Univerzitet u Beogradu

Matematički fakultet

ALGORITMI ZA GENERISANJE PARTICIJA

Master rad

Autor
Dejan Tomić, 1026/2014

Mentor
prof. dr Miodrag Živković

Beograd

22. septembar 2020

Sadržaj

1 Uvod

Reč particija ima više značenja u matematici. Particija predstavlja bilo kakvu podelu nekog objekta na manje podobjekte. To npr. može biti podela skupa na disjunktne podskupove, ili predstavljanje jednog broja u obliku zbira nekoliko manjih. Tema rada su particije u drugom navedenom značenju – particije (prirodnih) brojeva.

Gde god se vrši prebrojavanje ili klasifikacija diskretnih objekata, može se pronaći primena teorije particija prirodnih brojeva. U vezi sa particijama su mnogi kombinatorni problemi i problemi iz oblasti teorije brojeva. Zapravo, particije na interesantan način povezuju te dve važne grane matematike.

Teorija particija ima dug istorijat – još iz srednjeg veka se mogu naći neki određeni problemi u vezi sa particijama. Od XVIII veka Ojler postavlja moderne osnove teorije particija, dokazavši nekoliko značajnih teorema. U nastavku se navode neke ključne teoreme i formule koje su nastale kao rezultat Ojlerovih otkrića. Pored Ojlera, razvoju teorije particija puno su doprineli i Hardi, Ramanudžan, Rademaher, kao i Gaus, Jakobi, Lagranž, Ležandr i drugi poznati matematičari. [2]

Ovaj rad je najvećim delom zasnovan na Knutovoj knjizi *,,The Art of Computer Programming"* [1] – ona predstavlja izvor dosta navedenih primera, koncepata i pratećih tekstova o particijama, kao i nekoliko algoritama za nabrajanje particija prirodnih brojeva navedenih u radu.

Opšta šema koja obuhvata particije i druge kombinatorne objekte može se formulisati kao dvanaestorka problema raspoređivanja kuglica u kutije (*"Twelvefold Way"*). Tu se razmatraju različiti problemi u odnosu na to da li su kutije i/ili kuglice obeležene, te ograničenja: da li svaka kutija mora imati bar po jednu kuglicu, najviše jednu kuglicu, ili bilo koji broj kuglica. Problemi su prikazani u Tabeli 1. [1]

Broj kuglica u kutiji	Bilo koji	Najviše jedna	Barem jedna
n različitih kuglica, m različitih kutija	n-torke brojeva od 1 do m; (m,n)-varijacije	n-permutacije brojeva od 1 do m	**uređene particije skupa** $\{1..n\}$**, sa tačno m delova**
n istih kuglica, m različitih kutija	n-kombinacije s ponavljanjem brojeva od 1 do m	n-kombinacije (bez ponavljanja) brojeva od 1 do m	kompozicije broja n sa tačno m delova
n različitih kuglica, m istih kutija	**particije skupa** $\{1..n\}$ **sa najviše** m **delova**	n zečeva u m kaveza	**particije skupa** $\{1..n\}$ **sa tačno** m **delova**
n istih kuglica, m istih kutija	**particije broja** n **sa najviše** m **delova**	n klonova u m kaveza	**particije broja** n **sa tačno** m **delova**

Tabela 1: Problemi raspoređivanja kuglica u kutije

Kao što vidimo, ako su kuglice označene, a kutije nisu, dobijamo problem ekvivalentan podeli (particionisanju) skupa. Ako ni kuglice nisu označene, problem je ekvivalentan particionisanju celog broja n. Pritom, ograničenjem da svaka kutija sadrži bar jednu kuglicu dobijamo specijalan problem particionisanja, gde se traže particije sa tačno m delova.

Teorija particija ima široku primenu u teoriji polinomnih identiteta (i specijalnih funkcija uopšte). Jedan od razloga zašto je ova teorija tako bogata i zanimljiva je taj što ne postoji jednostavna formula za efikasno računanje broja particija. Hardi i Ramanudžan su postigli izvanredna dostignuća po tom pitanju; no formule nisu nimalo jednostavne, što će se videti u nastavku teksta.

U nastavku rada dati su osnovni pojmovi iz teorije particija, načini predstavljanja particija i njihova svojstva, a zatim nekoliko značajnih rezultata iz oblasti teorije brojeva i teorije particija, koji su iskorišćeni u radu – prvenstveno vezanih za broj particija, kao veoma značajni deo ove teorije.

