosti  $n$  veće od 2 (jer za takve vrijednosti  $n$  važi  $7 \cdot 2^n \leq 7 \cdot 2^n + 8 \leq 8 \cdot 2^n$ ).

- $2^n + n^2 = O(2^n)$

Za  $c = 2$ , važi  $2^n + n^2 \leq c \cdot 2^n$  za sve vrijednosti  $n$  veće od 3 (jer za takve vrijednosti  $n$  važi  $2^n \leq 2^n + n^2 \leq 2 \cdot 2^n$ ; da važi  $n^2 \leq 2^n$  za  $n > 3$  može se dokazati, na primjer, matematičkom indukcijom).

- $5 \cdot 3^n + 7 \cdot 2^n = O(3^n)$



Za  $c = 12$ , važi  $5 \cdot 3^n + 7 \cdot 2^n \leq c \cdot 3^n$  za sve vrijednosti  $n$  veće od 0 (jer za takve vrijednosti  $n$  važi  $5 \cdot 3^n \leq 5 \cdot 3^n$  i  $7 \cdot 2^n \leq 7 \cdot 3^n$ ; da važi  $2^n \leq 3^n$  za  $n > 0$  može se dokazati, na primjer, matematičkom indukcijom).

- $2^n + 2^n n = O(2^n n)$

Za  $c = 2$ , važi  $2^n + 2^n n \leq c \cdot 2^n n$  za sve vrijednosti  $n$  veće od 0 (jer za takve vrijednosti  $n$  važi  $2^n \leq 2^n n$  i  $2^n n \leq 2^n n$ ).

**Teorema 1.** Važe sljedeća svojstva:

- Ako su  $a$  i  $b$  realni brojevi i  $a > 0$ , onda važi  $af(n) + b = O(f(n))$  (tj. multiplikativne i aditivne konstante ne utiču na klasu kojoj funkcija pripada).
- Ako važi  $f_1(n) = O(g_1(n))$  i  $f_2(n) = O(g_2(n))$ , onda važi i  $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$ .
- Ako važi  $f_1(n) = O(g_1(n))$  i  $f_2(n) = O(g_2(n))$ , onda važi i  $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$ .

**Definicija 2.2.** Ako postoje pozitivne realne konstante  $c_1$  i  $c_2$  i prirodan broj  $n_0$  takvi da za funkcije  $f$  i  $g$  nad prirodnim brojevima važi

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ za sve vrijednosti } n \text{ veće od } n_0$$

onda pišemo

$$f(n) = \Theta(g(n))$$

i čitamo „ **$f$  je veliko 'teta' od  $g$** “.

Ako važi  $f(n) = \Theta(g(n))$ , onda važi i  $f(n) = O(g(n))$  i  $g(n) = O(f(n))$ .

**Primjer 2.** Može se dokazati da važi:

- $5 \cdot 2^n + 9 = \Theta(2^n)$

Za  $c_1 = 5$ , važi  $c_1 \cdot 2^n \leq 5 \cdot 2^n + 9$  za sve vrijednosti  $n$  veće od 0 (jer za takve vrijednosti  $n$  važi  $5 \cdot 2^n \leq 5 \cdot 2^n$  i  $0 \leq 9$ ).

Za  $c_2 = 6$ , važi  $5 \cdot 2^n + 9 \leq c_2 \cdot 2^n$  za sve vrijednosti  $n$  veće od 3 (jer za takve vrijednosti  $n$  važi  $5 \cdot 2^n \leq 5 \cdot 2^n$  i  $9 \leq 2^n$ ). Iz navedena dva tvrđenja slijedi zadato tvrđenje.

- $2^n + 2^n n = \Theta(2^n n)$

Za  $c_1 = 1$ , važi  $c_1 \cdot 2^n n \leq 2^n + 2^n n$  za sve vrijednosti  $n$  veće od 0 (jer za takve vrijednosti  $n$  važi  $1 \leq 2^n$  i  $2^n n \leq 2^n n$ ).

Za  $c_2 = 2$ , važi  $2^n + 2^n n \leq c_2 \cdot 2^n n$  za sve vrijednosti  $n$  veće od 0 (jer za takve vrijednosti  $n$  važi  $2^n \leq 2$  i  $2^n n \leq 2^n n$ ). Iz navedena dva tvrđenja slijedi zadato tvrđenje.

**Definicija 2.3.** Ako je  $T(n)$  vrijeme izvršavanja algoritma  $A$  (čiji ulaz karakteriše prirodan broj  $n$ ) i ako važi  $T(n) = O(g(n))$ , onda kažemo da je algoritam  $A$  složenosti ili reda  $O(g(n))$  ili da algoritam  $A$  pripada klasi  $O(g(n))$ .

Analogno prethodnoj definiciji definiše se kada algoritam  $A$  pripada klasi  $\Theta(g(n))$ .

Informacija o složenosti algoritma u terminima  $\Theta$  (koja daje i gornju i donju granicu) preciznija je nego informacija u terminima  $O$  (koja daje samo gornju granicu). Međutim, obično je složenost algoritma jednostavnije iskazati u terminima  $O$  nego u terminima  $\Theta$ . Štaviše, za neke algoritme složenost se ne može iskazati u terminima  $\Theta$ . Npr., ako za neke ulaze algoritam troši  $n$ , a za neke  $n^2$  vremenskih jedinica, za taj algoritam se ne može reći ni da je reda  $\Theta(n)$  ni reda  $\Theta(n^2)$ , ali jeste reda  $O(n^2)$ . Kada se kaže da algoritam pripada klasi  $O(g(n))$  obično se podrazumijeva da je  $g$  najmanja takva klasa (ili makar — najmanja za koju se to može dokazati). I  $O$  i  $\Theta$  notacija se koriste i u analizi najgoreg slučaja i u analizi prosječnog slučaja.

Lako se dokazuje da funkcija koja je konstantna pripada klasama  $O(1)$  i  $\Theta(1)$ . Štaviše, svaka funkcija koja je ograničena odozgo nekom konstantom pripada klasama  $O(1)$  i  $\Theta(1)$ .

Priroda parametra klase složenosti (npr.,  $n$  u  $O(n)$  ili  $m$  u  $O(2m+k)$ ) zavisi od samog algoritma.

Složenost nekih algoritama zavisi od vrijednosti argumenata, složenost nekih algoritama zavisi od broja argumenata, a nekad se pogodno iskazuje u zavisnosti od broja bitova potrebnih da se zapiše ulaz. Npr., složenost funkcije za izračunavanje faktoriijela ulazne vrijednosti  $m$  zavisi od  $m$  i jednaka je (za razumnu implementaciju)  $O(m)$ . Složenost funkcije koja računa prosjek  $k$  ulaznih brojeva ne zavisi od vrijednosti tih brojeva, već samo od toga koliko ih ima i jednaka je  $O(k)$ . Složenost funkcije koja sabira dva broja (fiksne širine) je konstantna tj. pripada klasi  $O(1)$ . Složenost izračunavanja neke funkcije može da zavisi i od više parametara. Npr., algoritam koji za  $n$  ulaznih tačaka provjerava da li pripadaju unutrašnjosti  $m$  ulaznih trouglova, očekivano ima složenost  $O(mn)$ .

Za algoritme složenosti  $O(n)$  kažemo da imaju **linearnu**, za  $O(n^2)$  **kvadratnu**, za  $O(n^3)$  **kubnu**, za  $O(n^k)$  za neko  $k$  **polinomsku**, a za  $O(\log n)$  **logaritamsku** složenost.

### 2.3.3. Izračunavanje složenosti funkcija

Izračunavanje (vremenske i prostorne) složenosti funkcija se zasniva na određivanju tačnog ili približnog broja instrukcija koje se izvršavaju i memorijskih jedinica koje se koriste. Tačno određivanje tih vrijednosti najčešće je veoma teško ili nemoguće, te se obično koriste razna pojednostavljivanja. Na primjer, u ovom kontekstu pojam „jedinična instrukcija“ se obično pojednostavljuje, pa se smatra da i poređenje i sabiranje i množenje i druge pojedinačne naredbe troše po jednu vremensku jedinicu. Ono što je važno je da takva pojednostavljivanja ne utiču na klasu složenosti kojoj algoritam pripada (jer, kao što je rečeno, multiplikativni faktori ne utiču na red algoritma).

