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 uvedemo i objasnimo dodatne 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 pretraga i
  - binarna pretraga.

Brute force pretraga

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

```
int bf_pretraga(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. Ako element postoji u nizu i ako su sve pozicije jednako vjerovatne, u **prosjeku** vršimo **n/2** poređenja.

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

Binarna pretraga

- Binarna pretraga polazi od pretpostavke da je niz sortiran.
- Pretpostavimo da je niz sortiran u neopadajući redosljed.
- Prvo posmatramo element na sredini niza. Ako je taj element jednak traženom broju, vraća se pozicija sredine niza, i to je kraj pretrage. Ako nije, provjerava se da li je element na sredini veći od traženog. Ako jeste, to znači da se pretraživanje može lokalizovati na lijevu polovinu niza, a u suprotnom na desnu.
- Ova pretraga pripada porodici podijeli pa vladaj (eng. divide and conquer) algoritama.

Primjer

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

Složenost binarne pretrage

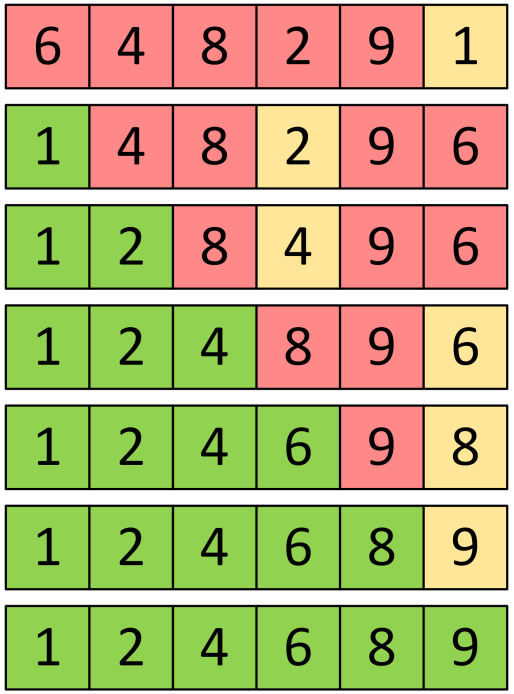
- Nakon jednog poređenja, pretragu nastavljamo u jednom od podnizova od približno N/2 članova.
- Dakle, možemo zapisati $f(N) = 1 + f(N/2)$.
- Funkcija koja zadovoljava ovu rekurziju je $f(N) = \log_2 N$.
- Na primjer, za N=1024 nam, u ovom algoritmu, treba u najgorem slučaju desetak poređenja, dok u prosječnom slučaju kod brute force algoritma treba 512.
- Naravno, sortiranje je cijena koju smo platili da bismo koristili binarnu pretragu.

Algoritmi za sortiranje

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

Metod ponovljenog minimuma

- Obradićemo samo prva dva algoritma na predavanjima.
- Kod metode ponovljenog minimuma, poznate i pod nazivom selection sort:
  - u prvom prolasku kroz niz se odredi minimum niza i on se postavi na prvo mjesto u nizu;
  - u drugom prolasku se odredi minimum podniza od drugog do posljednjeg elementa i on se postavi na drugom mjesto u nizu;
  - u trećem prolasku se odredi minimum podniza od trećeg do posljednjeg elementa i on se postavi na treće mjesto u nizu, itd.



Metod ponovljenog minimuma

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

i=0 je iteracija petlje za određivanje minimuma čitavog niza, i=1 je iteracija za podniz od drugog do posljednjeg elementa itd.

Postavljanje minimuma tekućeg podniza na pravu poziciju

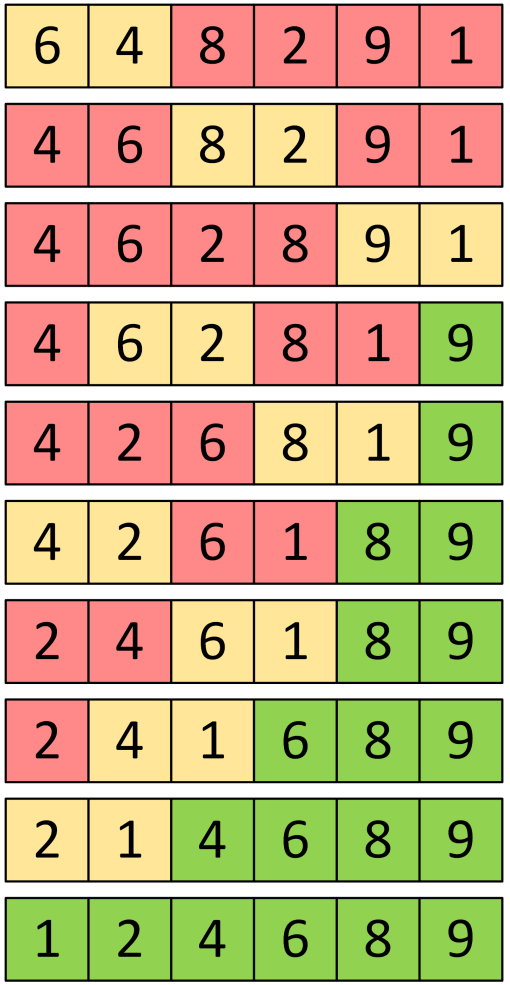
Metod ponovljenog minimuma

- Svi algoritmi za sortiranje posjeduju dio u kojem se mijenja pozicija članova niza:
  temp = a[i]; a[i] = a[ind_min]; a[ind_min] = temp;
- Cilj ovog koraka je da a[i] i a[ind_min] zamijene mjesta.
- Koliko je operacija poređenja potrebno za ovaj algoritam?
- Za i=0 se poredi član sa indeksom i=0 sa ostalih n-1 članova (n-1 poređenja), za naredni imamo n-2 poređenja itd.
- Kod ponovljenog minimuma potrebno je:

  (N-1)+(N-2)+(N-3)+...+3+2+1 = N(N-1)/2 poređenja.
- Problem kod metode ponovljenog mininuma je što se poređenja uvijek obavljaju, čak i kada je niz u startu sortiran.

Bubble sort

- Kao malo unaprjeđenje 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, ..., pretposljednji sa posljednjim.
- Nakon prvog prolaza posljednji element je maksimum.
- U narednom prolazu elementi se porede od prvog do pretposljednjeg.



Bubble sort algoritam

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