Dati su različiti algoritmi za nabrajanje svih, ili nekih, particija prirodnih brojeva, uz analizu rada algoritama, te algoritmi za računanje broja particija. Upravo za računanje broja particija najviše značaja imaju otkrića Ojlera i drugih velikih matematičara koja su prikazana u nastavku.

Napisani su programi u jeziku C++ za nabrajanje particija i računanje broja particija, korišćenjem svih opisanih algoritama, i testirana im je realna efikasnost izvršavanjem algoritama za različite velike i male ulazne vrednosti.

2 Particije brojeva

U ovom delu definišu se particije, kao i ostali povezani pojmovi, a zatim se razmatra predstavljanje particija, njihova svojstva i teoreme vezane za particije brojeva.

2.1 Osnovni pojmovi

Definicija 1 (particija prirodnog broja)**.** Neka je n prirodan broj. Nerastući niz prirodnih brojeva $(p_1, p_2, \ldots, p_k)$ takav da je $n = p_1 + p_2 + ... + p_k$ naziva se *particijom* prirodnog broja n.

Dakle, particija prirodnog broja nije ništa drugo nego njegovo predstavljanje u obliku sume prirodnih brojeva. U okviru klasifikacije problema u Tabeli 1 radi se o varijanti u kojoj se n kuglica grupiše na neki proizvoljan način i raspoređuje u kutije (najviše n kutija), pri čemu se ni kuglice ni kutije ne razlikuju. Dakle, grupa od n kuglica se deli u više grupa koje sadrže manji broj kuglica. Svaka od tih grupa kuglica smesti se u jednu kutiju. S obzirom na komutativnost operacije sabiranja, nije bitno tačno u kojoj kutiji se nalazi koliko kuglica, već je samo bitan broj kuglica u svakoj od kutija.

Primer. Neka je $n = 6$. S obzirom na to da su svi načini da se 6 predstavi u obliku nekoliko sabiraka opisani jednakostima:

$$\begin{aligned}
6 &= 6 \\
6 &= 5+1 \\
6 &= 4+2 \\
6 &= 4+1+1 \\
6 &= 3+3 \\
6 &= 3+2+1 \\
6 &= 3+1+1+1 \\
6 &= 2+2+2 \\
6 &= 2+2+1+1 \\
6 &= 2+1+1+1+1 \\
6 &= 1+1+1+1+1+1,
\end{aligned}$$

particije broja $n = 6$ su nizovi $(6), (5,1), (4,2), \ldots, (1,1,1,1,1,1)$.

Tema ovog rada su algoritmi koji za dati prirodan broj n određuju njegove particije (neke ili sve), ili pak njihov broj. Na primer, jedan algoritam o kome će biti reči u daljem radu nalazi particije koje se sastoje od tačno k elemenata ($k \leq n$). Za takvu particiju kažemo da se sastoji od k delova.

Definicija 2. Neka je $(p_1, p_2, \ldots, p_k)$ particija prirodnog broja n. Za broj k kažemo da je *broj delova* te particije broja n.

Osim celih brojeva, i skupovi mogu da se dele na particije. Ovo odgovara donjem redu tabele 1 – slučaju sa *različitim* kuglicama i istim kutijama. Podsetimo se, particijama celih brojeva odgovaraju *iste* kuglice.

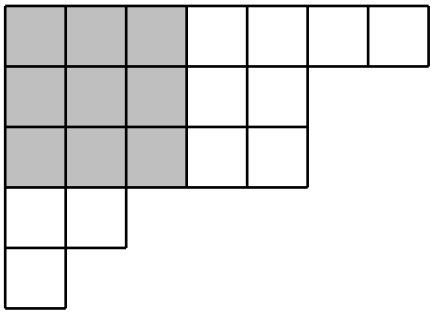
Definicija 3 (particija skupa)**.** Neka je dat neprazan konačan skup X. Pod *particijom skupa* X podrazumeva se familija njegovih podskupova $\{X_1, \ldots, X_k\} \subseteq P(X)$, pri čemu su svaka dva skupa X_i i X_j disjunktna i $\bigcup_{k=1}^{n} X_k = X$ (ovde $P(X)$ označava partitivni skup skupa X).