Ukoliko se dio programa sastoji od nekoliko instrukcija bez grananja, onda se procjenjuje da je njegovo vrijeme izvršavanja uvijek isto, konstantno, te da pripada klasi  $O(1)$ . Ukoliko dio programa sadrži petlju koja se izvršava  $n$  puta, a vrijeme izvršavanja tijela petlje je konstantno, onda ukupno vrijeme izvršavanja petlje pripada klasi  $O(n)$ . Ukoliko dio programa sadrži jednu petlju koja se izvršava  $m$  puta i jednu petlju koja se izvršava  $n$  puta, a vremena izvršavanja tijela ovih petlji su konstantna, onda ukupno vrijeme izvršavanja petlje pripada klasi  $O(m + n)$ . Generalno, ukoliko program ima dva dijela, složenosti  $O(f(n))$  i  $O(g(n))$ , koji se izvršavaju jedan za drugim, ukupna složenost je  $O(f(n) + g(n))$ . Ukoliko dio programa sadrži dvostruku petlju – jednu koja se izvršava  $m$  puta i, unutar nje, drugu koja se izvršava  $n$  puta i ukoliko je vrijeme izvršavanja tijela unutrašnje petlje konstantno, onda ukupno vrijeme izvršavanja petlje pripada klasi  $O(m \cdot n)$ . Ukoliko dio programa sadrži jedno

grananje i ukoliko vrijeme izvršavanja jedne grane pripada klasi  $O(m)$  a druge grane pripada klasi  $O(n)$ , onda ukupno vrijeme izvršavanja tog dijela programa pripada klasi  $O(m + n)$ .

Analogno se računa složenost za druge vrste kombinovanja linearnog koda, grananja i petlji. Za izračunavanje složenosti rekurzivnih funkcija potreban je matematički aparat za rješavanje rekurentnih jednačina i na ovo ćemo se vratiti nakon što se upoznamo sa rekurzivnim algoritmima.

#### 2.3.4. Popravljanje vremenske složenosti

Ako performanse programa ne zadovoljavaju, potrebno je razmotriti zamjenu ključnih algoritama tj. onih koji dominantno utiču na složenost programa. Ako to nije moguće, onda se efikasnost programa treba popraviti u pogledu broja pojedinačnih izvršenih naredbi (koje utiču na ukupno utrošeno vrijeme). Najčešće ovo zahtijeva neka specifična rješenja, ali postoji i puno ideja koje se mogu primjeniti u većini slučajeva. To su:

- Koristiti optimizacije kompajlera. Moderni kompajleri podrazumijevano generišu veoma efikasan kôd, ali mogu omogućiti primjene i dodatnih tehnika optimizacije. Pritom, treba biti oprezan sa optimizacijom jer izvršni kôd ne odgovara direktno izvornom (npr. moguće je da se optimizacijom u izvršnom programu ne predviđa prostor za promjenljive za koje je utvrđeno da se ne koriste), pa optimizacija može tada da promijeni i očekivano ponašanje programa. Također, zbog optimizacijom narušene veze između izvornog i izvršnog programa, često je nemoguće debug-erom analizirati program. Kompajliranje sa intenzivnom optimizacijom traje duže od običnog kompajliranja. Zbog svega toga, potrebno je optimiziranje programa primjeniti nakon intenzivnog testiranja i otklanjanja svih otkrivenih grešaka. Naravno, potrebno je također uraditi i testiranje optimizirane verzije izvršnog programa.
  - Optimizacijama sprovedenim od strane kompajlera moguće je ubrzati program i do nekoliko desetina procenata, ali je isto tako moguće i da optimizovani program na kraju bude sporiji od neoptimiziranog.
  - Kompajleri obično nude mogućnost eksplicitnog odabira tehnika optimizacije ili nivo optimizacije.
  - Kako optimizacija produžava vrijeme kompajliranja, poželjno ju je koristiti samo u završnim fazama razvoja programa.
- Ne treba optimizovati nebitne dijelove programa. Ako neki dio programa vrlo malo utiče na njegovu efikasnost, ne treba ga optimizovati, bolje da ostane u jednostavnom i lako razumljivom obliku. Treba optimizovati one dijelove programa koji troše najviše vremena.
- Izdvojiti izračunavanja koja se ponavljaju. Izračunavanje koje troši resurse ne ponavljati. U dolje navedenom primjeru vrijednosti funkcije `cos` i `sin` se izračunavaju dva puta za istu vrijednost argumenta. Kako su ove funkcije vremenski veoma zahtijevne, bolje je izračunavanje uraditi jednom i u pomoćnim promjenljivima sačuvati te vrijednosti prije korištenja.

```
x = x0*cos(0.01) - y0*sin(0.01);  
y = x0*sin(0.01) + y0*cos(0.01);
```

Umjesto gornjeg koda, bolje je uraditi:

```
cs = cos(0.01);
sn = sin(0.01);
x = x0*cs - y0*sn;
y = x0*sn + y0*cs;
```

- Izdvojiti kod izvan petlje. Ukoliko postoje izračunavanja koja su ista u svakom prolasku kroz petlju, potrebno ih je izdvojiti iz petlje. Tako, umjesto:

```
for (int i = 0; i < s.length(); i++)
{
    if (s[i] == c)
        ...
}
```

puno je bolje napisati:

```
len = s.length();
for (int i = 0; i < len; i++)
{
    if (s[i] == c)
        ...
}
```

(Primjer lošijeg koda u C++)

```
#include<iostream>
#include<string>
using namespace std;
void main()
{
    int br = 0;
    string s = "danass";
    char c = 'a';
    for (int i = 0; i < s.length(); i++)
    {
        if (s[i] == c)
            br++;
    }
    cout << br << endl;
}
```

(Primjer boljeg koda u C++)

```
#include<iostream>
#include<string>
using namespace std;
void main()
{
    int br = 0, len;
    string s = "danass";
    char c = 'a';
    len = s.length();
    for (int i = 0; i < len; i++)
    {
        if (s[i] == c)
            br++;
    }
}
```

```

}
cout << br << endl;
}

```

U prvom slučaju kroz petlju se prolazi  $n$  puta, gdje je  $n$  dužina stringa  $s$ . Ali, u svakom prolazu kroz petlju se poziva funkcija `length()` za argument  $s$  i u svakom pozivu se prolazi kroz cijeli string  $s$ . Zbog ovoga je složenost prve petlje barem  $O(n^2)$ . Nasuprot tome, u drugom slučaju kod optimizovane verzije kôda funkcija `length()` se poziva samo jednom i složenost tog poziva i petlje koja slijedi ukupno je  $O(n)$ .

Slično umjesto kôda:

```

for (i=0; i < N; i++) {
    x = x0*cos(0.01) - y0*sin(0.01);
    y = x0*sin(0.01) + y0*cos(0.01);
    ...
    x0 = x;
    y0 = y;
}

```

puno je bolji kôd:

```

cs = cos(0.01);
sn = sin(0.01);
for (i=0; i < N; i++) {
    x = x0*cs - y0*sn;
    y = x0*sn + y0*cs;
    ...
    x0 = x;
    y0 = y;
}

```

- Zamijeniti skupe operacije sa jeftinim. Npr., uslov  $\sqrt{x_1x_1 + y_1y_1} > \sqrt{x_2x_2 + y_2y_2}$  je ekvivalentan  $x_1x_1 + y_1y_1 > x_2x_2 + y_2y_2$ , ali je u programu umjesto uslova `sqrt(x1*x1+y1*y1)>sqrt(x2*x2+y2*y2)` koristiti `x1*x1+y1*y1>x2*x2+y2*y2` jer se na ovaj način izbjegava pozivanje veoma skupe funkcije `sqrt`. Analogno ovome, ako je moguće poželjno je izbjegavati i skupe trigonometrijske funkcije, umjesto “većih” poželjno je koristiti “manje” tipove podataka itd.
- Ne ostavljati za fazu izvršavanja izračune koji se mogu obaviti ranije. Ako se tokom izvršavanja programa više puta koriste vrijednosti iz malog skupa, uštedu može da donese njihovo izračunavanje unaprijed i uključivanje rezultata u izvorni kod programa. Npr. ako se u nekom programu koriste vrijednosti kvadratnog korijena od 1 do 100, te vrijednosti se mogu izračunati unaprijed i njima se može inicijalizovati konstantni niz. Ovo se koristi ako je kritični resurs vrijeme, a ne prostor.
- Napisati kritične dijelove koda na assembleru. Moderni kompajleri generišu veoma kvalitetan kod. Ukoliko se koriste i raspoložive optimizacije, kod takvog kvaliteta može da napiše rijetko koji programer. Ipak, u nekim slučajevima, za neke vremenski kritične dijelove programa, opcija je pisanje tih dijelova programa na assembleru.