```
for(i=n-1; i>0; i--) {
    ind = 1;    // ind=1 nije došlo do zamjene elemenata; 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 zamjene elemenata
        }
    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 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 tome više riječi kasnije.

const kvalifikator tipa

- "Promjenljiva" koja je deklarisana na jedan od ovih načina:

  const int i = 0;
  int const i = 0;

  je konstanta čiji pokušaj promjene uzrokuje grešku.

- Smisao ovog kvalifikatora tipa (eng. *type qualifier*) 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 ne mijenja, treba je proglasiti konstantom.

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 kvalifikatora **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 izmjeni promjenljivu na koju pokazivač pokazuje.
- Često cilj nije izmjena argumenta funkcije (npr. stringa), već određivanje neke karakteristike tog argumenta.
- U tom slučaju, ne želimo da funkcija mijenja vrijednost argumenta. Takav argument funkcije se može deklarisati kao:

```
int fun(const char *s) {}
```

- Pokušaj izvršenja `s[i] = 'A'` bi doveo do greške i prekida programa.

Promjena konstante

- Postoje trikovi koji omogućavaju promjenu i konstantnih promjenljivih, pokazivača, a i konstantnih argumenata funkcije. Jedan ovakav trik je:

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

Cast-ovanje vršimo jer &x vraća adresu tipa const int.

- I ovdje postoje izuzeci. Ako je x globalna konstanta, pokušaj promjene vrijednosti preko pokazivača bi vjerovatno doveo do greške.
- U mnogim kompajlerima, globalne konstante se smještaju u read-only dijelu memorije.
- Napominjemo da je cilj **const** kvalifikatora tipa da zabrani slučajno ugrožavanje konstantosti, a ne situacije da neko nasilnim putem ugrožava konstantnost.

Pretprocesor. #include

- Pretprocesor predstavlja dio kôda u programskom jeziku C kojem kompajler pristupa prije nego počne da procesira (odakle mu i naziv) glavninu kôda.
- Sve pretprocesorske naredbe (direktive) 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.
- Za uključivanje u program funkcija koje smo mi ili neki drugi programeri kreirali, sintaksa je #include "nase_fun.h".

#define

- Tri osnovne primjene pretprocesorske direktive #define su:
  - makro-zamjena,
  - makro-razvoj i
  - makro-definicija.
- Primjer makro-zamjene 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 makro-zamjene.

#define

- Pošto se makro-zamjena vrši prije kompajliranja, ne postoji mogućnost za provjeru ispravnosti zamijenjenog teksta.
- Kažemo da makro-zamjena 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 makro-razvoja 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 makro-razvoj:
  #define zbir(a,b) a+b
- Sada je poziv z=zbir(x,y); u glavnom programu ekvivalentan z=x+y;
- I makro-razvoj 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 makro-razvoj:
  #define zbir(a,b) ((a)+(b))

#define

- #define može da posluži i za makro-definiciju:
  #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 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, 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 makro-definicije 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
```