2.2 Fererov dijagram

Fererov dijagram predstavlja vizuelizaciju jedne particije. Kao što znamo, particija broja n predstavlja niz delova (sabiraka) čiji je zbir dati broj n. Fererov dijagram je sastavljen od tačaka ili kvadratića – jedinica, podeljenih u redove, gde svaki red predstavlja jedan deo particije (element niza). Uz to se poštuje standardni poredak delova particije – nerastući.

Dakle, Fererov dijagram za određenu particiju prirodnog broja predstavlja svaki deo particije u jednom redu, sa odgovarajućim brojem tačaka (jedinica); broj jedinica po redovima je opadajući, i njihov ukupan broj je upravo n.

Primer. Fererov dijagram particije $(7, 5, 5, 2, 1)$ broja 20 izgleda ovako:



Slika 1: Fererov dijagram

Fererov dijagram se može podeliti na 3 dela. Centralni deo – *Darfijev (Durfee) kvadrat* – predstavlja najveći „zajednički" kvadrat kome je teme gornji levi ugao dijagrama. On razdvaja deo dijagrama sa njegove desne strane i deo ispod njega. Na dijagramu u gornjem primeru osenčen je Darfijev kvadrat. Očigledno, broj delova particije predstavljene dijagramom je jednak broju redova dijagrama, dok broj kolona predstavlja vrednost najvećeg dela (sabirka).

Definicija 4 (konjugovana particija)**.** Transponovanjem Fererovog dijagrama (zamenom kolona i redova) bilo koje particije nekog prirodnog broja n dobija se Fererov dijagram njene *konjugovane particije*.

Primer. Konjugovana particija particiji $(7, 5, 5, 2, 1)$ je $(5, 4, 3, 3, 3, 1, 1)$.

Posmatrajući sliku 1, može se zaključiti da transponovanjem dijagrama Darfijev kvadrat ostaje isti, dok se delovi sa njegove desne i donje strane transponuju i zamenjuju mesta. Može se zaključiti da za svaku particiju njoj konjugovana particija ima Darfijev kvadrat iste dimenzije.

3 Broj particija

Već mnogo godina jedan od najznačajnijih problema sa particijama je određivanje njihovog broja $p(n)$. Na osnovu poznatih algoritama vrednosti $p(n)$ su do sada izračunate za vrlo velike vrednosti n. Osim određivanja samog broja particija, od interesa su i problemi određivanja broja n za koji je broj particija $p(n)$ prost, kao i broj particija broja n od tačno k delova (obeležava se sa $p_k(n)$), broj particija čiji su svi delovi neparni, različiti (strogo opadajući)... Jedan od još uvek nerešenih problema je i određivanje, u opštem slučaju, da li je $p(n)$ parno ili neparno (a da pritom ne mora da se izračuna sam broj $p(n)$).

3.1 Teorema o petougaonim brojevima

Jedan od načina da se izračuna broj particija $p(n)$ je upotrebom stepenih redova.

Teorema 1. *Beskonačni proizvod* $\prod_{n=1}^{\infty} \frac{1}{1-x^n}$ *konvergira za* $|x| < 1$. *Štaviše, taj beskonačni proizvod se može napisati u obliku stepenog reda, gde je koeficijent uy* x^n *upravo* $p(n)$, *odnosno*

$$\prod_{n=1}^{\infty} \frac{1}{1-x^n} = \sum_{n=0}^{\infty} p(n)x^n.$$

Dokaz. Za svako n je $1 - x^n > 0$, pa je i $\frac{1}{1-x^n} > 0$, pa je moguće preći iz proizvoda u sumu logartimovanjem. Dobija se $\sum_{n=1}^{\infty} \ln \frac{1}{1-x^n} = \sum_{n=1}^{\infty} -\ln(1-x^n)$. Kako za velike n važi $\ln(1-x^n) = -x^n + o(x^n)$, a red $\sum_{n=1}^{\infty} x^n$ je konvergentan, sledi da je i polazni red $\sum_{n=1}^{\infty} \ln \frac{1}{1-x^n}$ konvergentan, te je i proizvod $\prod_{n=1}^{\infty} \frac{1}{1-x^n}$ kovergentan. S druge strane, za svako n je $\frac{1}{1-x^n} = \sum_{k=0}^{\infty} (x^n)^k = \sum_{k=0}^{\infty} x^{n\cdot k} = 1 + x^n + x^{n\cdot 2} + x^{n\cdot 3} + \dots$, pa je

$$\prod_{n=1}^{\infty} \frac{1}{1-x^n} = (1 + x + x^2 + x^3 + \dots)(1 + x^2 + x^4 + \dots)(1 + x^3 + x^6 + \dots)\cdots$$