### 2.3.5. Popravljanje prostorne složenosti

Na današnjim računarima memorija obično nije kritični resurs. Optimizacija se obično svodi na uštedu vremena, a ne prostora. Ipak, postoje situacije kada je potrebno voditi računa o memoriji u cilju njene uštede : kada program barata sa ogromnom količinom podataka, kada je i sam program velik i zauzima značajan dio RAM-a ili kada se program pokreće i izvršava na nekom specifičnom uređaju koji ima ograničenu tj. malu količinu memorije. Često ušteda memorije zahtijeva specifična rješenja, ali postoje i neke ideje koje su primjenljive u većem broju slučajeva:

- Koristiti najmanje moguće tipove podataka. Za cjelobrojne tipove umjesto tipa `int` često je dovoljan tip `short` ili čak `char`. Za predstavljanje realnih brojeva, ako preciznost nije kritična, umjesto tipa `double` treba koristiti tip `float`. Za predstavljanje logičkih vrijednosti dovoljan je jedan bit, a više takvih vrijednosti može da se čuva u jednom bajtu (i da im se pristupa koristeći bitovske operatore).
- Ne čuvati ono što može da se lako izračuna. U slučaju da je kritična brzina, a ne prostor, dobro da se vrijednosti koje se često koriste u programu izračunaju unaprijed i uključe u izvorni kod programa. Ako je kritična memorija, onda treba uraditi suprotno i ne čuvati nikakve vrijednosti koje se mogu izračunati u fazi izvršenja.

#### Zaključak - O notacija

Za dovoljno veliki  $n$  vrijedi:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \dots < O(2^n) < O(n!)$$

- $O(1)$  znači da je vrijeme izvođenja ograničeno konstantom
- ostale vrijednosti, do predzadnje, predstavljaju polinomna vremena izvođenja algoritma, svako sljedeće vrijeme izvođenja je veće za red veličine
- predzadnji izraz predstavlja eksponencijalno vrijeme izvođenja, ne postoji polinom koji bi ga mogao ograničiti jer za dovoljno veliki  $n$  ova funkcija premašuje bilo koji polinom
- algoritmi koji zahtijevaju eksponencijalno vrijeme mogu biti nerješivi u razumnom vremenu, bez obzira na brzinu računara.

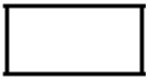
### 3. STRUKTURE PODATAKA

#### 3.1. Elementi od kojih se grade strukture podataka

Struktura podataka se sastoji od manjih cjelina koje se udružuju u veće i međusobno povezuju vezama.

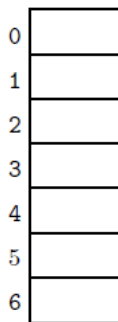
Uvodimo posebne nazive za cjeline, načine udruživanja i načine povezivanja. Također, uvodimo pravila kako se strukture prikazuju dijagramima.

Ćelija ... varijabla koju promatramo kao zasebnu cjelinu. Svaka ćelija ima svoj tip i adresu.

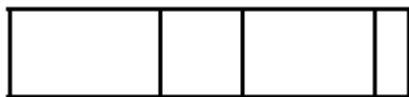


Polje ... (array u C++-u). Mehanizam udruživanja manjih dijelova strukture u veće. Polje čini više ćelija istog tipa pohranjenih na uzastopnim adresama. Broj ćelija je unaprijed zadan i nepromjenljiv.

Jedna ćelija se zove element polja i jednoznačno je određena pripadnom vrijednošću indeksa. Po ugledu na C++, uzimamo da su indeksi 0, 1, 2, ..., N-1, gdje je N cjelobrojna konstanta.



Zapis ... (slog, structure u C++-u). Također mehanizam udruživanja manjih dijelova strukture u veće. Zapis čini više ćelija, koje ne moraju biti istog tipa, no koje su pohranjene na uzastopnim adresama. Broj, redosljed i tip ćelija je unaprijed zadan i nepromjenljiv. Pojedina ćelija se zove komponenta zapisa. Polja i zapisi se mogu kombinirati. Na primjer, možemo imati polje zapisa, zapis čije pojedine komponente su polja, polje od polja, zapis čija komponenta je zapis, i slično.



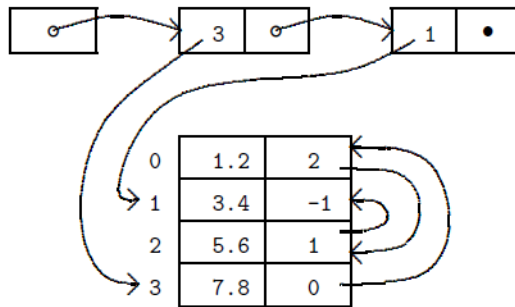
Pointer ... služi za uspostavljanje veze između dijelova strukture. Pointer ili pokazivač je ćelija koja pokazuje neku drugu ćeliju. Sadržaj pointera je adresa ćelije koju treba pokazati.



Kursor ... također služi za uspostavljanje veze između dijelova strukture. Kursor je ćelija tipa int koja pokazuje na element nekog polja. Sadržaj kursora je indeks elementa kojeg treba pokazati.



Dijagram ispod pokazuje primjer strukture podataka koja se sastoji od niza zapisa povezanih pomoću pointera, te od jednog polja zapisa. Zapisi su još međusobno povezani kursorima.



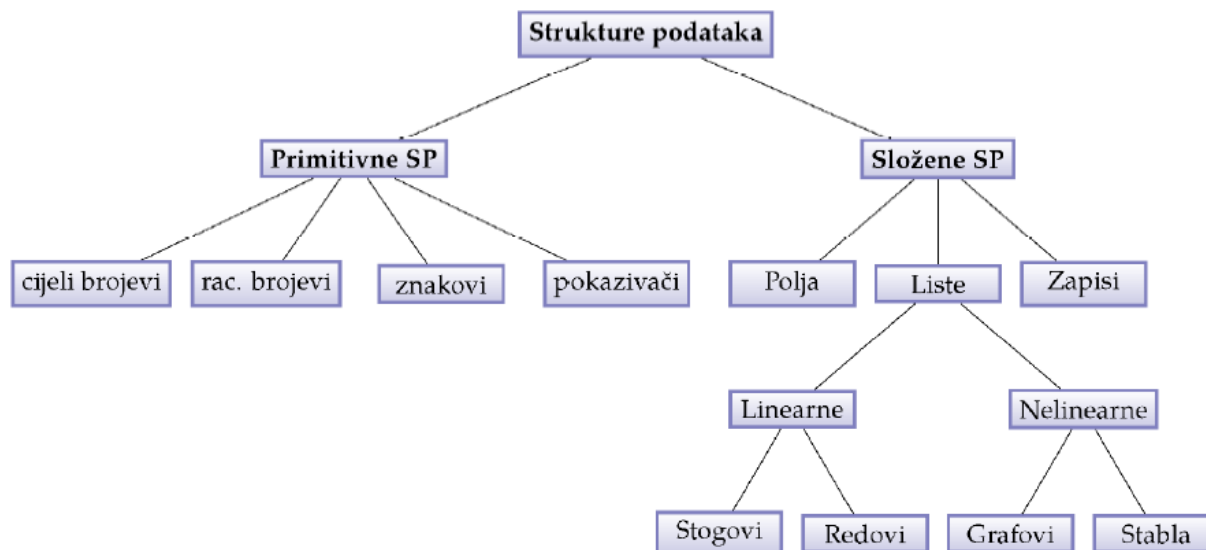
Slika 1.2 Primjer strukture podataka.

### 3.2. Podjela struktura podataka

Generalno strukture podataka možemo klasificirati u 2 kategorije prikazane na slici ispod:

- Primitivne strukture podataka – osnovne strukture sa kojima se radi direktno sa mašinskim instrukcijama. U ovu grupu spadaju: cijeli brojevi, brojevi sa pomičnim zarezom, znakovi i pokazivači ili pointeri.
- Složene strukture podataka - napredniji oblik grupisanja podataka istog ili različitog tipa. Kreiraju se od primitivnih tipova podataka. U ovu grupu spadaju: polja, liste, stabla i grafovi kao i razne varijacije ovih struktura.





