Oblikovanje i analiza algoritama
9. predavanje

Saša Singer

singer@math.hr

web.math.pmf.unizg.hr/~singer

PMF – Matematički odsjek, Zagreb

Sadržaj predavanja

- Fibonaccijevi brojevi:
  - Uvod, problem FIB, DeMoivreova formula.
  - FIB — rekurzivni algoritam.
  - FIB — aditivni algoritam.
  - Potenciranje i binarno potenciranje.
  - Brzo računanje n-tog člana rekurzije.
  - FIB — brzi algoritam.
  - FIB — sve znamenke i stvarna složenost.

Informacije — web stranica

Moja web stranica za Oblikovanje i analizu algoritama je

https://web.math.pmf.unizg.hr/~singer/oaa/

ili, skraćeno

https://web.math.hr/~singer/oaa/

Kopija je na adresi

http://degiorgi.math.hr/~singer/oaa/

Službena web stranica za Oblikovanje i analizu algoritama je

https://web.math.pmf.unizg.hr/nastava/oaa/

Fibonaccijevi brojevi — Uvod, DeMoivreova formula

Pojednostavljenje DeMoivreove formule

Egzaktna DeMoivreova formula je

$$F_n = \frac{1}{\sqrt{5}} \left(\Phi^n - (-\Phi)^{-n} \right), \quad n \in \mathbb{N}_0.$$

Kako je $\Phi^{-1} < 1$, onda je $\Phi^{-n}/\sqrt{5} < 1/2$, za svaki $n \in \mathbb{N}_0$, pa drugi član možemo zanemariti, tj. vrijedi

$$F_n = \left[\frac{1}{\sqrt{5}} \Phi^n \right], \quad n \in \mathbb{N}_0,$$

gdje $[x]$ označava "najbliže cijelo" od x.

To odgovara korektnom zaokruživanju realnog broja $\Phi^n/\sqrt{5}$ na najbliži cijeli broj!

Implementacija svih algoritama za F_n

Kod implementacije i usporedbe različitih algoritama za računanje F_n, za prikaz Fibonaccijevih brojeva koristimo

- "najveći" realni tip, koji je korektno podržan arhitekturom i programskom bibliotekom u danom jeziku.

U nastavku, standardno koristim tip `double` — za Intel C.

Mogu još koristiti `extended` — za GNU C, ili Pascal/Delphi (ali samo na ia32).

Razlozi za izbjegavanje cjelobrojnih tipova = F_n brzo rastu:

- mali raspon prikazivih cijelih brojeva (do F_{93}, sa 64 bita),
- modularna aritmetika cijelih brojeva — dobivamo samo donje bitove pravog rezultata.

Implementacija DeMoivreove formule

U implementaciji DeMoivreove formule u realnoj aritmetici,

- izbacujemo i "najbliže cijelo", tj. vraćamo realni broj.

Kod ispisa, rezultat se ionako "zaokružuje" na zadani broj decimala, tj. ispis s 0 decimala (`%.0f`) radi to što treba!

Funkcija za računanje F_n po DeMoivreovoj formuli:

```
double Fib(int n)
{
    double Sqrt5 = sqrt(5.0), Fi = (1 + Sqrt5) / 2;

    return pow(Fi, n) / Sqrt5;
}
```

Rezultati za $n = 78$ (Intel C 18.0.1, ia32)

Najveći egzaktno prikazivi Fibonaccijev broj u tipu `double` je

$$F_{78} = 8\ 944\ 394\ 323\ 791\ 464 \approx 8.9 \cdot 10^{15}.$$

Međutim, zbog grešaka zaokruživanja u realnoj aritmetici i u funkcijama `sqrt`, `pow` iz `<math.h>`,

- izračunati rezultat je za 24 prevelik (gubitak točnosti od 4–5 bitova, od ukupno 53 bita).

