

Programiranje 1

10. predavanje — dodatak

Saša Singer

PMF – Matematički odsjek, Zagreb

Sadržaj predavanja — dodatka

- Primjeri funkcija za neke probleme:
 - Prosti brojevi.
 - Prosti faktori broja.
 - Binomni koeficijenti i Pascalov trokut.
 - Obično i binarno potenciranje.

Prosti brojevi i prosti faktori broja

Sadržaj

- Prosti brojevi:
 - Definicija prostog broja.
 - Provjera je li broj prost — primjeri “ne tako!”.
 - Bolji algoritmi i pripadne funkcije.
- Prosti faktori broja:
 - Teorem o rastavu na proste faktore.
 - Nalaženje prostih faktora broja — po definiciji.
 - Bolji algoritmi i pripadne funkcije.

Definicija prostog broja

Definicija. Prirodni broj $n \geq 2$ je **prost** ako i samo ako je

- djeljiv **samo** s **1** i sa **samim sobom**, tj.
- ima samo **trivijalne** djelitelje.

Broj $n = 1$ **nije** prost, po definiciji.

Razlog: zato da vrijedi teorem

- o **jednoznačnom** rastavu broja $n \geq 2$ na **proste faktore**.

Zadatak. Napisati dio programa, odnosno, funkciju

- za **provjeru** je li učitani (ili zadani) broj **prost** ili **ne**.

Napomena: radimo u tipu **int** i **ignoriramo** brojeve $n \leq 0$.

Za početak, par “**bisera**” — **viđenih** na praktičnom kolokviju!

Provjera “je li broj prost” — ne tako! (1)

Biser 1. Brojimo **sve** različite djelitelje broja n i onda

$$n \text{ je prost} \iff \text{broj djelitelja} = 2.$$

Pripadni dio programa izgleda ovako (v. [pr_lose_1.c](#)):

```
int n, d, broj_dj = 0;
...
for (d = 1; d <= n; ++d)
    if (n % d == 0) ++broj_dj;

if (broj_dj == 2)
    printf(" Broj %d je prost\n", n);
else
    printf(" Broj %d nije prost\n", n);
```

Provjera “je li broj prost” — ne tako! (2)

Biser 2. Zbrojimo(!) sve različite djelitelje broja n i onda

$$n \text{ je prost} \iff \text{suma djelitelja} = n + 1.$$

Pripadni dio programa izgleda ovako (v. `pr_lose_2.c`):

```
int n, d, suma_dj = 0;
...
for (d = 1; d <= n; ++d)
    if (n % d == 0) suma_dj += d;

if (suma_dj == n + 1)
    printf(" Broj %d je prost\n", n);
else
    printf(" Broj %d nije prost\n", n);
```

Poboljšanje = netrivialni djelitelji, prekid petlje

Tko zna otkud takva rješenja!? **Komentari:**

- Trivijalne djelitelje 1 i n **nema** smisla “testirati”, kad su **sigurno** djelitelji broja!
- Brojanje ili zbrajanje **svih** djelitelja je **besmisleno**, čak i kad je broj **prost**,
- a za **složene** brojeve — petlju možemo **prekinuti** čim nađemo **prvi netrivialni** djelitelj broja!

Dakle, u petlji testiramo

- sve moguće **netrivijalne** djelitelje, od 2 do $n - 1$,
- petlju **prekidamo** čim nađemo na **prvi** takav, ako postoji.

Varijabla **prost** je odgovor na pitanje “je li broj prost”.

Varijanta 1 — Varijabla `prost` u uvjetu petlje

Varijabla `prost` je u **uvjetu** petlje da osigura prekid.
Pripadni dio programa (v. `pr_1.c`):

```
int n, d, prost;
...
prost = (n >= 2);    /* Inicijalizacija. */

d = 2;
while (prost && d < n) {
    if (n % d == 0) prost = 0;
    ++d;
}
```

Mana = **stalno** testiranje varijable `prost` za nastavak petlje.

Varijanta 2 — petlja `while` s prekidom petlje

Umjesto toga, iskoristimo naredbu `break` za prekid petlje. Pripadni dio programa (v. `pr_2.c`):

```
int n, d, prost;
...
prost = (n >= 2);    /* Inicijalizacija. */

d = 2;
while (d < n) {
    if (n % d == 0) {
        prost = 0; break;
    }
    ++d;
}
```

Varijanta 3 — petlja `for` s prekidom petlje

Umjesto `while`, koristimo `for` petlju — za **kraći** tekst. Zbog definicije `for` preko `while`, izvršavanje traje **isto** kao i s `while`. Pripadni dio programa (v. `pr_3.c`):

```
int n, d, prost;
...
prost = (n >= 2);    /* Inicijalizacija. */

for (d = 2; d < n; ++d)
    if (n % d == 0) {
        prost = 0;  break;
    }
```

Funkcija `prost_1` — *svi netrivialni djelitelji*

Realizacija funkcijom — umjesto `break`, odmah se `vratimo` s `return` (v. `pr_fun_1.c`):

```
int prost_1(int n)
{
    int d;    /* Potencijalni djelitelj. */

    if (n <= 1) return 0;

    for (d = 2; d < n; ++d)
        if (n % d == 0) return 0;

    return 1;
}
```

Komentar i poboljšanje

Nažalost, ovaj algoritam je **vrlo spor** baš za **proste** brojeve n .

U čemu je **problem**? Ako je n **prost**, onda testiramo

- **sve** moguće netrivialne djelitelje — a njih je $n - 2$,
- za zaključak da n **nema** netrivialnih djelitelja.

Međutim, to možemo zaključiti i mnogo **ranije**!

Naime, broj n je **složen** ako i samo ako se može prikazati kao **produkt 2** netrivialna djelitelja — a i b , i neka je $a \leq b$.

Dakle, n je **složen** ako i samo ako

- **postoje** prirodni brojevi a i b , takvi da je $2 \leq a \leq b$ i vrijedi $n = a \cdot b$.

Za početak, odavde slijedi $4 \leq n$, za najmanji složeni broj n .

Gornja granica za manji djelitelj

Međutim, iz $n = a \cdot b$ i $a \leq b$ odmah slijedi da je

$$a^2 \leq n, \quad \text{odnosno,} \quad a \leq \sqrt{n}.$$

Ono što je **bitno** — dobili smo

- **gornju** granicu za **manji** od dva djelitelja **složenog** broja.

Ta granica za a je **samo** \sqrt{n} , a **ne** više $n - 1$ (ili $n/2$).

Možda nije sasvim očito, ali **korist** od toga je **trenutna**,

- uz propisnu interpretaciju petlje u prethodnom algoritmu.

Algoritam treba zamisliti

- kao provjeru **složenosti** broja n (uz **prost** = **!slozen**),

- a petlja po d = traženje **najmanjeg** djelitelja a .

Dakle, za **gornju** granicu te petlje možemo uzeti \sqrt{n} .