Slika 3.1. Klasifikacija struktura podataka

### 3.2.1. Polja

U programiranju se često susrećemo s upotrebom jednodimenzionalnih polja odnosno nizova. Korištenjem polja možemo na smislen način grupisati više podataka istoga tipa. Npr., želimo li izračunati ali i pohraniti ocjene sa ispita iz nekog predmeta za sve studente koji su izašli na ispit, jedan od načina je korištenjem polja.

Polje, dakle, predstavlja sljedni niz memorijskih lokacija rezervisan za određeni tip podatka. Svakom elementu polja možemo pristupiti na temelju indeksnih varijabli. Operacije pristupa elementima polja zahtjeva  $O(1)$  vrijeme.

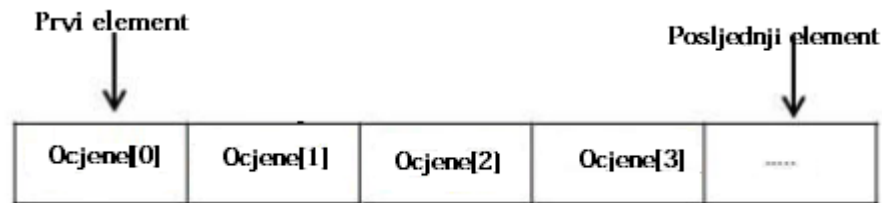
U ovom dijelu obradit će se osnovni pojmovi iz nizova i stringova u programskom jeziku C++. To su:

- Deklaracija, inicijalizacija i korištenje jednodimenzionalnih i višedimenzionalnih nizova.
- Nizovi i funkcije
- Pokazivači i nizovi
- Nizovi karaktera u C-u (C-stringovi)
- Pokazivači na nizove karaktera
- Funkcije za rad sa C-stringovima

#### 3.2.1.1. Jednodimenzionalna polja ili nizovi

Promjenljive sa kojima smo radili do sada mogle su istovremeno da čuvaju samo jednu vrijednost. Nizovi omogućavaju da koristimo jednu promjenljivu koja će čuvati ogroman broj podataka. Članovi niza zauzimaju susjedne memorijske lokacije (Slika 3.2.). Adresa sa najmanjom vrijednošću (najniža adresa) odgovara prvom članu niza dok najviša adresa

odgovara posljednjem članu niza. Članovima niza pristupamo na osnovu indeksa ili kursora koji označava poziciju člana u nizu. Prvi član niza ima indeks 0 dok se indeks zatim povećava za jedan za svaki uzastopni član niza. Ustvari, indeks nekog elementa niza govori koliko je taj element udaljen od prvog elementa niza.



Slika 3.2. Predstavljanje niza u memoriji računara

Korištenje jedne promjenljive za niz, koja čuva 100 vrijednosti, ima mnogo veće pogodnosti nego korištenje 100 različitih promjenljivih gdje svaka može da čuva samo jednu vrijednost. Također, mnogo je lakše pratiti promjene jedne u odnosu na 100 različitih promjenljivih. Najveća pogodnost korištenja nizova je ta što se mogu koristiti petlje (ciklusi) da bi se pristupilo svakom uzastopnom elementu niza.

Kada je u pitanju rad sa karakterima, često se dešava da se korisnički ulaz u program sastoji iz više od jednog karaktera. Pojedinačni karakteri se također mogu organizovati u niz. Nizovi karaktera obično sadrže null karakter kao posljednji član niza. Takvi nizovi karaktera koji na kraju sadrže null karakter se nazivaju „C-stringovi“. Na kraju lekcije će biti opisane funkcije koje se koriste za C-stringove.

#### Predstavljanje niza

Niz je struktura podatka koja služi da se u njoj smjesti kolekcija elemenata istoga tipa i fiksne je veličine. Deklaracija niza u C++-u ima sljedeći oblik (prvo se navodi tip promjenljive, zatim ime promjenljive, a u uglastim zagradama [ ] se navodi broj elemenata niza – tj, veličina niza):

Sintaksa :

```
tip_podatka ime_polja [ veličina_niza ];
```

Ovo je jednodimenzionalno polje ili niz. Veličina niza (veličina\_niza) mora biti pozitivna cjelobrojna vrijednost (veća od nule) dok tip (tip\_podatka) može biti bilo koji validan C/C++ tip podatka (primitivan ili složen).

Primjer niza:

```
int Ocjene[ 5 ];
```

#### Inicijalizacija niza

Inicijalizacija niza je dodjeljivanje vrijednosti elementima niza u istom iskazu u kome se vrši deklaracija niza.

Inicijalizacija niza se može izvršiti ili član po član, ili je moguće koristiti sljedeći izraz:

```
int Ocjene[5] = {10, 6,8,7,8};
```

pri čemu broj vrijednosti unutar vitičastih zagrada { } ne smije biti veći od maksimalnog broja elemenata niza koji smo naveli u okviru uglastih zagrada [ ]. Može biti manji.

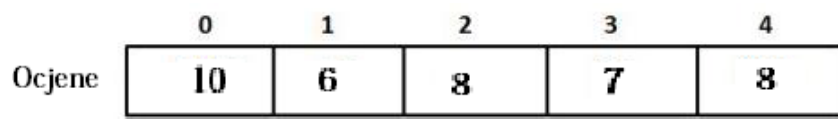
Također je moguće u toku deklaracije izostaviti maksimalnu dimenziju, a samo u okviru vitičastih zagrada inicijalizovati članove niza, što će automatski setovati maksimalnu veličinu niza na broj unesenih elemenata. Stoga, ukoliko napišemo:

```
int Ocjene[] = {10, 6, 8, 7, 8};
```

kreiraće se potpuno identičan niz kao u prethodnom primjeru, gdje je niz imao 5 elemenata. U nastavku je dat primjer dodjele vrijednosti odgovarajućem članu niza, kojem naravno pristupamo preko indeksa:

```
Ocjene [4] = 8;
```

Prethodni izraz dodjeljuje petom članu niza vrijednost 8. Kao i u Java-i, indeks niza počinje sa nulom (Ocjene [0]) dok posljednji član niza ima indeks N-1, gde je sa N označena dimenzija niza. Na narednoj slici je grafički prikazan niz iz prethodnog primjera:



Slika 3.3. Predstavljanje niza Ocjene u memoriji računara

Operator sizeof () i nizovi

Operator sizeof koristimo za određivanje broja elemenata niza u slučaju kada dimenzija niza nije navedena u toku inicijalizacije.

```
int Ocjene[] = { 10, 6, 8, 7, 8 }; // deklaracija niza od 5 elemenata
cout << sizeof(Ocjene);           // prikazuje 20 (5 elemenata * 4 bajta za svaki)
```

U programskom jeziku C++ ne postoji direktan način da se ispita kolika je veličina niza ali je moguće to utvrditi korišćenjem operatora sizeof na sljedeći način:

```
int nElements = sizeof(Ocjene) / sizeof(Ocjene[0]);
cout << nElements << endl;
```

Obzirom da svi elementi niza imaju istu veličinu (pošto se radi o istom tipu podatka), dijeljenjem ukupne količine memorije niza sa veličinom memorije koja odgovara jednom elementu niza dobijamo broj elemenata. Pri tome treba koristiti element sa indeksom 0 s obzirom da niz koji je deklarisan mora imati bar jedan element.

### *Pristupanje elementu niza*

Elementu niza se pristupa preko indeksa koji stoji u uglastim zagradama uz ime niza.

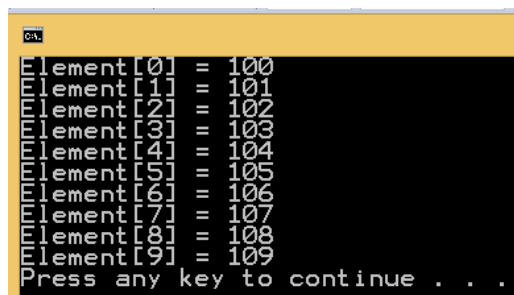
```
#include<iostream>
#include <cstdlib>
using namespace std;
void main()
{
    const int N = 10;
    int n[N]; /* n je niz od 10 cjelih brojeva */
```

```

int i, j;
for (i = 0; i < N; i++)
{
    n[i] = i + 100; /* vrijednost elementa na poziciji i postavi na i + 100 */
}
/* ispis vrijednosti svakog elementa niza */
for (j = 0; j < N; j++)
{
    printf("Element[%d] = %d\n", j, n[j]);
}
}

```

Nakon izvršavanja prethodnog programa, na ekranu će biti ispisan sljedeći rezultat:



```

Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
Press any key to continue . . .