Izračunati rezultat i pripadno vrijeme je:

```
     F(n) = F_n                          T(n) [s]
=============================================================
 F(   78) =   8944394323791488     2.260000e-08
=============================================================
```

Rezultati za $n = 2^k$ (Intel C 18.0.1, ia32)

Repeat count: Rpt = 10000000 = 10^7

F(n) = F_n			Rpt * T(n) [s]	c_n = T(n)
F(2)	=	1.1708203932e+00	0.226	2.260000e-08
F(4)	=	3.0652475842e+00	0.228	2.280000e-08
F(8)	=	2.1009519494e+01	0.226	2.260000e-08
F(16)	=	9.8700020263e+02	0.228	2.280000e-08
F(32)	=	2.1783090000e+06	0.227	2.270000e-08
F(64)	=	1.0610209858e+13	0.226	2.260000e-08
F(128)	=	2.5172882568e+26	0.226	2.260000e-08
F(256)	=	1.4169381771e+53	0.294	2.940000e-08
F(512)	=	4.4893845313e+106	0.293	2.930000e-08
F(1024)	=	4.5066996337e+213	0.293	2.930000e-08

FIB — rekurzivni algoritam

Implementacija rekurzivnog algoritma za F_n

Rekurzivna funkcija za računanje F_n:

```
double Fib(int n)
{
    if (n > 1)
            /* Rekurzivni pozivi. */
        return Fib(n - 1) + Fib(n - 2);

    else    /* n <= 1. */
            /* Pretpostavljamo da je n = 0, 1,
               tj., n >= 0. */
        return n;
}
```

Rezultati za $n = 34, \ldots, 45$ (Intel C 18.0.1, ia32)

F(n) = F_n		T(n) [s]	c_n = T(n) / F(n)
F(34) =	5702887	0.045	7.890740e-09
F(35) =	9227465	0.074	8.019537e-09
F(36) =	14930352	0.120	8.037319e-09
F(37) =	24157817	0.193	7.989132e-09
F(38) =	39088169	0.312	7.981955e-09
F(39) =	63245986	0.505	7.984696e-09
F(40) =	102334155	0.816	7.973877e-09
F(41) =	165580141	1.320	7.971971e-09
F(42) =	267914296	2.134	7.965234e-09
F(43) =	433494437	3.452	7.963193e-09
F(44) =	701408733	5.581	7.956844e-09
F(45) =	1134903170	9.029	7.955745e-09

Predviđanje za $n = 78$ (Intel C 18.0.1, ia32)

Uzmimo da je u modelu složenosti $c = 7.9 \cdot 10^{-9}$ (sekundi).

Iz prethodne tablice, to je prilično točna aproksimacija (na 2 dekadske znamenke, za još veće n).

Koristeći tu vrijednost, izračunajmo (približno) potrebno vrijeme rekurzivnog algoritma za računanje

$$F_{78} = 8\ 944\ 394\ 323\ 791\ 464 \approx 8.9 \cdot 10^{15}.$$

Dobivamo

$$T(78) = c \cdot F_{78} \approx (7.9 \cdot 10^{-9}) \cdot (8.9 \cdot 10^{15}) \approx 7.03 \cdot 10^{7}.$$

U prijevodu, algoritam treba oko 70 milijuna sekundi.

Kad uzmemo da je broj sekundi u jednoj godini oko $3.05 \cdot 10^{7}$, slijedi da rekurzivni algoritam za F_{78} traje oko 2.3 godine!

FIB — aditivni algoritam

Implementacija aditivnog algoritma za F_n

Aditivna funkcija za računanje F_n:

```
double Fib(int n)
{
    double F, F1, F2;
    int i;

    if (n <= 1)     /* Pretp. da je n = 0, 1. */
        return n;

    else {  /* n > 1. Inicijaliziraj ‘‘prozor’’
               (prva dva clana). */
        F1 = 0;     // Fib(0) = 0.
        F  = 1;     // Fib(1) = 1.
```