Korištenje korijena i zaokruživanje

Preciznije, **gornju** granicu za djelitelje treba **jednom** izračunati i spremiti **prije** petlje — na primjer, u varijablu **max_d**

```
max_d = (int) sqrt(n);
```

zato da se **izbjegne** stalno računanje korijena u uvjetu petlje.

Problem: u **realnoj** aritmetici dolazi do grešaka **zaokruživanja!**

Stvarno **opasan** je samo sljedeći slučaj:

- broj n je **potpun kvadrat**, tj. $n = a^2$,
- a funkcija **sqrt** napravi malu grešku **nadolje**, tako da dobijemo $\text{max_d} = a - 1$.

Za konkretni **prevoditelj** i **biblioteku** ovo treba **provjeriti!**

Provjera zaokruživanja korijena i popravak

Na Intel C prevoditelju s Microsoftovom bibliotekom za `sqrt`,
to se **nikad** ne događa, na **cijelom** tipu `int`.

Pogledajte rezultate test-programa `t_isqrt.c`, `t_isqrt1.c`.

Ako se takva greška **zaokruživanja** može dogoditi, onda imamo dvije mogućnosti.

1. Malo “**povećamo**” n pod korijenom i računamo

$$\text{max_d} = (\text{int}) \sqrt{n \cdot (1 + cu)},$$

gdje je u jedinična greška zaokruživanja, a c je neka mala konstanta (na primjer, između 2 i 4).

2. **Izbjegnemo** realnu aritmetiku — tako da u petlji, umjesto testa $d \leq \text{max_d}$, koristimo test $d * d \leq n$.
Mana = **stalno** kvadriranje potencijalnog djelitelja d .

Funkcija `prost_2` — svi djelitelji do \sqrt{n}

Pripadna funkcija (v. `pr_fun_2.c`):

```
int prost_2(int n)
{
    int d, max_d;

    if (n <= 1) return 0;

    max_d = (int) sqrt(n);
    for (d = 2; d <= max_d; ++d)
        if (n % d == 0) return 0;

    return 1;
}
```

Funkcija `prost_3` — *parnost i neparni do \sqrt{n}*

Još **bolje** — **prvo** testiramo **parnost**, a zatim idemo samo po **neparnim** djeliteljima do \sqrt{n} (v. `pr_fun_3.c`):

```
int prost_3(int n)
{
    int d, max_d;

    if (n <= 1) return 0;
    if (n % 2 == 0) return (n == 2);
    max_d = (int) sqrt(n);
    for (d = 3; d <= max_d; d += 2)
        if (n % d == 0) return 0;
    return 1;
}
```

Usporedba složenosti — vremena

Za jako male brojeve n — razlika u brzini se ne vidi.

Međutim, uzмимо malo veći broj n , na primjer

• $n = 1\,000\,000\,007$ — najmanji prosti broj veći od 10^9 .

Onda su razlike u trajanju vidljive:

• $\text{prost}_1(n)$ — 2.573 s (proporcionalno s n),

• $\text{prost}_2(n)$ — 0.000081 s (proporcionalno s \sqrt{n}),

• $\text{prost}_3(n)$ — 0.000041 s (još dvostruko brže).

Zadnje dvije funkcije su prebrze za štopericu, pa je cijeli račun ponovljen 10000 puta, a vrijeme je podijeljeno s tim faktorom.

Za detaljniju usporedbu ove tri funkcije pogledajte directory **VRIJEME**.

Tablica prostih brojeva

Program `pr_tab.c` ispisuje tablicu svih prostih brojeva

- do zadanog broja `max_p = 100 000`,

po ugledu na zadnju funkciju `prost_3` — prvo ispiše 2, a zatim testira samo neparne brojeve tom istom funkcijom (zato nema bitnog ubrzanja obzirom na `prost_2`).

Rezultat osvane trenutno!

Međutim, za veće granice `max_p`, problem postaje

- veličina ispisa, odnosno, izlazne datoteke.

Za jako velike (prikazive) granice, problem postaje i vrijeme!

Teorem o broju prostih brojeva

Laički rečeno, razlog tome je da prostih brojeva ima

● relativno mnogo, iako postaju sve rjeđi.

Neka je $\pi(n)$ = broj prostih brojeva manjih ili jednakih n .

Poznati teorem o prostim brojevima kaže da je

$$\pi(n) \approx n / \ln(n), \quad \text{za jako velike } n.$$

Na primjer, $\pi(10^9) = 50\,847\,534$.

(v. https://en.wikipedia.org/wiki/Prime_number_theorem).

Oдавде vidimo da je broj svih prikazivih prostih brojeva u tipu `int` na 32 bita

$$\pi(\text{INT_MAX} = 2\,147\,483\,647) \approx 100\,000\,000.$$

Broj prikazivih prostih brojeva — i oprez

Program `pr_svi.c` nalazi točan broj svih prikazivih prostih brojeva u tipu `int` i ispisiuje $\pi(\text{INT_MAX})$.

- Ovdje treba biti oprezan s gornjom granicom petlje,
- tako da `n += 2` daje prikazivi rezultat!

Zato se `INT_MAX` provjerava posebno, nakon petlje.

Samo oprez, prije nego što probate izvršiti taj program,

- budite svjesni da će to potrajati ...

Evo zašto — zbog rezultata na puno “manjem” testu:

- Traženje samo prvih 10 000 prostih brojeva većih od 10^9 već traje oko 0.490 sekundi.

(v. `VRIJEME\primes_test.out`, zadnji blok).

Broj prikazivih prostih — rezultat i trajanje

Cijeli posao u `pr_svi` traje okruglo 10 000 puta **dulje**, tj.

🕒 oko 5000 s ili **preko 80** minuta!

Zato **prvo** probajte “**mali**” test, a onda odlučite o “**cijelom**”.

Na **starom** kompu, “**mali**” test je trajao oko 2.2 sekunde, tj.

🕒 “**cijeli**” posao bi potrajao oko 6 sati!

To **nisam** probao, pa **nije bilo** rezultata.

Međutim, **novi** komp je preko 4 puta **brži**, pa nisam odolio :-)

Rezultat:

$$\pi(\text{INT_MAX} = 2\,147\,483\,647) = 105\,097\,565.$$

Stvarno trajanje: 4937.111 s (v. `VRIJEME\primes_svi.out`).

Složenost provjere “je li broj prost”

Napomena. Ovdje opisani algoritmi imaju složenost (trajanje)

- reda veličine \sqrt{n} (ili čak n).

To nisu polinomni, već eksponencijalni algoritmi, jer je

- duljina ulaza (= zapisa broja n) proporcionalna s $\log n$.

Tek 2002. godine pronađen je prvi polinomni algoritam za provjeru “je li n prost”. Složenost takvog algoritma je

- reda veličine $p(\log n)$, gdje je p neki fiksni polinom.

