Programiranje I

Algoritmi za pretraživanje i sortiranje

Malo digresije

Dokle smo stigli

- Do sada smo se upoznali sa materijom uglavnom u “rastućem smjeru”, od prostijih ka složenijim pojmovima i konceptima.
- Tokom te priče dosta ne manje važnih detalja smo preskočili kako ne bi ovu (već dosta komplikovanu) priču učinili još komplikovanijom.
- Stoga ćemo iskoristiti ovu lekciju da sistematizujemo do sada izloženo gradivo, te da objasnimo pomoćne koncepte u programskom jeziku C.
- Prije nego se uputimo u tom pravcu obradićemo algoritme za sortiranje i pretraživanje.

Algoritmi za pretraživanje

- U bazama podataka, u aplikacijama za obradu teksta, u brojnim inženjerskim algoritmima, koriste se algoritmi za pretraživanje i sortiranje.
- Cilj algoritama za pretraživanje je najčešće da odrede da li u datom nizu/kolekciji postoji traženi podatak i da vrate njegovu poziciju (indeks).
- Suštinski postoje dva različita algoritma za ovo:
  - brute force algoritam i
  - podjeli pa vladaj (divide&conquer) algoritam.

Brute force algoritam

- Brute force algoritam provjerava svaki član niza redom.
- Data je realizacija funkcije koja ima tri argumenta: pokazivač na niz, broj članova niza i broj koji se u nizu traži (funkcija vraća –1 ako ne pronađe traženi element niza):

```
int brute_force(int *a, int n, int x) {
      int i;
      for(i=0;i<n;i++)
            if(a[i]==x) return i;
      return –1;
}
```

U **najgorem slučaju** vršimo **n** poređenja, a ako element postoji u nizu i ako je jednako vjerovatna bilo koja pozicija onda vršimo u **prosjeku n/2** poređenja.

Složenost algoritama se saopštava ili za najgori slučaj (češće) ili za prosječni (rjeđe).

“Podijeli pa vladaj” algoritam

- Podijeli pa vladaj algoritam polazi od pretpostavke da je niz sortiran.
- Pretpostavimo da je niz sortiran u neopadajući redosljed.
- Ovaj algoritam prvo posmatra srednji član niza (član na sredini niza). Ako je taj član niza jednak broju koji se traži vraća se rezultat, a ako nije provjerava se da li je taj član veći od onog koji se traži. Ako jeste, to znači da se pretraživanje može lokalizovati na “donju” polovinu niza, a u suprotnom na “gornju”.
- Ovakvo pretraživanje se još naziva binarno pretraživanje.

Primjer

- Neka je niz: 1 3 6 10 15 21 28 36 45.
- Neka je element koji se traži 28. Niz ima 9 članova i "srednji" (peti) je 15. Kako je traženi element veći od njega, pretraživanje se nastavlja, ali sada od 6-tog do 9-tog elementa niza. Sada se uzima srednji elemenat tog podniza, to je (6+9)/2=7 u cijelobrojnim vrijednostima, i to je u ovom slučaju traženi broj.
- Koliko nam u ovom slučaju treba pretraživanja u najgorem slučaju? Neka je to neki broj **f(N)** poređenja, gdje je **N** dužina niza.

Složenost binarnog pretraživanja

- Nakon jednog poređenja pretraživanje nastavljamo u jednom od podnizova od približno N/2 članova.
- Dakle, možemo zapisati: $f(N)=1+f(N/2)$.
- Funkcija koja zadovoljava ovu rekurziju je $f(N)=\log_2 N$.
- Na primjer, za N=1024 nam, u ovom algoritmu, treba u najgorem slučaju desetak poređenja, dok u prosječnom slučaju kod brute force algoritma treba 512.
- Naravno, nešto smo platili time što algoritam radi samo ako je niz unaprijed sortiran.

Binarno pretraživanje - sami

- Sami napišite program za algoritam binarnog pretraživanja.
- Napišite i funkciju koja ga realizuje.
- Sami odradite i slučaj niza sortiranog u opadajući redosljed.
- Odradite obije varijante algoritama za pretraživanje ako se sprovode nad podacima stringovima.