Primetimo da se proizvod može zapisati i kao suma nad $k_1, k_2, \dots$:

$$(1+x+x^2+x^3+\dots)(1+x^{2\cdot 1}+x^{2\cdot 2}+x^{2\cdot 3}+\dots)(1+x^{3\cdot 1}+x^{3\cdot 2}+\dots)\cdots = \sum_{k_1,k_2,\dots\geq 0} x^{k_1+2k_2+3k_3+\dots}$$

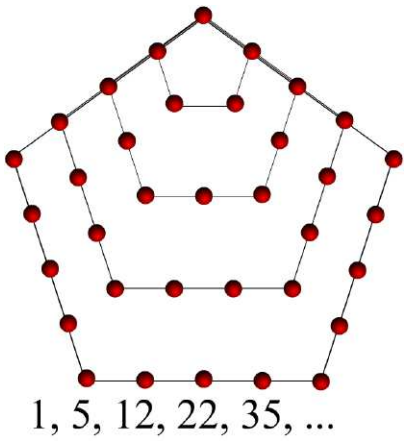
gde su $k_1, k_2, \dots \geq 0$ prirodni brojevi.

Iz ovog zapisa direktno sledi da je koeficijent uz x^n upravo broj načina da se n zapiše kao suma $k_1 + 2k_2 + 3k_3 + \dots$; primetimo pritom da za svako $s > n$ mora da bude $k_s = 0$. Ovakav zapis broja n je, međutim, samo jedna od predstava njegove particije; traženi broj načina je time $p(n)$ – broj particija broja n. □

Data formula se, s obzirom na pokazani identitet, može koristiti kao *generatrisa* niza $p(n)$. Očigledno, dovoljno je izračunati proizvod prvih n činilaca i to samo do stepena x^n kako bi se dobila vrednost $p(n)$ (kao i $p(k)$ za svako $k < n$).

Veoma značajna za računanje broja particija je *teorema o petougaonim brojevima* [2, str. 10-13]. Petougaoni brojevi su brojevi oblika $g_k = \frac{3k^2-k}{2}$, za svaki ceo broj k (i pozitivan i negativan); g_k je *k-ti petougaoni broj*. Razlika uzastopnih petougaonih brojeva je

$$\begin{aligned} g_{k+1} - g_k &= \frac{3(k+1)^2 - (k+1)}{2} - \frac{3k^2 - k}{2} \\ &= \frac{3(k+1-k)(k+1+k) - k - 1 + k}{2} \\ &= \frac{3(2k+1) - 1}{2} = \frac{6k+3-1}{2} \\ &= 3k + 1. \end{aligned}$$



Slika 2: Petougaoni brojevi

Teorema 2 (o petougaonim brojevima)**.** *Za realan broj x, $|x| < 1$, važi sledeći identitet:*

$$\prod_{n=1}^{\infty}(1-x^n) = \sum_{k=-\infty}^{\infty}(-1)^k x^{\frac{3k^2-k}{2}} = 1 - x - x^2 + x^5 + x^7 - x^{12} - x^{15} + \dots$$

(pri čemu se za k uzimaju redom vrednosti $0, 1, -1, 2, -2, \dots$).

Dokaz. Razvojem proizvoda sa leve strane jednakosti dobija se slična suma kao u slučaju proizvoda redova $1 + x^n + x^{2n} + x^{3n} + ...$; razlika je u tome što sada svaki od brojeva $k_1, k_2, ...$ može da bude ili 0 ili 1, i što znak uz x^n zavisi od zbira izabranih k:

$$(1-x)(1-x^2)(1-x^3)\cdots = \sum_{k_1,k_2,\dots\in\{0,1\}} (-1)^{k_1+k_2+k_3+\dots} x^{k_1+2k_2+3k_3+\dots}$$

To znači da se sabirak x^n pojavljuje onoliko puta na koliko načina n može da se zapiše kao zbir *različitih* prirodnih brojeva, odnosno koliko ima particija broja n sa različitim delovima. Znak uz x^n će pritom biti $+$ ako je broj delova te particije paran, a $-$ ako je neparan.