Prvi p je bio stupnja 12, a do danas je stupanj smanjen na 6.

- Ovi brzi algoritmi koriste vrlo ozbiljnu teoriju brojeva!

Pogledati https://en.wikipedia.org/wiki/Primality_test.

Prosti faktori broja

Teorem o rastavu na proste faktore

Teorem. Svaki prirodni broj $n \geq 2$ može se **jednoznačno** napisati u obliku produkta **prostih** faktora

$$n = p_1^{e_1} \cdot p_2^{e_2} \cdot \dots \cdot p_k^{e_k},$$

gdje su $2 \leq p_1 < p_2 < \dots < p_k$ međusobno **različiti prosti** brojevi, a $e_1, e_2, \dots, e_k > 0$ su pripadni prirodni **eksponenti**.

Zadatak. Napisati dio programa, odnosno, funkciju koja

- za učitani (ili zadani) broj $n \geq 2$,
- ispisuje, broji, zbraja, množi, \dots , (tj. “**obrađuje**”)
- sve **različite** proste faktore broja n , s tim da svaki faktor obrađuje točnom **jednom**.

Napomena: radimo u tipu **int** i **ignoriramo** brojeve $n \leq 0$.

Što je cilj?

Cilj: Napraviti ćemo nekoliko (5) sve **boljih** algoritama za rješenje ovog zadatka.

Ti algoritmi **minimalno** ovise o **vrsti obrade** prostih faktora, bitno je samo

- da se svaki **različiti** faktor p_i obrađuje točno **jednom**.

Obrada = **ispis** (pogledati vježbe, zadaci 8.5.3. i 10.1.10.):

- Programi i funkcije nalaze se u directoryju **ISPIS**.

Za testiranje, **ulazni** broj je $n = 4814451 = 3^3 \cdot 17^2 \cdot 617$, a **izlaz** svih programa je **3 17 617** — kako i treba.

Međutim, za prikaz **složenosti** i mjerenje **vremena**, puno bolje je uzeti **obrada** = **brojanje faktora**, jer ne ovisi o brzini pisanja.

Algoritam 1 — provjera “po definiciji”

Algoritam 1. Testiramo **sve** moguće faktore d broja n , koji mogu biti **prosti**, tj.

- idemo od **2** (najmanji prosti broj), pa **sve** do n ,
- provjerimo je li d **faktor** od n i je li d **prost** broj.

Za provjeru je li d **prost**, koristimo funkciju `prost_3`,

- kao **najbržu** od svih ranijih funkcija.

Još jedna napomena — vrijedi za **sve** funkcije u nastavku:

- Broj $n = 1$ **nema** rastav na proste faktore.
- Radimo u tipu `int` i **ignoriramo** sve brojeve $n \leq 0$.

To testiramo odmah na **početku** funkcije

- i **vratimo** rezultat `nula` — kao signal za “grešku”!

Algoritam 1 — nastavak

Funkcija za broj različitih prostih faktora (v. `pfbr_f_1.c`):

```
int broj_prost_fakt_1(int n)
{
    int br_pfakt = 0, d;

    if (n <= 1) return 0;

    for (d = 2; d <= n; ++d)
        if (n % d == 0 && prost_3(d))
            ++br_pfakt;    /* Obrada d, jednom. */

    return br_pfakt;
}
```

Algoritam 1 — komentari

Uočite da kod provjere

- je li d faktor od n i je li d prost broj,
- koristimo skraćeno računanje logičkih izraza,

tako da se `prost_3(d)` poziva samo kad je d faktor od n .

Unatoč tome, mana ovog algoritma je `sporost`, slično kao kod funkcije `prost_1`. Kao i tamo,

- testiramo sve moguće netrivialne faktore — a njih je sad $n - 1$, jer testiramo i n (može biti prost).

Ovome bi se moglo “doskočiti”, tako da uočimo sljedeće:

- ako n ima barem dva različita prosta faktora, onda je najveći prosti faktor najviše jednak $n/2$.

Dakle, skratimo petlju do $n/2$, a iza provjerimo je li n prost.

Algoritam 1 — komentari i poboljšanje

Međutim, **ne isplati** se — bar ne tu, jer cijeli algoritam ima još puno **veću manu**:

- kad jednom **nađemo** neki **prosti** faktor $d = p_i$, onda testiramo i **sve** njegove **višekratnike** do n ,

što je potpuno **nepotrebno** — jer sigurno **nisu prosti**!

Stvarno, ovom algoritmu **fali** malo “**matematike**”.

Za početak, iz rastava broja $n \geq 2$ na proste faktore $n = p_1^{e_1} \cdot p_2^{e_2} \cdot \dots \cdot p_k^{e_k}$, odmah vidimo da vrijedi:

- ako je d **najmanji** faktor (djelitelj ≥ 2) broja n ,
- onda d mora biti **prost** broj, tj. vrijedi $d = p_1$.

Dakle, **ne treba** provjeravati je li d prost — **mora** biti!

Poboljšanje = eliminacija prostog faktora

Ostaje još pitanje kako izbjeći

- testiranje svih višekratnika od $d = p_1$, koji su $\leq n$.

No, i to je jednostavno.

Onog trena kad smo našli najmanji prosti faktor $d = p_1$,

- treba ga “eliminirati” iz broja,
- tako da n dijelimo s d , sve dok je n djeljiv s d .

Broj ovih dijeljenja je upravo jednak eksponentu e_1 , pa tako (brojanjem dijeljenja) nalazimo e_1 , ako nam treba.

Sjetite se provjere je li “ n potencija od d ”,

- tamo smo radili isto, samo je d bio unaprijed zadan!

Algoritam 2 — provjera “dijeljenjem”

Nakon eliminacije prostog faktora $d = p_1$, dobivamo broj

$$n_2 = p_2^{e_2} \cdots p_k^{e_k}.$$

Znamo da je $n_2 < n$ i još **znamo** da je $p_2 > p_1$, pa

🔴 **nastavimo** potragu **najmanjeg** faktora, počev od $p_1 + 1$.

Naravno, broj n_2 je uredno spremljen u istoj varijabli n .

Kad ga nađemo, **najmanji** faktor d (djelitelj ≥ 2) broja n_2

🔴 je sigurno **prost** broj $d = p_2$.

Obradimo ga i eliminiramo **dijeljenjem** iz n_2 , što daje broj n_3 .

I tako redom, ...

🔴 sve dok ne nađemo i eliminiramo **sve** proste faktore.

Na kraju dobijemo $n_{k+1} = 1$, tj. ponavljamo sve dok je $n > 1$.