Algoritmi za sortiranje

- Cilj algoritama za sortiranje je da od polaznog niza dobijemo niz sa elementima preuređenim tako da rastu, opadaju, ne rastu ili ne opadaju (sortirani niz).
- Sortiranje u neopadajući ili nerastući redosljed podrazumijeva da ima ponavljanja elemenata u nizu.
- Mi ćemo nizove sortirati u neopadajući redosljed (od najmanjeg ka najvećem, moguća ponavljanja).
- Veliki broj algoritama je razvijen za ove namjene:
  - metod ponovljenog minimuma,
  - bubble sort algoritam,
  - insertion sort algoritam,
  - quick sort algoritam, itd.

Metod ponovljenog minimuma

- Obradićemo samo prva dva algoritma na predavanjima.
- Kod metoda ponovljenog minimuma vrši se poređenje prvog člana niza sa svim ostalim. Ako je element na prvoj poziciji veći od elementa sa kojim se poredi to znači da ovi elementi nijesu u dobrom međusobnom odnosu i da treba izvršiti zamjenu.
- Kada se završi poređenje prvog elementa sa svim ostalim zasigurno je prvi elemenat minimalan (odnosno, prvi u sortiranom nizu), pa se dalje operacije obavljaju na podnizu od drugog elementa nadalje, pa od trećeg nadalje itd.

Metod ponovljenog minimuma

```
int main() {
   int n, a[100], temp, i, j;
   scanf("%d",&n);
   for(i=0;i<n;i++)  scanf("%d",a+i);
   for(i=0;i<n-1;i++)
      for(j=i+1;j<n;j++)
        if(a[i]>a[j])
           {temp=a[i]; a[i]=a[j]; a[j]=temp;}
   for(i=0;i<n;i++)
      printf("%d ", a[i]);
}
```

i=0 je iteracija petlje za određivanje minimuma niza, i=1 za sljedeći itd.

provjera da li su elementi neregularno sortirani

ciklus koji tekući elemenat poredi sa ostalim

Metod ponovljenog minimuma

- Svi algoritmi za sortiranje posjeduju dio u kojem se mijenja pozicija članova niza tipa:
  temp=a[i]; a[i]=a[j]; a[j]=temp;
- Cilj ovog koraka je da a[i] i a[j] zamijene mjesta. Zadajte vrijednosti a[i] i a[j] i provjerite.
- Koliko je operacija poređenja potrebno za ovaj algoritam?
- Za i=0 poredi se član sa indeksom i=0 sa svih ostalih n-1 (znači, treba n-1 poređenja), za naredni je to n-2 poređenja itd.

Metod ponovljenog minimuma

- Kod ponovljenog minimuma potrebno je:
  (N-1)+(N-2)+(N-3)+...+3+2+1=N(N-1)/2 poređenja.
- Veći je problem što se poređenja uvijek obavljaju, pa čak i kada je niz u startu sortiran.
- Kao malo unaprijeđenje ponovljenog minimuma, koristi se bubble sort algoritam koji vrši poređenje susjednih elemenata niza i ako naiđe na nesortiranost vrši zamjenu mjesta.
- U prvom prolazu se porede: prvi sa drugim, drugi sa trećim, treći sa četvrtim, ..., pretposljednji sa posljednjim.
- Nakon prvog prolaza posljednji element je zasigurno maksimum, pa se u narednom prolazu vrše poređenja od prvog do pretposljednjeg elementa.

Bubble sort algoritam

- Ako se u bilo kom prolazu ne izvrši nijedna zamjena zaključujemo da je niz sortiran i da nema potrebe nastavljati dalju proceduru.
- Ovdje dajemo samo dio koda koji sortira niz.

```
for(i=n-1;i>0;i--) {
   Ind=1; /*Ind=1 nije došlo do promjene pozicije; Ind=0 jeste.*/
      for(j=0;j<i;j++)
         if(a[j]>a[j+1]) {
            temp=a[j]; a[j]=a[j+1]; a[j+1]=temp;
            Ind=0; /*Došlo je do promjene pozicije*/
         }
   if(Ind==1) break;
}
```

Bubble sort algoritam

- Ključni element u algoritmu je indikator **Ind** koji se u svakoj iteraciji postavlja na **1**.
- Jedina pozicija gdje se događa promjena ovog indikatora je u selekciji gdje se vrši zamjena mjesta članova niza.
- Ako vrijednost indikatora **Ind** do kraja programske petlje ostane **1** to ukazuje da je ostatak niza koji se trenutno ispituje sortiran i da nema potrebe za daljim poređenjima.
- Ipak, u najgorem slučaju složenost ovog algoritma ostaje **N(N-1)/2**.