Na osnovu toga proizvod se može zapisati kao sledeća suma:

$$(1-x)(1-x^2)(1-x^3)\cdots = \sum_{n=1}^{\infty}(p_e(n)-p_o(n))x^n,$$

gde je $p_e(n)$ broj particija broja n sa parnim brojem različitih delova, a $p_o(n)$ sa neparnim brojem različitih delova.

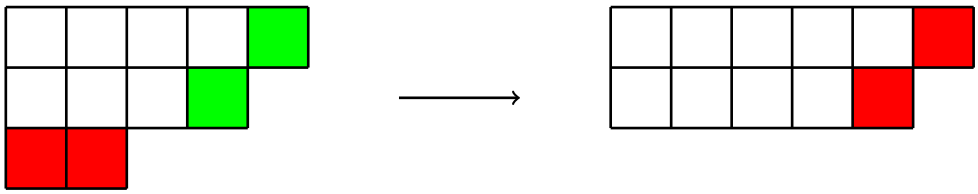
Za dokaz teoreme je dovoljno još pokazati da je $p_e(n)-p_o(n)=0$ za svako n koje nije petougaoni broj, te utvrditi vrednost ovog izraza za petougaone brojeve n.

Neka je data particija broja n sastavljena od k različitih delova i predstavljena Ferrerovim dijagramom. Označimo sa p njen najmanji deo (broj kolona poslednje vrste dijagrama), i sa q broj prvih elemenata particije koji su uzastopni brojevi. Ovo je osenčeno na dijagramu ispod (slika 3).

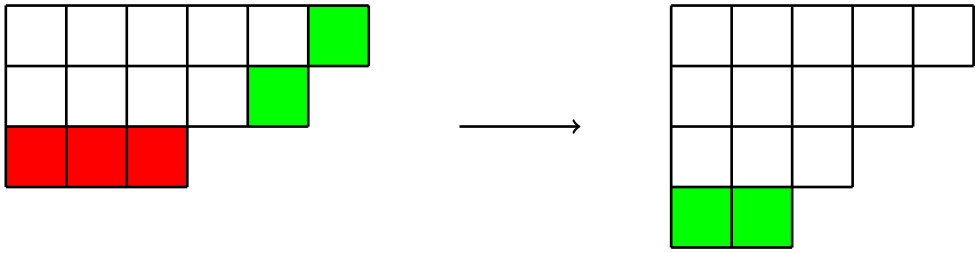
Postoje sada dva opšta slučaja:

1. U slučaju $p \leq q$ može se „premestiti" donji red na desnu stranu dijagrama (kao na slici 3) i dobija se ponovo particija sa različitim delovima;

2. Ako je, međutim, $p > q$, desna dijagonala dijagrama (poslednja polja prvih q redova) se može premestiti na dno dijagrama (slika 4) i time, opet, dobiti particija sa različitim delovima.



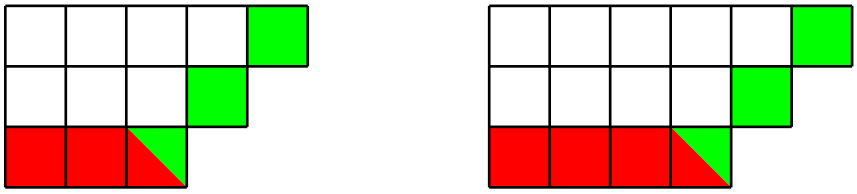
Slika 3: Particija $(5,4,2)$ $(p=2,q=2)$ i njoj odgovarajuća $(6,5)$



Slika 4: Particija $(6,5,3)$ $(p=3,q=2)$ i njoj odgovarajuća $(5,4,3,2)$

Primetimo da u prvom slučaju, nakon „premeštanja" vrednost q nove particije postaje jednaka prethodnom p; to je manje od novog p s obzirom na to da su svi delovi particije različiti. Dakle, u novoj particiji važi $p > q$. Izuzetak je $p = q = k$ (slika 5), kada ne možemo „premestiti" ceo donji red; jedno polje bi moralo ostati na mestu, te bi u novoj particiji bilo $p = 1 \leq q$.