Algoritam 2 — nastavak

Funkcija za broj različitih prostih faktora (v. [pfbr_f_2.c](#)):

```
int broj_prost_fakt_2(int n)
{
    int br_pfakt = 0, d;

    if (n <= 1) return 0;
```

Algoritam 2 — nastavak

```
for (d = 2; n > 1; ++d)
    if (n % d == 0) {
        ++br_pfakt;    /* Obrada d, jednom. */

        do    /* Dijeli n s d, sve dok mozes. */
            n /= d;
        while (n % d == 0);
    }

return br_pfakt;
}
```

Algoritam 2 — komentari

Ovdje nam provjera **prostog** broja više **ne treba**, o tom se brine

- **rastući** poredak testiranja faktora = smjer petlje za d ,
- zajedno s **eliminacijom** svakog nađenog faktora.

Za **složene** brojeve n , dobivamo očito **ubrzanje** obzirom na prethodni algoritam — **nema** provjere višekratnika od p_i .

Međutim, ako je n **prost**, tj. za $n = p_1$, ušteda se svodi

- **samo** na izbjegavanje provjere je li broj prost.

Naime, **najveća** vrijednost test-faktora d je baš **jednaka** n .

Drugim riječima, imamo potpuno **istu** situaciju kao

- u prethodnom algoritmu, odnosno, u funkciji **prost_1**.

Algoritam 2 — komentari i poboljšanje

Za razliku od malo prije, ovdje **ima smisla** probati smanjiti **gornju** granicu za faktore d koje provjeravamo **u petlji**.

Ključna promjena u **ovom** algoritmu, obzirom na prethodni, je

- da petlja traži **najmanji** (prosti) faktor trenutnog broja n .

Onda, slično kao za **prost_2**, možemo drastično **smanjiti gornju** granicu za **manji** faktor, tako da uočimo sljedeće:

- ako n ima barem **dva** različita prosta faktora, onda je **manji** prosti faktor **najviše** jednak \sqrt{n} .

Ideja: **Prvo** nađemo sve “**manje**” proste faktore,

- one za koje vrijedi $p_i \leq \sqrt{n}$,

a **onda** vidimo što je ostalo u n , **nakon** njihove eliminacije.

Algoritam 3 — provjera “dijeljenjem” do \sqrt{n}

Za početak, **skratimo** petlju za “**manje**” faktore d do \sqrt{n} ,

- koristeći gornju granicu $\text{max_d} = (\text{int}) \text{sqrt}(n)$.

Nakon eliminacije **svih** tako nađenih faktora $p_i \leq \sqrt{n}$,

- u varijabli n može preostati samo **najveći** prosti faktor p_k .

To se događa ako i samo ako broj n

- **ima** prosti faktor $p_k > \sqrt{n}$, i onda je **nužno** $e_k = 1$.

Za “preostali” n onda vrijedi $n = p_k$ — sigurno je **prost**!

Dakle, **iza** petlje, samo provjerimo je li **preostali** $n > 1$. Ako je,

- **obradimo** ga kao **prosti** faktor $n = p_k$.

Algoritam 3 — nastavak

Funkcija za broj različitih prostih faktora (v. [pfbr_f_3.c](#)):

```
int broj_prost_fakt_3(int n)
{
    int br_pfakt = 0, d, max_d;

    if (n <= 1) return 0;

    max_d = (int) sqrt(n);
```

Algoritam 3 — nastavak

```
for (d = 2; d <= max_d; ++d)
    if (n % d == 0) {
        ++br_pfakt;    /* Obrada d, jednom. */

        do    /* Dijeli n s d, sve dok mozes. */
            n /= d;
        while (n % d == 0);
    }

if (n > 1)    /* Najveci prosti faktor. */
    ++br_pfakt;    /* Obrada d, jednom. */

return br_pfakt;
}
```


Algoritam 3 — komentar i poboljšanje

Za **Algoritam 2**, zaključili smo da je bitna stvar

- da petlja traži **najmanji** (prosti) faktor **trenutnog** broja n .

Ovdje je **namjerno** istaknuto “**trenutnog**”.

Naime, u **Algoritmu 3**, gornja granica za “**manje**” faktore d ,

- `max_d = (int) sqrt(n)`,
- postavljena je **jednom** — na početku, prema **početnoj** vrijednosti za n .

Međutim, čim **nađemo** i **eliminiramo** neki faktor $p_i \leq \sqrt{n}$,

- to **smanjuje** trenutnu vrijednost od n ,
- pa se i gornja granica `max_d` može **smanjiti** — prema **trenutnoj** vrijednosti od n .

Algoritam 4 — dinamička granica \sqrt{n}

Funkcija za broj različitih prostih faktora (v. [pfbr_f_4.c](#)):

```
int broj_prost_fakt_4(int n)
{
    int br_pfakt = 0, d, max_d;

    if (n <= 1) return 0;

    /* Pocetna granica max_d za ulazni n. */
    max_d = (int) sqrt(n);
```

Algoritam 4 — nastavak

```
for (d = 2; d <= max_d; ++d)
    if (n % d == 0) {
        ++br_pfakt;    /* Obrada d, jednom. */

        do    /* Dijeli n s d, sve dok mozes. */
            n /= d;
        while (n % d == 0);
        /* Popravak max_d za trenutni n. */
        max_d = (int) sqrt(n);
    }
if (n > 1)    /* Najveci prosti faktor. */
    ++br_pfakt;    /* Obrada d, jednom. */
return br_pfakt;
}
```

Algoritam 4 — završno poboljšanje

Završno poboljšanje algoritma ekvivalentno je

- poboljšanju funkcije `prost_2` u `prost_3`.

Prvo obradimo jedini mogući **parni** prosti faktor $d = 2$,

- a zatim idemo **samo** po **neparnim** faktorima do \sqrt{n} ,
- tj. korak za d u petlji je jednak **2**, a **ne 1**.

Ovom modifikacijom dobivamo skoro **dvostruko ubrzanje**, tj.

- potrebno vrijeme se skoro **prepolovi!**

Zbog **loše** rezolucije štoperice, ovo **ubrzanje** se baš i **ne vidi**

- u izmjerenim vremenima u **tablici** (v. malo kasnije).

Algoritam 5 — parnost i neparni do \sqrt{n}

Funkcija za broj različitih prostih faktora (v. `pfbr_f_5.c`):

```
int broj_prost_fakt_5(int n)
{
    int br_pfakt = 0, d, max_d;

    if (n <= 1) return 0;

    /* Pocetna granica max_d za ulazni n. */
    max_d = (int) sqrt(n);
```

Algoritam 5 — nastavak

```
    /* Prvo testiramo parni faktor d = 2. */  
  
if (2 <= max_d && n % 2 == 0) {  
    ++br_pfakt;    /* Obrada d, jednom. */  
  
    do    /* Dijeli n s 2, sve dok mozes. */  
        n /= 2;  
    while (n % 2 == 0);  
  
        /* Popravak max_d za trenutni n. */  
max_d = (int) sqrt(n);  
}  
  
/* Zatim neparni faktori d, pocev od 3. */
```