```

### *Najčešća greška pri radu sa nizom: IZLAZAK IZ OPSEGA*

Izlazak iz opsega je jedna od težih grešaka za otkrivanje a može da izazove ozbiljne probleme

```

{
    const int duzina_niza = 5;
    int niz[duzina_niza] = { 6, 8, 2, 4, 9 };
    int maxValue = 0;
    for (int nIndex = 0; nIndex <= duzina_niza; nIndex++)
        if (niz[nIndex] > maxValue)
            maxValue = niz[nIndex];
    cout << "Max el je " << maxValue << endl;
}

```

Problem sa prethodnim kodom je taj što je uslovni izraz u okviru for naredbe netačan. Deklarisani niz ima 5 elemenata, tako da indeksi idu u opsegu od 0 do 4. Međutim, petlja ide od 0 do 5. Kao posljedica, u posljednjem ciklusu for petlje izvršit će se sljedeća sekvenca kôda:

```

if (niz[5] > maxValue)
    maxValue = niz[5];

```

ali član niz [5] našeg niza nije definisan! Ovo može da izazove različite probleme, a najvjerovatnije je da će niz [5] imati neku nedodijeljenu vrijednost ili beskonačno veliki broj. Kao posljedica greške u pisanju kôda promjenljivoj maxValue će biti dodijeljena pogrešna vrijednost.

Ono što je još veći problem kod izlaska iz opsega je dodjela vrijednosti članu niz[5], što može dovesti do izmjene neke druge promjenljive (ili dijela promjenljive). Ovakve greške u programu su jedne od težih za otkrivanje a mogu da izazovu ozbiljne probleme!

### Prosljeđivanje niza funkciji

Da bi se niz prosljedio funkciji kroz listu argumenata, postoje tri načina na koja se to može uraditi. Sva tri načina su veoma slična međusobno, i proizvode isti rezultat, zato što svi slučajevi šalju poruku kompajleru da se radi sa pokazivačem na cjelobrojnu vrijednost. Na sličan način je moguće proslijediti i višedimenzionalne nizove što će biti pokazano u nastavku. Postoje tri načina da se niz kroz listu argumenata proslijedi funkciji: kao pokazivač, kao dimenzionisani niz i kao nedimenzionisani niz.

- Prosljeđivanje niza korištenjem pokazivača kao formalnog parametra:

```
void myFunction(int *param)
{
    ...
}
```

- Formalni parametar kao dimenzionisani niz, kao što je dato u nastavku:

```
void myFunction(int param[10])
{
    ...
}
```

- Formalni parametar kao nedimenzionisani niz:

```
void myFunction(int param[])
{
    ...
}
```

### Primjer korištenja nizova u funkciji

Kada se ime niza proslijedi funkciji kroz listu argumenata prosljeđivanje se vrši po adresi.

Donji primjer koristi funkciju `getProsjek()` koja kao argumente uzima nedimenzionisani niz `arr` i veličinu niza `size`, a kao rezultat vraća srednju vrijednost članova niza:

```
#include<iostream>
using namespace std;
double getProsjek(int arr[], int size)
{
    int i; double avg, sum = 0;
    for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }
    avg = sum / size;
    return avg;
}
int main()
{
    const int N = 5;
    /* cjelobrojni niz sa 5 elemenata */
    int ocjene[N] = { 10, 6, 8, 7, 8 };
    double avg;
    /* proljeđuje pointer na niz kao argument */
}
```

```

    avg = getProsjek(ocjene, N);
    /* prikaz izlazne vrijednosti*/
    cout << "Prosjecna vrijednost:" << avg << endl;
    return 0;
}

```

Kao što se vidi iz prethodnog primjera, nije neophodno navesti dimenziju niza u listi formalnih parametara jer C/C++ ne provjerava granice formalnih parametara (isto je kod C# i Java).

### *Problem pretraživanja*

Polje kao skup podataka istog tipa može biti efikasna struktura podataka za sortiranje ili pretraživanje brojeva.

Npr., neka imamo:

Ulaz: Niz od  $n$  brojeva  $A = \{a_1, a_2, \dots, a_n\}$  i vrijednost  $x$

Izlaz: Indeks  $i$  takav da  $x = A[i]$  ili specijalna vrijednost NIL ukoliko  $x$  nije u  $A$ .

U pseudo jeziku predstaviti ćemo algoritam koji dati broj  $x$  nalazi u zadanom polju u  $O(n)$  vremenu. Definirat ćemo funkciju koja će nam vratiti indeks lokacije u kojoj je nađeno  $x$ . Ukoliko  $x$  nije u polju vratit će se prazan indeks. Koristit ćemo algoritam sekvencijalnog pretraživanja.

seqPretrazi( $A, x$ )

```

1 for i ← 1 to length(n)
2     do if x = A[i]
3         then return i
4 return NIL

```

Uočavamo da vrijeme izvršenja algoritma zavisi od broja iteracija for petlje što nam daje  $O(n)$  vremensku složenost. Korektnost algoritma je jasna: prolazimo kroz sve elemente polja, ukoliko takav broj nađemo, vraćamo indeks tog mjesta. Ako ne nađemo, sa return NIL vraćamo prazan indeks. U pseudokodu je indeks prvog elementa označen brojem 1.

Konkretni programski kôd za slučaj pronalaska zajedničkih elemenata u dva 1D polja uz pomoć algoritma sekvencijalnog pretraživanja izgleda:

```

#include <iostream>
#include <string>
using namespace std;
int seqPretrazivanje(const int x[], int size, int key)
{
    for (int i = 0; i < size; i++)
        if (key == x[i]) return i;
    return -1;
}
void presjek(const int set1[], int size1,
            const int set2[], int size2, int s[], int &size)
{
    size = 0;
    for (int i = 0; i < size1; i++)

```

```

        if (seqPretrazivanje(set2, size2, set1[i]) > -1)
            s[size++] = set1[i];
    }
    void display(const int x[], int size)
    {
        for (int i = 0; i < size; i++)
            cout << x[i] << endl;
    }
    void main()
    {
        int x[] = { 1,2,3,4 };
        int y[] = { 10,2,30,4,50 };
        int size = 4;
        int p[4];
        presjek(x, 4, y, 5, p, size);
        cout << "Zajednicki elementi su:" << endl;
        display(p, size);
    }

```

Izlaz:

```

Zajednicki elementi su:
2
4

```

### Problem sortiranja

Promotriti ćemo dalje problem sortiranja polja. Polje može sadržavati elementarne tipove podataka (alfanumerički podaci).

Ulaz: Niz od  $n$  brojeva  $A = \{a_1, a_2, \dots, a_n\}$ .

Izlaz: Permutacija  $\{a_1', a_2', \dots, a_n'\}$  ulaznog niza tako da je  $a_1' \leq a_2' \leq \dots \leq a_n'$

U pseudo jeziku predstaviti ćemo algoritam koji sortira naš niz, a koji se može opisati analogijom slaganja karata: svaka nova karta koja se izvuče iz špila dodaje se u skup karata koje se već nalaze u ruci. Ukoliko je taj skup karata sortiran treba novu kartu dodati na odgovarajuće mjesto. Ideja je da se pomaknu sve one karte koje su veće od te karte u desno i na prazno mjesto koje je ostalo stavi se ta karta.