U drugom slučaju „premeštanjem" q polja na dno dijagrama vrednost q se u dobijenoj particiji ne može smanjiti; sa druge strane vrednost p postaje jednaka staroj vrednosti q, tako da važi $p \leq q$ odnosno slučaj 1. U slučaju $p = q+1$ i $q = k$ (slika 5), međutim,



Slika 5: Specijalan slučaj: particije $(5,4,3)$ $(p=3,q=3)$ i $(6,5,4)$ $(p=4,q=3)$

dobija se particija sa poslednja dva jednaka dela. Ovakve particije ne razmatramo prilikom računanja koeficijenta uz x^n iz tvrđenja teoreme.

Dakle, osim u dva navedena specijalna slučaja, particije koje ispunjavaju slučajeve 1 i 2 se mogu upariti – transformacija je bijektivna. Pritom se broj delova nove particije u slučaju 1 smanjuje za jedan, a u slučaju 2 povećava za jedan – u oba slučaja parnost broja delova uparenih particija se razlikuje, što znači da je $p_e(n) = p_o(n)$.

Sada ostaje istražiti dva navedena specijalna slučaja: u slučaju $p = q = k$ važi

$$n = p + (p+1) + \cdots + (p+k-1) = k \cdot \frac{p+p+k-1}{2} = \frac{3k^2-k}{2}$$

a slučaju $p = k+1, q = k$ je

$$n = p + (p+1) + \cdots + (p+k-1) = k \cdot \frac{k+1+2k}{2} = \frac{3k^2+k}{2}$$

U oba slučaja je n petougaoni broj – u prvom slučaju $n = g_k$, a u drugom $n = g_{-k}$. Ovde se može upotrebiti i činjenica sa početka dokaza da particije sa parnim brojem delova povećavaju koeficijent uz x^n, a sa neparnim smanjuju – u slučaju parnog k je $p_e(g_k) - p_o(g_k) = 1$, a u slučaju neparnog je -1.

Time je, konačno, dokazano da je

$$\prod_{n=1}^{\infty}(1-x^n) = \sum_{k=-\infty}^{\infty}(-1)^k x^{g_k}$$

□

Iz Ojlerove jednakosti za proizvod $\frac{1}{1-x^n}$ (generatrise $p(n)$), kao i teoreme o petougaonim brojevima, direktno sledi $(p(0)=1)$:

$$(1 + p(1)x + p(2)x^2 + p(3)x^3 + \cdots) \cdot (1 - x - x^2 + x^5 + x^7 - \dots) = 1.$$

Kao direktna posledica poslednje jednakosti dobija se sledeća rekurentna formula za broj particija, koju su izveli Mek Mahon i Ramanudžan [3]:

$$p(n) = p(n-1) + p(n-2) - p(n-5) - p(n-7) + p(n-12) + \dots$$

Znak $+$ ide uz sabirke $p(n-g_k)$ za petougaoni broj g_k sa parnim (pozitivnim ili negativnim) indeksom k, a znak $-$ za neparne indekse k.

Pošto je $p(n) = 0$ za svako $n < 0$, broj sabiraka u ovom izrazu je konačan (manji od n) za svako n, na osnovu čega se može efikasno izračunati $p(n)$.

3.2 Asimptotska formula za $p(n)$

Asimptotska svojstva niza $p(n)$ odredili su Hardi, Ramanudžan i Rademaher. Oni su otkrili sledeću asimptotsku formulu za aproksimaciju broja particija [4, str. 14]:

$$p(n) \sim \frac{1}{4n\sqrt{3}} e^{\pi\sqrt{\frac{2n}{3}}} \quad (n \to \infty)$$

Pritom data formula predstavlja gornju granicu za $p(n)$.

Kasnije nakon ove aproksimacije Hardija i Ramanudžana, Rademaher je „poboljšao" formulu i došao do sledeće, **egzaktne** formule za $p(n)$:

$$p(n) = \frac{1}{\pi\sqrt{2}} \sum_{k=1}^{\infty} A_k(n)\sqrt{k} \frac{d}{dn} \left(\frac{\sinh\left(\frac{\pi}{k}\sqrt{\frac{2}{3}(n-\frac{1}{24})}\right)}{\sqrt{n-\frac{1}{24}}} \right)$$

Ovde je $A_k(n)$ funkcija definisana kao

$$A_k(n) = \sum_{\substack{h=0 \\ (h,k)=1}}^{k-1} e^{i\pi s(h,k) - 2i\pi n\frac{h}{k}}$$

(gde su h i k uzajamno prosti), a $s(h,k)$ je *Dedekindova suma*:

$$s(h,k) = \sum_{r=1}^{k-1} \frac{r}{k} \left(\frac{hr}{k} - \left\lfloor \frac{hr}{k} \right\rfloor - \frac{1}{2} \right)$$

Prvi član ovog razvoja za $p(n)$ je upravo gornja aproksimacija.