Algoritam 5 — nastavak

```
for (d = 3; d <= max_d; d += 2)
    if (n % d == 0) {
        ++br_pfakt;    /* Obrada d, jednom. */

        do    /* Dijeli n s d, sve dok mozes. */
            n /= d;
        while (n % d == 0);
        /* Popravak max_d za trenutni n. */
        max_d = (int) sqrt(n);
    }
if (n > 1)    /* Najveci prosti faktor. */
    ++br_pfakt;    /* Obrada d, jednom. */
return br_pfakt;
}
```

Usporedba složenosti — vremena

Za usporedbu ovih 5 algoritama (v. `VRIJEME\pfbr_test.c`),

🔴 računamo **zbroj** brojeva različitih prostih faktora, za **sve** brojeve n od 2 do 100 000. Evo što kaže štoperica:

funkcija	vrijeme (s)
broj_prost_fakt_1	14.231
broj_prost_fakt_2	2.095
broj_prost_fakt_3	0.065
broj_prost_fakt_4	0.025
broj_prost_fakt_5	0.014

Zadnja dva vremena **nisu** jako točna. No, očito se **isplate**

🔴 **korijenska** granica \sqrt{n} i njezino **dinamičko smanjenje!**

Usporedba složenosti — vremena (nastavak)

Rezolucija štoperice je oko 0.001 sekundi, pa ubrzanje zadnjeg algoritma izgleda manje no što stvarno je.

Za bolju usporedbu zadnja 3 algoritma (v. `pfbr_test1.c`), pustimo da n ide do 1 000 000 = deset puta više nego prije.

funkcija	vrijeme (s)
broj_prost_fakt_3	1.839
broj_prost_fakt_4	0.538
broj_prost_fakt_5	0.279

Ovdje se ubrzanje zadnjeg algoritma dobro vidi.

Prednost dinamičkog smanjenja granice \sqrt{n} se bolje vidi tek za mnogo veće brojeve — brojevi do 1 000 000 mogu imati najviše 7 različitih prostih faktora, a prosjek je samo 2.85!

Binomni koeficijenti i Pascalov trokut

Binomni koeficijenti i Pascalov trokut

Primjer. Treba napisati funkciju koja ima dva cjelobrojna argumenta n i k (tipa `int`). Funkcija treba izračunati i vratiti binomni koeficijent

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Binomni koeficijent je korektno definiran za $n \geq 0$ i $0 \leq k \leq n$. Zato provjeravamo ulazne vrijednosti.

🔴 U slučaju greške, vraćamo vrijednost 0.

Glavni program treba ispisati Pascalov trokut za $n \leq 20$.

🔴 U liniji s indeksom n nalaze se brojevi $\binom{n}{k}$, za sve vrijednosti $k = 0, \dots, n$.

Binomni koeficijent (nastavak)

Binomni koeficijenti su **doobar** primjer problema u kojem

- treba voditi računa o veličini i **prikazivosti** rezultata u cjelobrojnoj aritmetici računala.

Znamo da **faktorijele** vrlo brzo **rastu**. Zato originalna formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

nije dobra, jer već **13! nije** prikaziv u tipu **int** (na **32** bita).

Puno bolje je **skratiti jedan** od faktora iz nazivnika, **k!** ili **(n - k)!** — sigurno je i u brojniku. Samo **koji?**

- **Veći** od ta dva, naravno!

Binomni koeficijent (nastavak)

No, umjesto da testiramo i radimo s dvije formule, uočimo da su binomni koeficijenti **simetrični** u k i $n - k$, tj. vrijedi

$$\binom{n}{k} = \binom{n}{n-k}.$$

Zbog toga, po potrebi, možemo **zamijeniti** uloge k i $n - k$.

• Ako je $k > n - k$, **zamijenimo** im uloge: $k = n - k$.

• Nakon toga je sigurno $k \leq n - k$.

Zatim, **skratimo** zadnji (ujedno i **veći**) faktor $(n - k)!$, pa je

$$\binom{n}{k} = \frac{n \cdot (n - 1) \cdots (n - k + 1)}{1 \cdot 2 \cdots k}.$$

Binomni koeficijent (nastavak)

I sad ide **ključna** “sitnica”.

- Kojim **poretkom** izvršavamo aritmetičke **operacije** množenja i dijeljenja u ovoj formuli?

“Očiti” poredak operacija, s **jednim** dijeljenjem na **kraju**,

- pomnoži **brojnik**, pomnoži **nazivnik**, pa onda **podijeli**, **nije** dobar, jer **brojnik** opet **brzo naraste** (v. malo kasnije)!

Pravi algoritam je — pomnoži, **podijeli**, pomnoži, **podijeli**, ...

$$\binom{n}{k} = \frac{n}{1} \cdot \frac{n-1}{2} \cdots \frac{n-k+1}{k}.$$

Dodatno, zbog $k < n - k + 1$, svi faktori, uključivo i **zadnji**, su **veći** od 1. Rezultat stalno raste, i to puno **sporije**.

Binomni koeficijent — funkcija

Odgovarajuća funkcija je (v. `binom_f1.c`):

```
/* Funkcija binom(n, k) racuna binomni
   koeficijent n povrh k. */

int binom(int n, int k)
{
    int bin_coef, j;

    /* Provjera granica i signal greske. */
    if (n < 0 || k < 0 || k > n) return 0;

    /* Smanji donji argument. */
    if (k > n - k) k = n - k;
```

Binomni koeficijent — funkcija (nastavak)

```
if (k == 0) return 1;

bin_coef = n;
for (j = 2; j <= k; ++j)
    bin_coef = bin_coef * (n - j + 1) / j;

return bin_coef;
}
```

Uočite: Funkcija `binom` vraća **rezultat** tipa `int`.

Ako je tip `long` **veći** od `int`, isplati se staviti

• tip `long` za **vrijednost** funkcije i varijablu `bin_coef`,
jer binomni koeficijenti mogu biti **veliki**.

Binomni koeficijent — funkcija (nastavak)

Ključni dio funkcije `binom` možemo realizirati i `for` petljom (v. `binom_f2.c`):

```
bin_coef = n;  
for (j = 2, nmj = n - 1; j <= k; ++j, --nmj)  
    bin_coef = bin_coef * nmj / j;
```

Petlja “paralelno” mijenja dvije varijable:

- `j` za nazivnik — s pomakom `unaprijed`, i
- `nmj` za brojnik — s pomakom `unatrag`.

Operator zarez `,` ovdje služi za izvršenje

- po `dvije` naredbe u `inicijalizaciji` i `pomaku` petlje.

Sekvencijalno izvođenje tih operacija ovdje nije bitno!

Binomni koeficijent — *Ne tako!*

Na vježbama je napravljen algoritam koji odgovara sljedećem kôdu (v. `binom_fv.c`):

```
bin_coef = 1;
for (j = n; j > n - k; --j)
    bin_coef = bin_coef * j;
for (j = 2; j <= k; ++j)
    bin_coef = bin_coef / j;
```

Uočite da

- prvo množimo sve brojeve u brojniku (taj rezultat brzo raste),
- zatim dijelimo sa svim brojevima u nazivniku (rezultat stalno pada).