Aditivna funkcija za F_n (nastavak)

```
            /* Pomak (klizanje) ‘‘prozora’’
               od 3 susjedna clana. */

        for (i = 2; i <= n; ++i) {
            F2 = F1;          // Pomak F1 u F2.
            F1 = F;           // Pomak F u F1.
            F = F1 + F2;      // F = Fib(i) = F1 + F2.
        }

        return F;
    }
}
```

Rezultati za $n = 78$, 2^k (Intel C 18.0.1, ia32)

Repeat count: Rpt = 10000000 = 10^7

F(n) = F_n			Rpt * T(n) [s]	c_n = T(n) / n
F(78)	=	8944394323791464	0.505	6.474359e-10
F(4)	=	3.0000000000e+00	0.025	6.250000e-10
F(8)	=	2.1000000000e+01	0.042	5.250000e-10
F(16)	=	9.8700000000e+02	0.069	4.312500e-10
F(32)	=	2.1783090000e+06	0.164	5.125000e-10
F(64)	=	1.0610209858e+13	0.371	5.796875e-10
F(128)	=	2.5172882568e+26	0.924	7.218750e-10
F(256)	=	1.4169381771e+53	2.024	7.906250e-10
F(512)	=	4.4893845313e+106	4.219	8.240234e-10
F(1024)	=	4.5066996337e+213	8.614	8.412109e-10

Problem aditivnog algoritma

Problem. Da bismo izračunali n-ti član niza, ovdje je to F_n,

- redom računamo sve prethodne članove, do n-tog,
- tj., i sve prethodne Fibonaccijeve brojeve $F_0, F_1, \ldots, F_n$.

Zato je složenost algoritma (barem) linearna u n.

Ovo možemo usporediti s računanjem n-te potencije zadanog broja x (možemo uzeti da je x realan, nije bitno).

- Ako potenciranje radimo iterativnim (sukcesivnim) množenjem, onda računamo i sve potencije od x, do n-te.

Međutim, postoji i puno brži algoritam za računanje x^n,

- tzv. binarno potenciranje (v. Prog1, dodatak 10a).

Složenost tog algoritma je logaritamska u n.

Potenciranje i binarno potenciranje

Potenciranje i rekurzija prvog reda

Neka je zadan realni (kompleksni) broj x i eksponent $n \in \mathbb{N}_0$. Treba izračunati pripadnu potenciju x^n.

Problemu potenciranja odgovara sljedeća rekurzivna relacija

$$t_n = x \cdot t_{n-1}, \quad \text{za } n \geq 1,$$

uz početni uvjet $t_0 = 1$. Ovo je

- linearna homogena rekurzija prvog reda, s konstantnim koeficijentom $a_0 = x$.

Očito rješenje je $t_n = x^n$, za svaki $n \geq 0$.

Računanje $\texttt{pot} = x^n$, 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 točno n množenja.

Obično potenciranje — funkcija

Funkcija za obično potenciranje (v. pow_mul_0.c):

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

        /* Potenciranje mnozenjem. */
    for (i = 1; i <= n; ++i)
        pot *= x;

    return pot;
}
```

Obično potenciranje — bolja implementacija

Poboljšanje. Početno množenje s 1 može se izbjeći.

Sasvim općenito, umjesto inicijalizacije `pot = 1`,

- koja služi tome da algoritam radi i za $n = 0$ (kraći kod),

inicijalizaciju `pot` treba napraviti "na pravom mjestu".

Za potenciranje množenjem,

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

Dakle, inicijalizacija je `pot = x`, čim znamo da je $n \geq 1$.

Složenost: Ovdje treba točno $n - 1$ množenja, za $n > 0$.

Obično potenciranje — bez množenja s 1

```
double pow_mul(double x, int n)
{
    double pot;
    int i;

    if (n > 0) {
        pot = x;
        for (i = 2; i <= n; ++i)
            pot *= x;
        return pot;
    } else
        return 1.0;
}
```

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

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

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

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

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

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

uz početak $x^{2^0} = x^1 = x$.

Ako definiramo novi niz vrijednosti $b_i := x^{2^i}$, za $i = 0, \ldots, \ell$, 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 točno 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. pow_bin.c):

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

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

Binarno potenciranje — mala poboljšanja

Ubrzanja za test neparnosti i dijeljenje s 2, preko posebnih operatora u C-u — umjesto prve dvije naredbe u petlji