S obzirom na to da se zna da je $p(n)$ uvek ceo broj, razvoj se može zaustaviti nakon što greška vrednosti postane dovoljno mala – ovo se dešava nakon $O(\sqrt{n})$ članova. Ovo znači da, pored toga što je formula **egzaktna**, ona je i **konačna**! Ovo je jedno od najznačajnijih dostignuća po pitanju particija prirodnih brojeva, s obzirom na to da omogućava efikasno računanje njihovog broja.

3.3 Osnovna teorema o broju particija

Postoji više teorema koje utvrđuju određena pravila ili veze broja svih particija, ili particija sa određenim svojstvima (npr. broj sabiraka, parnost, maksimalni sabirak itd). Ispod se navodi jedna jednostavna takva teorema. [2]

Teorema 3. *Broj particija prirodnog broja n čiji je najveći sabirak jednak k, jednak je broju particija broja n sa tačno k delova.*

Dokaz. Transponovanjem Fererovog dijagrama za datu particiju broja n dobija se druga, *dualna* particija broja n. Ako je najveći sabirak prve particije jednak k (što je broj kolona dijagrama), onda je broj delova druge particije (što je broj vrsta dijagrama) jednak k. Transponovanje dijagrama je uzajamno jednoznačna operacija, pa ona uspostavlja bijekciju između skupa svih particija broja n i skupa njihovih dualnih particija. Iz ove činjenice sledi dokaz teoreme. □

4 Algoritmi

U ovom poglavlju opisuje se nekoliko algoritama za generisanje (ispis) particija celih brojeva, zajedno sa pseudokodom i kratkom analizom. Nakon algoritama koji ispisuju sve particije datog celog broja n (ili pak particije sa određenim svojstvom) opisuju se i algoritmi koji računaju $p(n)$ – broj particija broja n, te će biti reči o njihovoj složenosti, odnosno efikasnosti.

4.1 Rekurzivni algoritmi za nabrajanje particija

Najjednostavniji algoritam za nabrajanje je rekurzivni: za dato n uzeti, redom, brojeve od 1 do n za prvi sabirak (označimo ga sa x); za svaki izabrani broj x ponoviti algoritam za broj $n - x$.

Izgleda lako. Međutim, ovo nije dovoljno. Mora se voditi računa i o **redosledu** sabiraka, kako se ne bi više puta dobila ista particija. Po definiciji 1 (particija prirodnog broja) taj redosled je nerastući – dakle, posle izabranog sabirka mora se voditi računa da sledeći ne bude veći od njega.

Problem se rešava uvođenjem drugog argumenta, gornje granice za veličinu delova particije. Time funkcija sada ne nalazi sve particije prirodnog broja n ($P(n)$), već samo one čiji je **najveći deo** manji ili jednak od datog broja k; neka je taj skup označen sa $P(n, k)$.

Kako sada sastaviti $P(n, k)$? Na način sličan gore opisanom: uzeti za prvi sabirak, redom, brojeve od 1 do k (neka je taj broj x) i za svaki od njih naći particije broja $n - x$ sa najvećim sabirkom **najviše** x; odnosno $P(n, k) = \bigcup_{x=1}^{k} P(n - x, x)$.

Sada je još potrebno ispisati sve dobijene particije. Particija je kompletna tek kada se za x uzme n, odnosno kad je prvi argument u narednom pozivu jednak 0. Rekurzijom se dobijaju particije manjih brojeva, pa je potrebno svaku takvu particiju „dopisati" na prethodne delove – sabirke $n - x$ iz steka rekurzivnih poziva. Zbog toga je neophodno imati uvid u prethodne sabirke – odnosno, pamtiti sve delove trenutne particije na koju se „dopisuju" particije manjeg broja.