Binomni koeficijent — *Ne tako!* (nastavak)

Nažalost, to je samo **malo** bolje od onog ranijeg

• pomnoži **brojnik**, pomnoži **nazivnik**, pa onda **podijeli**,
jer **brojnik** odmah **brzo naraste**.

• Dijeljenja ima **puno**, ali su **prekasno** (opet na kraju)!

Ovaj algoritam prvi puta “**umire**” na

$$\binom{18}{9} = 48620.$$

Umjesto toga, vraćeni rezultat je **1276**.

Naša funkcija radi malo dalje. :-)

Binomni koeficijent — Zadaci

Zadatak. Naša funkcija `binom` računa **binomni koeficijent** tako da u brojniku i nazivniku idemo “**unaprijed**” (\rightarrow po formuli):

```
bin_coef = n;  
for (j = 2; j <= k; ++j)  
    bin_coef = bin_coef * (n - j + 1) / j;
```

Razmotrite je li **bolje** računati tako da u brojniku idemo “**unazad**”, a u nazivniku “**unaprijed**”:

```
bin_coef = n - k + 1;  
for (j = 2; j <= k; ++j)  
    bin_coef = bin_coef * (n - k + j) / j;
```

Pitanje: Moramo li u nazivniku ići “**unaprijed**”? **Zašto?**

Binomni koeficijent — Zadaci (nastavak)

Zadatak. Ispitajte **testiranjem** za koje ulazne brojeve n i k

• **razne** verzije funkcije **binom** rade **dobro**,
tj. korektno računaju binomni koeficijent $\binom{n}{k}$.

Odgovor (v. **binom_t.c**). U tipu **int** s **32** bita, obje funkcije

• množenjem “**unaprijed**” i “**unazad**” u brojniku,
prvi puta **griješe** na istom mjestu

$$\binom{30}{15} = 155\,117\,520 \neq -131\,213\,633 \quad (\text{vraćena vrijednost}).$$

Usput, prvi **neprikazivi** binomni koeficijenti su $\binom{34}{16}$ i $\binom{34}{17}$.
Kako biste to **testirali**?

Binomni koeficijent — Zadaci (nastavak)

Napomena. Još “pažljiviji” algoritam na bazi množenja možemo dobiti rastavom svih brojeva na **proste faktore**, tj.

- “praćenjem” **potencija prostih** faktora brojeva u brojniku i nazivniku (svi prosti faktori su $\leq n$).

No, to se **ne isplati** — predugo traje!

Zadatak. Probajte sastaviti odgovarajući algoritam.

Glavni program — Pascalov trokut

Primjer. Pascalov trokut za $n \leq 10$ izgleda ovako:

```
n = 0          1
n = 1        1  1
n = 2       1  2  1
n = 3      1  3  3  1
n = 4     1  4  6  4  1
n = 5    1  5 10 10  5  1
n = 6   1  6 15 20 15  6  1
n = 7  1  7 21 35 35 21  7  1
n = 8  1  8 28 56 70 56 28  8  1
n = 9  1  9 36 84 126 126 84 36  9  1
n = 10 1 10 45 120 210 252 210 120 45 10  1
```

Glavni program — Pascalov trokut (nastavak)

Radi jednostavnosti, **Pascalov** trokut ispisujemo poravnato po **lijevoj** strani, s **jednim** razmakom između brojeva:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
...
```


Glavni program — Pascalov trokut (nastavak)

```
#include <stdio.h>
    ...                /* Funkcija binom dodje tu. */
int main(void)
{
    int n, k;

    for (n = 0; n <= 20; ++n) {
        for (k = 0; k <= n; ++k)
            printf("%d ", binom(n, k));
        printf("\n");
    }
    return 0;
}
```

Pascalov trokut — Zadaci

Zadatak. Preuredite **glavni** program tako da ispisuje **Pascalov trokut centrirano**, kao u primjeru!

Zadatak. Kad napravimo strukturu **polja**, napravite program koji računa **red po red Pascalovog** trokuta, koristeći **polje** za jedan red trokuta, i formulu

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Ova formula vrijedi za $n > 1$ i $k \in \{1, \dots, n-1\}$. Rubovi su 1.

Dokažite da ovaj algoritam, zato što **zbraja**, **korektno** računa **sve prikazive** binomne koeficijente.

Kako biste **testirali** korektnost? (Pogledajte **binom_t.c.**)

Binomni koeficijent — aditivna rekurzija (Zlo!)

Primjer. Prethodna aditivna formula za Pascalov trokut,

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \quad n > 1, k \in \{1, \dots, n-1\},$$

može se iskoristiti i bez polja — za rekurzivno računanje binomnog koeficijenta, jer smanjuje n .

Treba joj samo dodati rubne i početne uvjete, za prekidanje rekurzije:

$$\binom{n}{0} = \binom{n}{n} = 1, \quad n \geq 0.$$

Za pojednostavljenije funkcije, očekujemo da su n i k korektno zadani, tj. $n \geq 0$ i $0 \leq k \leq n$, tako da nema provjere greške.

Provjeru argumenata treba napraviti prije — izvan rekurzije!

Binomni koeficijent — aditivna rekurzija (Zlo!)

Pripadna **rekurzivna** funkcija za binomni koeficijent $\binom{n}{k}$ je (v. `binom_r.c`):

```
int binom(int n, int k)
{
    /* Provjera granica za prekid rekurzije. */
    if (k == 0 || k == n) return 1;

    /* Rekurzija zbrajanjem. */
    return binom(n - 1, k) + binom(n - 1, k - 1) ;
}
```

Pitanje: I što je tu tako **zlo**? **Broj poziva** funkcije `binom`!

Za računanje $\binom{6}{2} = 15$, ima ih čak **29** — provjerite “na ruke”.

Binomni koeficijent — aditivna rekurzija (Zlo!)

Kroz rekurzivne pozive, kad argumenti “pomalo padaju”,

- puno puta se računaju isti binomni koeficijenti!

Zadatak. Izračunajte **ukupni** broj poziva funkcije **binom**, uključujući i vanjski poziv (onaj u glavnom programu),

- koji je potreban da se izračuna $\binom{n}{k}$.

Za olakšanje, možete funkciji **binom** dodati **brojač poziva**,

- bilo kao dodatni **varijabilni** argument, ili
- kao **globalnu** varijablu (v. **binom_r.c**).

Dokažite da je

$$\text{broj poziva} = 2 \binom{n}{k} - 1.$$

Obično i binarno potenciranje realnog broja

Cjelobrojna potencija realnog broja

Primjer. Zadani su **realni** broj x tipa **double** i **cijeli** broj n tipa **int**. Treba napisati funkciju koja računa i vraća

• n -tu **potenciju** broja x , tj. rezultat je x^n .

Kad je rezultat x^n **korektno** definiran, uz $x \in \mathbb{R}$ i $n \in \mathbb{Z}$?

• Ako je $x \neq 0$, onda x^n postoji za **bilo koji** $n \in \mathbb{Z}$.

• Ako je $x = 0$, onda je $x^n = 0$ za $n > 0$.

Inače, 0^0 je “neodređeni oblik” (kao limes), a $0^{-n} = \infty$.

Dogovor. U slučaju **greške** u argumentima, kad rezultat **nije** korektno definiran, **vraćamo** rezultat **nula**.

• Dakle, za $x = 0$ **uvijek** vraćamo rezultat $x^n = 0$, što je zgodno olakšanje.

Cjelobrojna potencija realnog broja (nastavak)

Uočimo da za $x \neq 0$ vrijedi

$$x^0 = 1, \quad x^{-n} = \frac{1}{x^n},$$

pa nam preostaje izračunati x^n za $n > 0$, odnosno, $x^{|n|}$.

Funkcija za cjelobrojnu apsolutnu vrijednost zove se **abs**. Deklarirana je u zaglavlju `<stdlib.h>`, a prototip joj je