Ostali algoritmi za sortiranje

- Na vježbama ćete se upoznati sa drugim algoritmima sortiranja. Naravno, ovdje se ne završava priča o sortiranju.
- Za samostalan rad! Kreirajte sve predmetne algoritme sortiranja za sortiranje u nerastući redosljed (najčešće je dovoljno zamijeniti samo jedan operator. Koji?). Algoritme odraditi i kao program i kao funkcije.
- Odradite varijante za sortiranje stringova.

typedef

- Vidjeli smo da tipovi podataka u C-u mogu imati izuzetno dugačke nazive, npr. unsigned short int.
- Naredbom typedef se može izvršiti definisanje novog imena za postojeće tipove podataka. Na primjer:
  typedef unsigned short int UISHORT;
- Ovo se radi prije prvog korišćenja tipa, a najbolje van svih funkcija.
- Od sada nadalje se promjenljive umjesto unsigned short int mogu deklarisati kao:
  UISHORT a,b;
- Ključnu riječ typedef opravdano je koristiti kod struktura, ali o tom-potom.

Modifikator const

- "Promjenljiva" koja je deklarisana kao:

  const int i=0;

  je konstanta čiji pokušaj promjene uzrokuje grešku.
- Smisao ovog modifikatora je da zabrani promjenu ove konstante u programu.
- Zamislimo sljedeći slučaj. Imamo promjenljivu a=12.345 koja nam ulazi u mnoštvo proračuna. Umjesto da je pišemo na svakom mjestu, možemo je deklarisati jednom, a kako je njen smisao da se nikad ne mijenja pogodno ju je deklarisati kao konstantu.

Konstantni pokazivači

- Sljedeće deklaracije:

  ```
  const int *i;
  int * const i;
  const int * const i;
  ```

  deklarišu promjenljivu **i** koja je pokazivač na cijeli broj, dok je ***i** vrijednost te promjenljive.

- Tumačenje ovih deklaracija se obavlja na sljedeći način.
  - Prvo se uklone svi **const** da bi se shvatilo što je promjenljiva koja se deklariše (u sva tri slučaju **i** je pokazivač, a ***i** je promjenljiva).
  - Zatim se pogleda što je sa desne strane od modifikatora **const**.
  - U prvom slučaju je konstantan cijeli broj, u drugom slučaju je konstantan pokazivač, dok su u trećem slučaju i promjenljiva i pokazivač konstantni.

Konstantni argumenti funkcije

- Jedini način da se niz proslijedi kao argument funkcije je putem pokazivača.
- Kad nešto proslijedimo putem pokazivača funkciji ne postoji zabrana da ta funkcija izmijeni promjenljivu na koju pokazivač pokazuje.
- Često cilj nije izmjena argumenta funkcije (npr. stringa), već određivanje neke karakteristike tog argumenta.
- U tom slučaju bismo željeli da funkcija ne može da promjeni vrijednost argumenta.

Konstantni argumenti funkcije

- Takav argument funkcije se može deklarisati kao:
  int fun(const char *s) {}
- Sada bi unutar tijela funkcije s[i]='A' dovelo do greške i prekida programa.
- Napominjemo da postoje trikovi koji omogućavaju promjenu i konstantnih promjenljivih, pokazivača, a i konstantnih argumenata funkcije. Ovi trikovi se razlikuju od kompajlera do kompajlera. Jedan ovakav trik je:

```
const int x=7;
*(int *)&x = 66;
```

Kastovanjem adrese dobijamo pokazivač za koji ne važi const ograničenje.

- Napominjemo da je cilj modifikatora **const** da zabrani slučajno ugrožavanje konstantosti, a ne situacije da neko nasilnim putem ugrožava konstantnost.