Algoritam koji sortira polje na takav način je poznat kao InsertionSort algoritam čiji pseudokôd glasi:

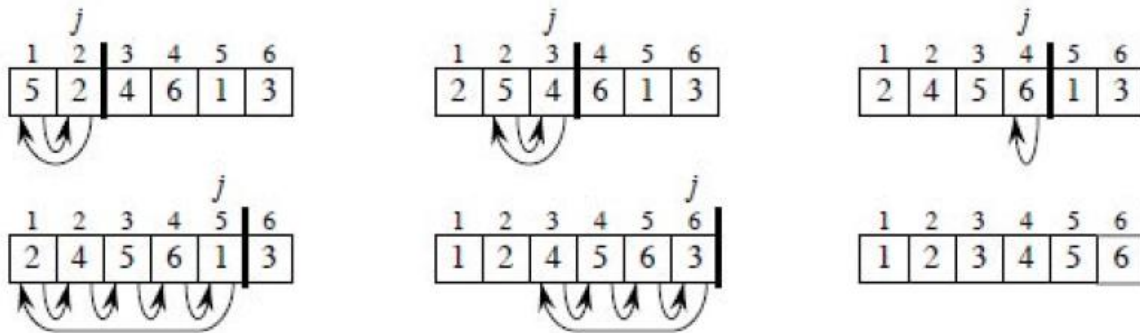
InsertionSort(A)

```

1 for  $j \leftarrow 2$  to length(A)
2     do key  $\leftarrow A[j]$ 
3      $i \leftarrow j-1$ 
4     while  $i > 0$  and  $A[i] > \text{key}$ 
5         do  $A[i+1] \leftarrow A[i]$ 
6          $i \leftarrow i-1$ 
7      $A[i+1] \leftarrow \text{key}$ 

```

U pseudokodu je indeks prvog elementa podrazumijevano 1. Za niz { 5,2,4,6,1,3 } grafički prikaz sortiranja uz pomoć InsertionSort algoritma predstavljen je na slici 3.6.



Slika 3.6. Grafički prikaz sortiranja uz pomoć InsertionSort algoritma za polje A={5,2,4,6,1,3}

Konkretni programski kôd za slučaj sortiranja elemenata u rastućem poretku u 1D polju uz pomoć algoritma sortiranja umetanjem (InsertionSort algoritam) izgleda:

```
#include <iostream>
#include <string>
using namespace std;
int indexOdNajveceVrijednosti(const int x[], int size)
{
    // Pretpostavimo da je najveća vrijednost x[0]
    int max = x[0];
    int index = 0;
    for (int i = 1; i < size; i++)
        if (max < x[i])
        {
            max = x[i];
            index = i;
        }
    return index;
}
void zamijeni(int &x, int &y)
{
    int pom = x;
    x = y;
    y = pom;
}
void insertionSort(int x[], int n)
{
    for (int i = 0; i < n; i++)
        zamijeni(x[indexOdNajveceVrijednosti(x, n - i)], x[n - i - 1]);
}
void display(string p, int x[], int size)
{
    cout << p;
    for (int i = 0; i < size; i++)
        cout << x[i] << " ";
}
```



```

        cout << endl;
    }
    void main()
    {
        int array[10] = { 4,1,3,2,7,6,10,9,5,8 };
        display("Nesortirano polje:", array, 10);
        insertionSort(array, 10);
        display("Sortirano polje: ", array, 10);
    }

```

Izlaz:

```

Nesortirano polje: 4 1 3 2 7 6 10 9 5 8
Sortirano polje:  1 2 3 4 5 6 7 8 9 10

```

Promotrimo vrijeme izvršenja ovog algoritma:

- Najbolji slučaj – situacija kad je polje već sortirano, korak 5. while petlje će vršiti samo 1 provjeru za svaku iteraciju for petlje, koraci 6,7 neće biti izvršavani.

$$T(n) = \sum_{j=2}^n O(1) = O(n).$$

- Najgori slučaj – input polje je obrnuto sortirano, za svaku j iteraciju for petlje while petlja će vršiti j - 1 pomjeranja elemenata.

$$T_w(n) = \sum_{j=2}^n O(j - 1) = O(n^2)$$

- Prosječni slučaj  $T_A(n) = \sum_{j=2}^n f_j O(1) = O(n^2)$  gdje je  $f_j$  broj iteracija ( $1 \leq f_j \leq j - 1$ ) while petlje u koraku 4., prilikom j-te iteracije for petlje.

Sortirano polje može ubrzati potragu za nekim elementom

Ako je polje dužine n sortirano, tada se traženje željene vrijednosti x može ubrzati i traje

$O(\log n)$  vremena. Takav algoritam koristi svojstvo sortiranosti kako bi u svakoj iteraciji "područje pretraživanja" reducirao za značajni faktor. Uzmimo da algoritam uspoređuje x sa središnjim elementom polja  $A[m]$  i na temelju te usporedbe odredi ili lijevu ili desnu polovicu polja u kojem će nastaviti pretraživanje. Pseudokodom algoritam možemo zapisati:

BinarySearch(A, x, p, r)

Algoritam u polju A traži element x od  $A[p]$  do  $A[r]$

```

1  while p < r
2      do  $m \leftarrow \frac{p+r}{2}$            //nađi indeks sredine polja
3          if x = A[m]
4              then return m
5          if x > A[m]
6              then  $p \leftarrow m + 1$ 
7          else  $r \leftarrow m - 1$ 
8  return NIL           // vrati prazan indeks

```

Algoritam završava ukoliko broj  $x$  je pronađen u polju  $i$  ukoliko ga nije našao (situacija kada  $p > r$ ) vraća praznu vrijednost.

Sljedeći algoritam za sortiranje je poznat kao algoritam mjehuričastog sortiranja (BubbleSort). Princip mu je baziran na uzastopnoj zamjeni susjednih elemenata koji nisu poredani.

BubbleSort(A)

Algoritam sortira polje A uspoređivanjem susjednih elemenata

```
1  for  $i \leftarrow 1$  to length[A]
2      do for  $j \leftarrow$  length[A] downto  $i + 1$ 
3          do if  $A[j] < A[j - 1]$ 
4              then exchange  $A[j] \leftrightarrow A[j - 1]$ 
```

Konkretni programski kôd za slučaj sortiranja elemenata u rastućem poretku u 1D polju uz pomoć algoritma mjehuričastog sortiranja (BubbleSort) izgleda:

```
#include <iostream>
#include <string>
using namespace std;
void zamijeni(int &x, int &y)
{
    int pom = x;
    x = y;
    y = pom;
}
void bubbleSort(int x[], int n)
{
    for (int i = 0; i < n; i++)
        for (int j = n - 1; j > i; j--)
            if (x[j] < x[j - 1])
                zamijeni(x[j], x[j - 1]);
}
void display(string p, int x[], int size)
{
    cout << p;
    for (int i = 0; i < size; i++)
        cout << x[i] << " ";
    cout << endl;
}
void main()
{
    int array[10] = { 4,6,3,9,7,1,10,2,5,8 };
    display("Nesortirano polje:", array, 10);
    bubbleSort(array, 10);
    display("Sortirano polje: ", array, 10);
}
```

Algoritmi Bubblesort i InsertionSort imaju  $O(n^2)$  vremensku složenost i ne predstavljaju najbrže algoritme za sortiranje, brži algoritmi kao što su MergeSort i QuickSort sortiraju polja u  $O(n \log n)$  vremenu, a o njima će biti više riječi kada budemo govorili o rekurzijama.

### *Pokazivači na nizove*

Ime niza ustvari predstavlja pokazivač na prvi element niza.

Za niz

```
int ocjene[10];
```

ocene je pokazivač tj. adresa prvog elementa niza ocjene (ocene[0]). Tako, u narednom dijelu koda, pokazivaču p ćemo dodijeliti adresu prvog elementa niza ocjene:

```
int *p;
```

```
int ocjene[10];
```

```
p = ocjene;
```

U C-u (a i C++-u) je dozvoljeno koristiti imena niza kao konstantne pokazivače, i obrnuto. Stoga, korištenje izraza `*(ocene + 4)` je legalan način da se pristupi podatku 8 člana niza odnosno članu ocjene[4] sa slike 3.3.

Treba znati da ako jednom pokazivaču p dodijelimo adresu niza onda možemo elementima niza pristupiti i pomoću: `*p`, `*(p+1)`, `*(p+2)` i tako dalje.

### *Pokazivačka aritmetika*

Operacije uvećanja i umanjeanja pokazivača na niz utiču na to da se pokazivač pomjeri na naredni odnosno prethodni element niza.