```
if (n % 2 == 1) pot *= b;
n /= 2;
```

možemo staviti

```
if ((n & 1) != 0) pot *= b;
n >>= 1;
```

Umjesto zadnje naredbe u petlji (test $n > 0$), možemo staviti

```
if (n == 0) break;
b *= b;
```

Binarno potenciranje — bolja implementacija

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.

Složenost: Ovo treba najviše $2\lfloor \log_2 n \rfloor$ množenja, za $n > 0$.
S donje strane, broj množenja je barem $\lfloor \log_2 n \rfloor$ (kvadriranja).

Dodatno, u ovoj varijanti algoritma, prirodno je

- okrenuti poredak naredbi za kvadriranje i akumulaciju produkta $\longrightarrow$ prvo kvadriraj, pa akumuliraj (ako treba).

Time se izbjegava test za kvadriranje unutar petlje!

Binarno potenciranje — bez množenja s 1

```
double pow_bin(double x, int n)
{
    double pot, b = x;

        /* Prvo rijesimo specijalni slucaj n = 0. */
    if (n == 0) return 1.0;

        /* Sve dok je (trenutni) n paran (zadnja
           binarna znamenka je 0), kvadriraj b i
           obrisi zadnju znamenku u n. */
    while ((n & 1) == 0) {    // (n % 2 == 0)
        b *= b;      // Kvadriraj b.
        n >>= 1;     // n /= 2;
    }
```

Binarno potenciranje — bez množ. s 1 (nast.)

```
        /* Zadnja binarna znamenka (trenutnog) n je
           jednaka 1, tj. n je neparan.
           Inicijaliziraj pot na b, obrisi zadnju
           znamenku (jedinicu). */

pot = b;
n >>= 1;     // n /= 2;
```

Binarno potenciranje — bez množ. s 1 (nast.)

```
        /* Petlja za preostale binarne znamenke
           od n. */
    while (n > 0) {
        b *= b;      // Kvadriraj b.

            /* Ako je n neparan, onda akumuliraj
               produkt u pot (pomnozi pot s b). */
        if ((n & 1) != 0) pot *= b;