Pretprocesor

- Pretprocesor predstavlja dio koda u programskom jeziku C kojem kompajler pristupa prije nego počne da procesira **(odakle mu i naziv)** glavninu koda.
- Sve pretprocesorske naredbe (odomaćio se termin direktive umjesto naredbe) počinju karakterom **#**, pa se stoga često nazivaju tarabama.
- Sa pretprocesorskom direktivom #include smo se već djelimično upoznali. U obliku:
  #include <stdio.h>
  nam je služila da u programski kôd uključimo sadržaj standardnih programskih biblioteka.

Pretprocesor

- Pretprocesorska direktiva #include može da posluži za uključivanje u kôd funkcija koje smo mi ili neki drugi programeri kreirali. Sintaksa je tada nešto drugačija: #include "nase_fun.h"
- Najpoznatija pretprocesorska direktiva je #define.
- Na osnovu njene česte upotrebe lako se prepoznaju stari C programeri koji su prešli na druge programske jezike.

#define

- Tri osnovne primjene pretprocesorske direktive #define su:
  - makrozamjena,
  - makrorazvoj i
  - makrodefinicija.
- Primjer makrozamjene je:
  #define MAX 100
- Pojavljivanje stringa MAX se u tekstu programa, na primjer:

  int a[MAX];

  mijenja sa stringom 100.
- Ovakav oblik je pogodan ako radimo sa nizovima i matricama sa istom maksimalnom dimenzijom. Promjena te dimenzije se obavlja na jednom mjestu pomoću makrozamjene.

#define

- Pošto se makrozamjena obavlja prije početka kompajliranja, to ne postoji mogućnost za provjeru ispravnosti zamijenjenog teksta.
- Kažemo da makrozamjena mijenja tekst “naslijepo”. Na primjer, ako u tekstu programa piše MAXMAX biće zamjenjeno sa 100100 bez obzira da li je to ono što je korisnik želio.
- Primjena makrorazvoja je u kreiranju jednostavnih funkcija.
- Naime, funkcija koja sabira dva broja je izuzetno jednostavna, ali operacije sa stekom i alokacionim zapisom čine da se vrijeme troši i na poziv ovakve funkcije.

#define

- Umjesto da se piše funkcija koja sabira dva broja može se definisati makrorazvoj:
  #define zbir(a,b) a+b
- Sada je poziv z=zbir(x,y); u glavnom programu ekvivalentan z=x+y;
- I makrorazvoj je promjena na slijepo. Tako z=zbir(4,2)/2; ne daje očekivani rezultat 3, već je ekvivalentan: z=4+2/2;
- Da bi se ovaj problem prevazišao treba definisati makrorazvoj:
  #define zbir(a,b) ((a)+(b))

#define

- #define može da posluži i za makrodefiniciju:
  #define MAK
- Provjera da li je makro definisan se obavlja sa:

```
#ifdef MAK
    neke direktive
#endif
```

svaki if se kod pretprocesora završava sa #endif; unutar ovog bloka se mogu naći druge pretprocesorske direktive

- Alternativa je pretprocesorska direktiva: #ifndef MAK koja je zadovoljena ako predmetni makro nije definisan.
- Pored ovoga, postoji i klasični

```
#if konstanti izraz
    blok
#endif
```

Uslovno izvršavanje

- Ako je konstantni izraz u #if dijelu tačan izvršava se blok naredbi koji slijedi; mogu postojati i #elif blokovi **(umjesto else if)** kao i #else dio. I ovaj oblik se završava sa #endif.
- Kada prestane potreba za makroom MAK njegovo važenje do kraja fajla se može ukinuti sa: #undef MAK.
- Razlozi za korišćenje makrodefinicije su u potrebi da se prilikom uključivanja više programskih biblioteka izbjegne višestruko najavljivanje iste funkcije što bi dovelo do prekida izvršavanja programa.
- Značaj je kod rada sa velikim programskim paketima i mi, početnici u ovoj oblasti, nećemo ulaziti u te detalje.

Ostale direktive pretprocesora

- Ako se u pretprocesoru naiđe na naredbu:
  #error string
  program se prekida i štampa se upozorenje dato string-om.
- Korišćenje direktive #pragma je ostavljeno na diskreciju kompajleru i svaki kompajler joj daje drugo značenje.
- Ovim nijesu iscrpljene opcije pretprocesora, ali za nas nije od posebnog značaja.

```
#include<stdio.h>
#ifndef MAK
    #error Nije definisan…
#else
    int main(){
            …
    }
#endif
```