U programskom jeziku C/C++ moguće je koristiti binarne operatore + i - u cilju obavljanja aritmetičkih operacija nad pokazivačima. Npr., može se modifikovati pokazivač na način da pokazuje na objekat koji je nekoliko memorijskih lokacija udaljen od objekta na koji pokazivač originalno pokazuje. Ovakve aritmetičke operacije sa pokazivačima su često pogodne kada se radi sa nizovima. Pretpostavimo da imamo pokazivač na cjelobrojnu promjenljivu (tipa int) i niz cjelih brojeva (tipa int):

```
int *p, ocjene[5] = { 10, 6, 8, 7, 8},      // Inicijalizacija niza
```

```
p = ocjene;                                // pokazivač na prvi element
```

U prethodnoj liniji smo postavili da pokazivač pokazuje na niz, što znači da će pokazivač u tom slučaju pokazivati na prvi član niza. Dodavanje jedne cjelobrojne vrijednosti pokazivaču ili oduzimanje jedne cjelobrojne vrijednosti od pokazivača dovodi do toga da pokazivač sada pokazuje na adresu koja je za sizeof(tip podatka) pomjerena u odnosu na originalnu poziciju. Odnosno, ako pomjeramo za više cjelobrojnih vrijednosti, kompajler će automatski pomnožiti taj cijeli broj sa veličinom objekta na koji se pokazivač originalno odnosi, kao što pokazuje sledeći primjer:

```
#include<iostream>
using namespace std;
int main()
```

```

{
    int *p, ocjene[5] = { 10, 6,8,7,8 };           // Inicijalizacija niza
    p = ocjene;                                   // Pokazivač na prvi element
    p = p + 1; // Napredovanje pokazivača na sljedeći element u nizu.
    p = 2 + p; // Napredovanje ne mora biti jedan za drugim nego i ovako gdje p sad
pokazuje na ocjene[3].
    cout << *p << endl; // Prikazuje element na koji pokazuje pokazivač.
    cout << *(p - 1) << endl; // Pokazuje na prethodni element bez izmjene pokazivača p
    return 0;
}

```

Izraz `p = p + 1;` dodaje veličinu memorije jednog elemeta niza pokazivaču, pa će stoga `p` pokazivati na sljedeći element niza, `ocjene[1]`. S obzirom da je `p` deklarisan kao pokazivač na `int`, njegova vrijednost će biti uvećana za `sizeof(int)`. Izraz `p = p + 1;` ima isti efekat kao neki od sljedećih operatora dodjele ili inkrementiranja:

```

p += 1;
++p;
p++;

```

### *Višedimenzionalni i 2D nizovi*

2D nizovi su najprostiji oblik višedimenzionalnih nizova i mogu se drugačije opisati kao nizovi nizova

```
type name[size1][size2]...[sizeN];
```

Tako, npr., ako želimo da definišemo niz dimenzija 4x9x3 to možemo uraditi na sljedeći način:

```
int troDim[4][9][3];
```

Najprostiji oblik višedimenzionalnih nizova su dvodimenzionalni nizovi (2D). 2D niz je u osnovi niz od 1D nizova. Osnovni način deklarisanja 2D nizova je:

```
type arrayName [ x ][ y ];
```

gdje `type` predstavlja tip, `arrayName` ime niza, dok se u uglastim zagradama navode dimenzije `X` i `Y` niza (`X` redova i `Y` kolona). 2D niz je ustvari tabela koja ima `X` redova i `Y` kolona. Proizvoljni 2D niz `a`, koji se sastoji iz 3 reda i 4 kolona može biti grafički predstavljen na način kao na slici 3.4.:

|       | Kolona 0 | Kolona 1 | Kolona 2 | Kolona 4 |
|-------|----------|----------|----------|----------|
| Red 0 | a[0][0]  | a[0][1]  | a[0][2]  | a[0][3]  |
| Red 1 | a[1][0]  | a[1][1]  | a[1][2]  | a[1][3]  |
| Red 2 | a[2][0]  | a[2][1]  | a[2][2]  | a[2][3]  |

Slika 3.4. Grafička predstava višedimenzionalnog niza

Svaki element 2D niza određen je indeksom reda i indeksom kolone na sljedeći način  $a[i][j]$ , gdje je A ime niza, dok I i J predstavljaju indekse reda i kolone.

### *Inicijalizacija i pristupanje članovima 2d niza*

Inicijalizacija i pristupanje članovima 2D niza se ostvaruje na sličan način kao i kod 1D niza osim što ovdje imamo jednu dimenziju više.

#### Inicijalizacija dvodimenzionalnog niza

Višedimenzionalni niz može biti inicijalizovan uređenom grupom vrijednosti oivičenim vitičastim zagradama, i svaka grupa predstavlja jedan red. U nastavku je dat primjer niza koji ima 3 reda i u svakom redu vrijednost za jednu od 4 kolona.

```
int a[3][4] = {
    {0, 1, 2, 3}, /* inicijalizacija reda 0 */
    {4, 5, 6, 7}, /* inicijalizacija reda 1 */
    {8, 9, 10, 11} /* inicijalizacija reda 2 */
};
```

Ugnježdene vitičaste zagrade koje uokviruju jedan red su sasvim proizvoljne (ne moraju se pisati), tako da se prethodni primjer može predstaviti i na sljedeći način:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

- Pristupanje članovima 2D niza

Pristupanje članu 2D niza se ostvaruje navođenjem indeksa reda i kolone u uglastim zagrada nakon imena niza, na sljedeći način:

```
int val = a[2][3];
```

pri čemu je iz 2D niza uzeta vrijednost koja se nalazi u 3. redu i 4. koloni, i ta vrijednost je dodijeljena promjenljivoj val. Kao što je opisano u prethodnom dijelu, moguće je generisati nizove sa više dimenzija, ali je najčešća praksa da se koriste 1D i 2D nizovi.

U nastavku je dat primjer sa 2D nizovima gdje je korištena ugnježdjena petlja za učitavanje i štampanje članova niza:

```
int main()
{
    /* 2D niz sa 5 redova i 2 kolone*/
    int a[5][2] = { { 0,0 }, { 1,2 }, { 2,4 }, { 3,6 }, { 4,8 } };
    int i, j;
    /* prikazi vrijednost svakog elementa niza */
    for (i = 0; i < 5; i++)
    {
        for (j = 0; j < 2; j++)
        {
            printf("a[%d][%d] = %d\n", i, j, a[i][j]);
        }
    }
    return 0;
}
```

Nakon izvršavanja prethodnog koda dobija se sljedeći rezultat:

```

a[0][0] == 3
a[0][1] == 4
a[1][0] == 9
a[1][1] == 5
a[2][0] == 7
a[2][1] == 1
Press any key to continue . . .

```

### Višedimenzionalni nizovi i funkcije

Višedimenzionalni nizovi se kroz listu argumenata prosleđuju funkciji na isti način kao i 1D nizovi.

```

#include<iostream>
using namespace std;
void prikazi(int n[3][2]);
int main()
{
    int matrica[3][2] = {
        { 3, 4 },
        { 9, 5 },
        { 7, 1 }
    };
    prikazi(matrica);
    return 0;
}
void prikazi(int n[3][2])
{
    cout << "Dvodimenzionalni niz: \n";
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 2; ++j)
        {
            cout << n[i][j] << "    ";
        }
        cout << endl;
    }
}

```

Višedimenzionalni nizovi se mogu kao formalni parametri proslijediti funkciji na sličan način kao kod 1D nizova. U prethodnom dijelu koda je dat primjer korištenja višedimenzionalnih nizova kao argumenata funkcije. Funkcija prikazi() služi za prikaz elemenata niza.

Kao rezultat, dobija se sljedeći izlaz:

```

Dvodimenzionalni niz:
3 4
9 5
7 1
Press any key to continue . . .

```

Ukoliko kod proslijeđivanja 2D niza kao argumenta funkciji ne navedemo dimenzije niza, kompajler će prijaviti grešku:

```
void prikazi(int n[[]], int , int);
```

```

..
void prikazi(int n[ ][ ], int br, int bk)
{
    cout << "Dvodimenzionalni niz: \n";
    for (int i = 0; i < br; ++i)
    {
        for (int j = 0; j < bk; ++j)
        {
            cout << n[i][j] << "    ";
        }
        cout << endl;
    }
}

```

U ovom slučaju mora se navesti druga dimenzija niza koji se proslijeđuje kao argument funkciji. Ispravan programski kod glasi:

```

#include<iostream>
using namespace std;
void prikazi(int n[ ][2], int , int);
int main()
{
    int matrica[3][2] = {
        { 3, 4 },
        { 9, 5 },
        { 7, 1 }
    };
    prikazi(matrica,3,2);
    return 0;
}
void prikazi(int n[ ][2], int br, int bk)
{
    cout << "Dvodimenzionalni niz: \n";
    for (int i = 0; i < br; ++i)
    {
        for (int j = 0; j < bk; ++j)
        {
            cout << n[i][j] << "    ";
        }
        cout << endl;
    }
}

```

## STRINGOVI

Stringovi NISU primitivni tipovi podataka. Da bismo ih mogli uopće koristiti potrebno je uključiti zasebnu biblioteku.