        n >>= 1;     // n /= 2;
    }

    return pot;
}
```

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 (pamtimo $y = x^3$, $z = y^2 = x^6$):

$$x^{15} = (x^3)^5 = \{y = x * x * x\} = y^5 = \{z = y * y\} = y * z * z.$$

Brzo računanje n-tog člana rekurzije

Rekurzija i potenciranje — uvod

Vidjeli smo da problemu potenciranja broja (x^n) odgovara

- linearna homogena rekurzija prvog reda, s konstantnim koeficijentom $a_0 = x$

$$t_n = x \cdot t_{n-1}, \quad \text{za } n \geq 1,$$

uz početni uvjet $t_0 = 1$.

- U tom slučaju, imamo brzi algoritam za računanje n-tog člana rekurzije, logaritamske složenosti u ovisnosti o n.

Pitanje je — može li se rekurzija višeg reda $k > 1$ svesti na problem potenciranja? Odgovor je potvrdan:

- Rekurzije višeg reda možemo prikazati kao sustav rekurzivnih jednadžbi "prvog" reda.
- Dimenzija tog sustava = red k.

Matrično–vektorski zapis rekurzije

Linearna homogena rekurzivna relacija reda k, s konstantnim koeficijentima ima oblik

$$t_n = a_{k-1}t_{n-1} + \cdots + a_0 t_{n-k}, \quad \text{za} \quad n \geq k, \tag{6}$$

s tim da se zadaje k početnih uvjeta $t_0, \ldots, t_{k-1}$.

Uvedimo vektorski niz $v^{(n)}$, dimenzije k, tj. $v^{(n)} \in \mathbb{C}^k$ (ili $\mathbb{R}^k$), formulom

$$v^{(n)} = \begin{bmatrix} t_n \\ t_{n+1} \\ \vdots \\ t_{n+k-1} \end{bmatrix}, \quad \text{za} \quad n \in \mathbb{N}_0.$$

Uočiti: vektor $v^{(0)}$ je upravo vektor početnih uvjeta rekurzije, tj. dovoljno je zadati $v^{(0)}$.

Matrično–vektorski zapis rekurzije (nastavak)

Rekurzivnu relaciju (6) možemo zapisati u matričnom obliku

$$v^{(n)} = X \cdot v^{(n-1)}, \quad \text{za} \quad n \geq 1,$$

gdje je X matrica reda k ($X \in \mathbb{C}^{k \times k}$), sljedećeg oblika

$$X = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ddots & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ a_0 & a_1 & a_2 & \cdots & a_{k-2} & a_{k-1} \end{bmatrix}.$$

Zato što rekurzija ima konstantne koeficijente, matrica X je konstantna matrica, tj. ne ovisi o n.

Matrično–vektorski zapis rekurzije (nastavak)

U zapisu $v^{(n)} = X \cdot v^{(n-1)}$, nova komponenta $v_i^{(n)}$ dobiva se kao

- skalarni produkt i-tog retka matrice X i prošlog vektora $v^{(n-1)}$.

Prvih $k-1$ redova matrice X opisuje pomak komponenti za jedno mjesto prema gore (ili "unatrag"),

$$v_i^{(n)} = v_{i+1}^{(n-1)}, \quad \text{za} \quad i = 1, \ldots, k-1.$$

Zadnji red matrice X opisuje računanje zadnje komponente $v_k^{(n)}$ novog vektora — prema rekurziji (6)

$$v_k^{(n)} = a_0 v_1^{(n-1)} + \cdots + a_{k-1} v_k^{(n-1)}, \quad \text{za} \quad n \geq 1.$$

Matrično–vektorska rekurzija i potenciranje

Dobili smo matrično–vektorsku rekurziju prvog reda

$$v^{(n)} = X \cdot v^{(n-1)}, \quad \text{za} \quad n \geq 1,$$

uz vektorski početni uvjet $v^{(0)} = [\ t_0\ \ldots\ t_{k-1}\ ]^T$.

Ova rekurzija ima upravo oblik rekurzije za potenciranje, samo što broj x prelazi u matricu X, reda k.

I ovdje se "opći" član $v^{(n)}$ trivijalno izražava preko početnog vektora $v^{(0)}$

$$v^{(n)} = X^n \cdot v^{(0)}, \quad \text{za} \quad n \geq 0.$$

Ova formula vrijedi i za $n = 0$, jer je $X^0 = I$.

Zaključak: Polazna "skalarna" rekurzija (6) (reda k) je svedena na potenciranje matrice X, reda k.