```
int abs(int)
```

Poziv `abs(n)` vraća vrijednost $|n|$.

Cijeli algoritam zovemo `int_pow`, što je skraćeno od

🔴 engl. “integer power” = cjelobrojna potencija.

Cjelobrojna potencija — kostur algoritma

Kostur algoritma `int_pow` onda ima sljedeći oblik:

```
    /* Provjera x = 0. */  
    if (x == 0.0) return 0.0;  
  
    /* Zapamti predznak od n. */  
    neg = n < 0;  n = abs(n);  
  
    /* Izracunaj pot = x^n, uz n >= 0. */  
    ...  
  
    if (neg) pot = 1.0 / pot;  
    return pot;
```

Računanje $\text{pot} = x^n$, za $n \geq 0$, realiziramo na dva načina.

Obično potenciranje — ponovljeno množenje

Spora varijanta rješenja je “ponovljeno množenje” broja x sa samim sobom, koliko puta treba,

• ovisno o inicijalizaciji za akumulaciju produkta.

Produkt (potenciju) akumuliramo u varijabli `pot`.

Ako želimo da algoritam radi i za $n = 0$, onda je zgodno inicijalizirati produkt `pot` na `1` — neutral za množenje.

Ovaj algoritam odgovara računanju potencija x^n po sljedećoj “rekurzivnoj” relaciji

$$x^n = \begin{cases} 1, & \text{za } n = 0, \\ x \cdot x^{n-1}, & \text{za } n > 0. \end{cases}$$

Složenost: Treba nam tačno n množenja.

Obično potenciranje — funkcija

Funkcija za obično potenciranje (v. `pot_mul.c`):

```
double int_pow_mul(double x, int n)
{
    double pot = 1.0;
    int neg, i;

    /* Provjera x = 0. */
    if (x == 0.0) return 0.0;

    /* Zapamti predznak od n. */
    neg = n < 0;
    n = abs(n);
```

Obično potenciranje — funkcija (nastavak)

```
    /* Potenciranje množenjem. */  
    for (i = 1; i <= n; ++i)  
        pot *= x;  
  
    if (neg) pot = 1.0 / pot;  
    return pot;  
}
```

Poboljšanje. Početno množenje s 1 može se izbjeći tako da

- pitamo je li $n > 0$ (u protivnom vratimo 1.0)
- i inicijaliziramo $pot = x$, a start petlje je $i = 2$.

Onda nam treba $n - 1$ množenja, za $n > 0$.

Obično potenciranje — glavni program

```
int main(void)
{
    double x = 2.0;
    int n;

    n = 5;
    printf(" Potencija %g na %2d = %g\n",
           x, n, int_pow_mul(x, n) );
    n = -5;
    printf(" Potencija %g na %2d = %g\n",
           x, n, int_pow_mul(x, n) );
    return 0;
}
```

Potenciranje — format ispisa i rezultati

Ispis. U oznaci konverzije `%2d`, broj `2` zadaje minimalnu širinu ispisa, tj. minimalni broj znakova koji će se ispisati.

Ako podatak treba:

- manje znakova od zadanog broja, bit će slijeva dopunjen bjelinama do tog broja znakova (osim ako nije zadano drugačije dopunjavanje — tzv. “zastavicama”);
- više znakova od minimalne širine ispisa, bit će korektno ispisan sa svim potrebnim znakovima.

Rezultati. Za $x = 2$ i $n = \pm 5$, dobivamo

Potencija 2 na 5 = 32

Potencija 2 na -5 = 0.03125

Binarno potenciranje — kvadriranje i množenje

Puno brža varijanta je “ponovljeno kvadriranje i množenje”,
ili, standardnim imenom, “binarno potenciranje”,
jer se dobiva iz binarnog zapisa eksponenta n .

Pretpostavimo da je $n > 0$ i neka je

$$n = n_\ell 2^\ell + n_{\ell-1} 2^{\ell-1} + \dots + n_1 2 + n_0 = \sum_{i=0}^{\ell} n_i 2^i$$

normalizirani prikaz broja n u bazi 2. Za znamenke n_i vrijedi

$$n_0, \dots, n_{\ell-1} \in \{0, 1\} \quad \text{i} \quad n_\ell > 0,$$

a broj znamenki u tom prikazu jednak je

$$\ell + 1 = \lfloor \log_2 n \rfloor + 1.$$

Binarno potenciranje (nastavak)

Onda je

$$x^n = x^{\left(\sum_{i=0}^{\ell} n_i 2^i\right)} = \prod_{i=0}^{\ell} x^{n_i 2^i}.$$

No, **binarne** znamenke n_i mogu biti **samo** 0 ili 1, pa je

$$x^{n_i 2^i} = \begin{cases} 1, & \text{za } n_i = 0, \\ x^{2^i}, & \text{za } n_i = 1. \end{cases}$$

Dakle, u gornjem produktu ostaju **samo** faktori za znamenke $n_i = 1$ u binarnom zapisu broja n

$$x^n = \prod_{\substack{i=0 \\ n_i=1}}^{\ell} x^{2^i}.$$

Binarno potenciranje (nastavak)

Faktori u tom produktu dobivaju se **kvadriranjem** prethodnog

$$x^{2^i} = x^{2 \cdot 2^{i-1}} = (x^{2^{i-1}})^2, \quad i > 0,$$

uz početak $x^{2^0} = x^1 = x$.

Ako definiramo **novi niz** vrijednosti $b_i := x^{2^i}$, za $i = 0, \dots, \ell$, onda članove tog niza računamo po “**rekurzivnoj**” relaciji

$$b_i = \begin{cases} x, & \text{za } i = 0, \\ (b_{i-1})^2, & \text{za } i > 0. \end{cases}$$

Tražena potencija je

$$x^n = \prod_{\substack{i=0 \\ n_i=1}}^{\ell} b_i.$$

Binarno potenciranje (nastavak)

Neka je potencija **pot** inicijalizirana na 1, kao prije. Algoritam **binarnog** potenciranja “**paralelno**” radi sljedeće **tri** operacije

- izdvaja **binarne** znamenke n_i eksponenta n ,
- računa članove niza b_i — **kvadriranjem** u varijabli **b**,
- akumulira u **pot** produkt članova b_i za koje je $n_i = 1$.

Na primjer, za $n = 6 = (110)_2$, imamo

$$x^6 = 1 \cdot (x^2) \cdot (x^2)^2 = 1 \cdot b_1 \cdot b_2.$$

Složenost: Treba nam **tačno** ℓ množenja za članove b_i i **najviše** još $\ell + 1$ množenja za akumulaciju potencije (kad binarni zapis broja n ima **sve** jedinice, tj. za $n = 2^{\ell+1} - 1$).

Dakle, treba nam **najviše** $2\lfloor \log_2 n \rfloor + 1$ množenja!

Binarno potenciranje — funkcija

Funkcija za binarno potenciranje (v. `pot_bin.c`):