String je sastavljen od simbola (karaktera - engl. character) koji jesu primarni tip podatka.

Slova poput a,b,c,d... su primarni tipovi podataka. Korištenjem slova stvaramo riječi. Riječ je složeni tip podatka koji je sastavljen od više vrijednosti koje su primarni tipovi podataka.

Stringovi ujedno spadaju pod objektno orijentisano programiranje. Važno je jedino zapamtiti kako se nad stringovima mogu vršiti razne operacije koje ćemo upoznati s vremenom i da je u tom slučaju u naš projekt potrebno uključiti posebnu biblioteku.

String konstante se pišu između dvostrukih navodnika. U programskom jeziku C/C++ string je predstavljen kao jednodimenzionalni niz karaktera koji se završava null karakterom „\0“.

### *String konstante*

String literali ili string konstante u programskom jeziku C/C++ (kao u C# i Javi) se pišu između navodnika “ ”. Kao i u Java-i, string konstanta se može sastojati iz karaktera, znakova interpunkcije, escape i univerzalnih karaktera. U C/C++-u je moguće presjeći dugačku rečenicu korištenjem literala koji su razdvojeni prazninama (whitespace). U nastavku je dat primjer korištenja string literala u C/C++-u.

### *Inicijalizacija stringova*

U programskom jeziku C/C++ string je predstavljen kao jednodimenzionalni niz karaktera koji se završava null karakterom “\0”. U narednoj liniji kôda izvršena je deklaracija i inicijalizacija stringa “Hello”. Treba imati na umu da je dimenzija niza pozdrav 6, jer je posljednje mjesto rezervisano za null karakter “\0”.

```
char pozdrav[6] = {"H", "e", "l", "l", "o", "\0"};
```

Na osnovu opisanog pravila koje važi za null karakter, moguće je prethodni string definisati korištenjem sljedećeg izraza:

```
char pozdrav[] = "Hello";
```

Predstavljanje stringova u memoriji i upotreba

Niz karaktera je kao i običan niz u memoriji smješten na susjednim lokacijama. Ime niza karaktera (stringa) također predstavlja pokazivač na prvi element niza.

Na slici 3.5. je dat grafički prikaz segmenta memorije u kome je smještena riječ “Hello”:

| Indeks       | 0       | 1       | 2       | 3       | 4       | 5       |
|--------------|---------|---------|---------|---------|---------|---------|
| Promjenljiva | H       | e       | l       | l       | o       | \0      |
| Adresa       | 0x22331 | 0x22332 | 0x22333 | 0x22334 | 0x22335 | 0x22336 |

Slika 3.5. Grafička prikaz memorije u kojoj je smještena riječ Hello

Prilikom definisanja string konstanti i literala nije neophodno staviti null karakter na kraj stringa pošto C/C++ kompajler to radi umjesto nas prilikom inicijalizacije stringa. U nastavku je dat C primjer koji ispisuje string "Hello" na standardni izlaz:

```
#include <stdio.h>
int main()
{
    char pozdrav[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };
    printf("Pozdravljam: %s\n", pozdrav);
    /*ne može
    cout << "Pozdravljam opet:" << pozdrav << endl;
    jer nedostaje:
    #include <iostream>
```



```

        using namespace std;
        */
        return 0;
}

```

Nakon izvršenja prethodnog kôda dobija se sljedeći rezultat:

```

04.
Pozdravljam: Hello
Press any key to continue . . .

```

### *Stringovi kao argumenti funkcije*

Stringovi se prosleđuju funkciji na isti način kao i nizovi. U nastavku je dat primjer koji demonstrira korištenje stringa u funkciji kao argumenta funkcije. Može se zaključiti da je isto kao što smo imali kod prosleđivanja 1D niza kao argumenta funkcije.

```

#include <iostream>
using namespace std;
void prikazi(char s[]);
int main()
{
    char str[100];
    cout << "Unesite string: ";
    cin >> str;
    prikazi(str);
    return 0;
}
void prikazi(char s[])
{
    cout << "Unijeli ste:" << s << endl;
}

```

Rezultat prethodnog koda je:

```

04.
Unesite string: Hello
Unijeli ste:Hello
Press any key to continue . . .

```

### *Nizovi stringova*

Nizovi stringova u C++-u su definisani kao 2D nizovi karaktera, pri čemu svaki string predstavlja sekvencu karaktera koja se završava null karakterom. Slijedi primjer niza stringova koji predstavljaju mjesece u godini.

Zašto su u primjeru dolje podvučeni tj. označeni kao netačni pojedini stringovi?

```

#include <iostream>
using namespace std;
#include <string.h>
#define MJESECI 12
#define MAX 5
void main()
{
    int j;
    char godina[MJESECI][MAX] = { "JANUAR", "FEBRUAR", "MART", "APRIL", "MAJ", "JUNI",
                                   "JULI", "AUGUST", "SEPTEMBAR", "OKTOBAR", "NOVEMBAR", "DECEMBAR"};
    for (j = 0; j<MJESECI; j++)
    {
        cout<<godina[j]<<endl;
    }
}

```

Ispravno napisan gornji programski kôd je:

```

#include <iostream>
using namespace std;
#include <string.h>
#define MJESECI 12
#define MAX 5
void main()
{
    int j;
    char godina[MJESECI][MAX] = { "JAN", "FEB", "MART", "APRI", "MAJ", "JUNI",
                                   "JULI", "AUGU", "SEPT", "OKT", "NOV", "DEC"};
    for (j = 0; j<MJESECI; j++)
    {
        cout<<godina[j]<<endl;
    }
}

```

Rezultat:



```

JAN
FEB
MART
APRI
MAJ
JUNI
JULI
AUGU
SEPT
OKT
NOV
DEC
Press any key to continue . . .

```

U prethodnom primjeru, koristili smo dvodimenzionalni niz karaktera koji u stvari predstavlja niz stringova. Ako posmatramo deklaraciju niza godina, prvi indeks predstavlja ukupan broj stringova dok drugi indeks predstavlja maksimalnu dužinu svakog stringa. Vrijednost konstante "MAX" je postavljena na 5. S obzirom da se neke riječi kao npr. "JANUAR" sastoji iz više od 5 karaktera zato je kompajler te riječi podvukao i označio neispravnim. I riječ "APRIL" je bila označena kao nekorektna jer je ona dužine 5 što je previše jer znamo da je posljednje polje ostavljeno za null karakter koji označava kraj stringa (tako da bi u ovom slučaju bilo ispravna riječ "APRI").

## *Pokazivači i stringovi*

Pokazivači na stringove u C++-u su deklarirani kao pokazivači na nizove karaktera. Pomoću funkcije `scanf` nije moguće čitati string korištenjem pokazivačke promjenljive.

Pokazivači na stringove u C++-u su deklarirani kao pokazivači na nizove karaktera, odnosno na prvi član niza. Kada se nekom pokazivaču na string dodijeli neka tekstualna vrijednost, automatski se na kraj tog stringa dodaje null karakter. U nastavku je dat primjer korištenja:

```
#include<iostream>
using namespace std;
int main()
{
    char *ps;
    ps = "HELLO";
    printf("%s\n", ps);
    //ili
    cout << ps << endl;
    return 0;
}
```

U nastavku je dat primjer gdje u cilju učitavanja nekog teksta koristimo definisani niz `nizS`, a zatim preko pokazivača `ps` koji pokazuje na `nizS` učitavamo tekst i prikazujemo na ekranu.

```
#include<iostream>
using namespace std;
int main()
{
    char nizS[10];
    char *ps;
    ps = nizS;
    cout << "Unesite rijec do 10 karaktera:";
    cin >> ps;
    printf("Unijeli ste: %s\n", ps);
    return 0;
}
```