Računanje n-tog člana — izbor eksponenta

Problem: Treba izračunati n-ti član t_n niza rješenja zadane rekurzije (6), uz zadane početne uvjete $t_0, \ldots, t_{k-1}$. Uzimamo da je $n \geq k$ — inače samo vratimo početni uvjet (bez računa).

Taj član t_n se prvi puta javlja u vektoru $v^{(n-k+1)}$, kao zadnja komponenta

$$t_n = v_k^{(n-k+1)}, \qquad v^{(n-k+1)} = X^{n-k+1} \cdot v^{(0)}.$$

No, možemo uzeti i neki drugi vektor, u kojem se član t_n javlja na mjestu i, za $i = 1, \ldots, k$. Takvi vektori su $v^{(n-i+1)}$

$$t_n = v_i^{(n-i+1)}, \qquad v^{(n-i+1)} = X^{n-i+1} \cdot v^{(0)}.$$

Dozvoljeni eksponenti su $m = n - i + 1 \in \{n - k + 1, \ldots, n\}$.

Obično i binarno potenciranje — za izabrani m

Uzmimo da smo izabrali eksponent m. Neka je $\ell = \lfloor \log_2 m \rfloor$.

Analogon običnog potenciranja, redom, računa sve vektore $v^{(1)}, \ldots, v^{(m)}$, množeći matricu X na prethodni vektor.

Analogon binarnog potenciranja računa niz matrica

$$B_i = \begin{cases} X, & \text{za } i = 0, \\ (B_{i-1})^2, & \text{za } i = 1, \ldots, \ell, \end{cases}$$

i akumulira produkt tih matrica na početni vektor

$$v^{(m)} = X^m \cdot v^{(0)} = \left(\prod_{\substack{i=0 \\ m_i=1}}^{\ell} B_i \right) v^{(0)},$$

gdje su $m_0, \ldots, m_\ell$ binarne znamenke broja m.

FIB — brzi algoritam

Implementacija brzog algoritma za F_n

Koristimo binarno potenciranje, bez dodatnog množenja s 1, i

- za parne n — eksplicitno postavljamo prvu matricu na F^2, a ne na F.

Rezultati za $n = 78$, 2^k (Intel C 18.0.1, ia32)

Repeat count: Rpt = 100000000 = 10^8

F(n) = F_n			Rpt * T(n) [s]	c_n = T(n) / lg(n)
F(78)	=	8944394323791464	1.144	3.788280e-09
F(4)	=	3.0000000000e+00	0.286	2.976358e-09
F(8)	=	2.1000000000e+01	0.372	2.580898e-09
F(16)	=	9.8700000000e+02	0.472	2.456015e-09
F(32)	=	2.1783090000e+06	0.598	2.489317e-09
F(64)	=	1.0610209858e+13	0.725	2.514988e-09
F(128)	=	2.5172882568e+26	0.891	2.649285e-09
F(256)	=	1.4169381771e+53	1.050	2.731797e-09
F(512)	=	4.4893845313e+106	1.254	2.900041e-09
F(1024)	=	4.5066996337e+213	1.455	3.028392e-09

Rezultati za $n = 2^k - 1$ (Intel C 18.0.1, ia32)

Repeat count: Rpt = 100000000 = 10^8

F(n) = F_n		Rpt * T(n) [s]	c_n = T(n) / lg(n)
F(3) =	2.0000000000e+00	0.373	4.898227e-09
F(7) =	1.3000000000e+01	0.567	4.203730e-09
F(15) =	6.1000000000e+02	0.843	4.491024e-09
F(31) =	1.3462690000e+06	1.166	4.898628e-09
F(63) =	6.5574703198e+12	1.461	5.087398e-09
F(127) =	1.5557697022e+26	1.813	5.399474e-09
F(255) =	8.7571595343e+52	2.149	5.595027e-09
F(511) =	2.7745922289e+106	2.521	5.831973e-09
F(1023) =	2.7852935507e+213	2.866	5.966044e-09

Usporedba algoritama (Intel C 18.0.1, ia32)

Pregled potrebnih vremena za računanje F_{1023} i F_{1024}, za razne algoritme:

algoritam	$T(1024)$	$T(1023)$
`f_moivre`	$2.930 \cdot 10^{-8}$	$2.920 \cdot 10^{-8}$
`f_summa`	$8.614 \cdot 10^{-7}$	$8.607 \cdot 10^{-7}$
`f_binpow`	$1.878 \cdot 10^{-8}$	$3.066 \cdot 10^{-8}$
`f_binpow_w`	$1.825 \cdot 10^{-8}$	$3.024 \cdot 10^{-8}$
`f_binpow_1`	$1.577 \cdot 10^{-8}$	$2.870 \cdot 10^{-8}$
`f_binpow_2`	$1.455 \cdot 10^{-8}$	$2.866 \cdot 10^{-8}$

Uočite da je dobro implementirano binarno potenciranje brže od DeMoivreove formule!

FIB — sve znamenke i stvarna složenost