```
double int_pow_bin(double x, int n)
{
    double pot = 1.0, b = x;
    int neg;

    /* Provjera x = 0. */
    if (x == 0.0) return 0.0;

    /* Zapamti predznak od n. */
    neg = n < 0;
    n = abs(n);
```

Binarno potenciranje — funkcija (nastavak)

```
    /* Potenciranje kvadriranjem i množenjem. */  
while (n > 0) {  
    if (n % 2 == 1) pot *= b;  
    n /= 2;  
    if (n > 0) b *= b;  
}  
  
if (neg) pot = 1.0 / pot;  
return pot;  
}
```

Za $x = 2$ i $n = \pm 5$, rezultati su, naravno, isti kao i prije.

Na dnu petlje, može i ovako: `if (n == 0) break; b *= b;`

Binarno potenciranje — bolji primjer

Ogromna razlika u brzini se baš i ne vidi, sve dok ne probate ovako nešto: $n = 10^9$ i $x = 1 + 10^{-9} = 1 + \frac{1}{n}$ (v. `pot_test.c`).

```
int main(void)
{
    double x = 1.0000000001;
    int n = 1000000000;

    printf(" Potencija %11.9f na %2d = %11.9f\n",
           x, n, int_pow_mul(x, n) );
    printf(" Potencija %11.9f na %2d = %11.9f\n",
           x, n, int_pow_bin(x, n) );
    return 0;
}
```

Preciznost ispisa realnih brojeva

Pored minimalne širine, moguće je zadati i **preciznost** ispisa. Kod realnih brojeva, **preciznost** je

- (najveći) broj **decimala** (za **%f** i **%e**), odnosno, **vodećih** znamenki (za **%g**), koje će biti ispisane.

Sintaksa:

- **%a.bf** ili **%a.be** ili **%a.bg**, gdje je
 - **a** — minimalna širina ispisa,
 - **b** — preciznost.

Primjer.

- **%11.9f** — znači ispis u **f** formatu s **najmanje 11** znakova, pri čemu je **najviše 9** znamenki iza decimalne točke.

Ispis **bez** specificirane **preciznosti** \implies **preciznost = 6**.

Binarno potenciranje — bolji primjer (nastavak)

Rezultati su:

Potencija 1.000000001 na 1000000000 = 2.718282052

Potencija 1.000000001 na 1000000000 = 2.718282031

Tu se dobro vide razlike u brzini i točnosti.

Na prvi rezultat (`int_pow_mul`) čekam oko 1 sekundu.

- 👉 Vjerovali ili ne, to je jako brzo, jer Intelov compiler još vektorizira petlju u funkciji!

Drugi rezultat (`int_pow_bin`) izađe “trenutno” i nešto je točniji, zbog manje akumulacije grešaka zaokruživanja.

Probajte na svom “kompu”!

Realna potencija realnog broja — funkcija `pow`

U matematičkoj biblioteci `<math.h>` postoji **opća** funkcija za potenciranje **realnih** brojeva tipa `double`. Prototip je

```
double pow(double, double)
```

a poziv `pow(x, y)` vraća vrijednost x^y .

Zadatak. Dodajte ispis vrijednosti `pow(x, n)` u glavni program iz prošlog primjera i provjerite **točnost** rezultata.

Točni rezultat na 25 decimala za $x = 1 + 10^{-9}$ i $n = 10^9$ je

$$x^n = 2.71828 18270 99904 32237 66440.$$

Binarno potenciranje — komentari

Zgodna ubrzanja za test neparnosti i dijeljenje s 2 — preko posebnih operatora u C-u:

• `if (n % 2 == 1) → if ((n & 1) != 0)`, ili samo
`if (n & 1)`

• `n /= 2; → n >>= 1;`

Poboljšanje. I ovdje se početno množenje s 1 može izbjeći, tako da “pažljivo” inicijaliziramo `pot`, kad je $n > 0$.

• Inicijalizacija je `pot = b`, u trenutku kad naiđemo

• na prvu (= najnižu) jedinicu u binarnom zapisu od n .

Onda nam treba najviše $2 \lfloor \log_2 n \rfloor$ množenja, za $n > 0$.

Zadatak. Sastavite taj algoritam! (v. directory Powers).

Binarno potenciranje *nije* optimalno

Uzmite da smo eliminirali nepotrebno množenje s 1. Čak i tad,

● binarno potenciranje *ne mora* biti optimalno!

Ako dozvolimo pamćenje “međurezultata” (nižih potencija),

● onda postoje $n \in \mathbb{N}$, za koje x^n možemo izračunati i s manje množenja.

Zadatak. Nađite najmanji eksponent n za kojeg binarno potenciranje *nije* optimalno,

● tj. nađite algoritam koji ima manje množenja za taj n .

Rješenje. Za $n = 15$, binarno potenciranje treba 6 množenja, a dovoljno je samo 5 množenja — računa se kao $x^{15} = (x^3)^5$.

Prvi sljedeći je $n = 23$. Nađite algoritam sa samo 6 množenja!

Složenost nalaženja *optimalnog* algoritma

Napomena. Za dani n , broj potrebnih množenja za računanje x^n je **barem** $\lfloor \log_2 n \rfloor$ — kao kod binarnog potenciranja.

Međutim, **nema** “jednostavne” **formule**, pa čak ni **algoritma**, za nalaženje **minimalnog** broja množenja za zadani n . Naime,

- problem nalaženja **minimalnog** broja množenja za n je **dokazano težak** — pripada klasi tzv. **NP-teških** problema, kao i problem trgovačkog putnika.

Ako vas zanima više o potenciranju, potražite na **Wikipediji**:

- **Exponentiation by squaring**,
- **Addition-chain exponentiation**.