Međutim, u tom slučaju broj k jednak je poslednjem (najmanjem) delu particije do sada, pa ga nije potrebno posebno navoditi. Konačan pseudokod bi onda bio sledeći:

```
partitions_rec(n, m, P):
// niz P sadrzi prethodne sabirke, m je njihov broj
        if n = 0:
                print P
                return
        k ← { n,          ako je m = 0
            { min(n, P_m), u protivnom
        for x in [1, k]:
                P_{m+1} ← x // prosirujemo niz P elementom m+1
                partitions_rec(n − x, m + 1, P)
```

Sve particije broja n dobijaju se pozivom `partitions_rec(n,0,())`. Algoritam nabraja particije u *leksikografskom poretku* – počevši od particije $(1, 1, 1, \ldots, 1)$, a završava particijom sa jednim sabirkom (n). Za *obrnuti leksikografski* poredak, opisani algoritam je moguće izmeniti tako da krene od $x = k$ pa ga smanjuje do 1.

Particije broja n se mogu nabrojati i metodom matematičke indukcije polazeći od particija broja $n-1$. Počnimo od baznog slučaja $n = 0$: broj particija broja 0 se definiše kao 1; pritom je jedina particija broja 0 prazan niz: $()$. Zatim, za svaku particiju $(p_1, p_2, \ldots, p_k)$ broja $n-1$ na najviše dva načina se mogu dobiti particije broja n:

1. Dodavanjem 1 na kraj dobija se particija $(p_1, p_2, \ldots, p_k, 1)$.

2. Ako je $p_{k-1} > p_k$ (za $k > 1$) ili $k = 1$, povećati p_k za 1, čime se dobija particija $(p_1, p_2, \ldots, p_k + 1)$.

Drugim rečima, ako particija p zadovoljava uslov $k = 1$ ili ($k > 1$ i $p_{k-1} > p_k$), onda postoje dve particije broja n koje počinju sa p, a u protivnom postoji samo jedna takva particija, koja se dobija dopisivanjem jedinice.

Pseudokod odgovarajućeg rekurzivnog algoritma je sledeći:

```
partitions_ind(n):
        if n = 0:
                return {()}
        P ← {}
        for each (p_1, p_2, ..., p_m) in partitions_ind(n − 1):
                P ← P ∪ (p_1, p_2, ..., p_m, 1)
                if m > 0 and (m < 2 or p_{m−1} > p_m):
                        P ← P ∪ (p_1, p_2, ..., p_{m−1}, p_m + 1)
        return P
```

Primer. Neka je $n = 4$. Particije broja 4 su $(1,1,1,1), (2,1,1), (2,2), (3,1)$ i (4). Sada treba naći particije broja 5 polazeći od navedenih particija.

- Od particije $(1,1,1,1)$ se dobija particija $(1,1,1,1,1)$ na način 1; način 2 nije primenljiv s obzirom na to da su poslednja dva člana jednaka ($m = 4, p_4 = p_3$).

- Slično, ni na particiju $(2,1,1)$ ne može da se primeni način 2 ($m = 3, p_3 = p_2$). Dobija se samo jedna nova particija $(2,1,1,1)$.

- Isti slučaj je i sa particijom $(2,2)$ ($p_2 = p_1 = 2$), pa se od nje dobija samo $(2,2,1)$.

- Od particije $(3,1)$ se dobija particija $(3,1,1)$, a s obzirom na $m = 2, 1 = p_2 < p_1 = 3$ dobija se i $(3,2)$.

- Na isti način se i od particije (4) dobijaju $(4,1)$ i (5) (s obzirom na $m = 1$).

Može se proveriti da su dobijene sve particije broja 5.

Particije broja 4 su u primeru namerno date u leksikografskom poretku. Može se uočiti da su u istom poretku dobijene i particije broja 5 izvršavanjem *partitions_ind* algoritma. Ispostavlja se da algoritam *partitions_ind*, baš kao i prethodno opisan *partitions_rec*, za svako n ispisuje particije upravo u leksikografskom poretku.

Teorema. *Algoritam* partitions_ind *generiše sve particije broja n tačno po jednom.*

Dokaz. Razmotrimo prvo opisana dva načina za dobijanje particija broja n od particija broja $n-1$.

- U slučaju načina 1, očigledno je da je zbir brojeva u nizu opisanim postupkom povećan za 1, kao i da se ne mogu dobiti dve iste particije broja n na taj način (inače bi i odgovarajuće particije broja $n-1$ bile identične).

- U slučaju načina 2, opisani postupak je dozvoljen s obzirom na to da je poslednji deo particije *strogo manji*; te se uvećavanjem za 1 ne bi narušio nerastući poredak delova. I u ovom slučaju je očigledno da se ne mogu dobiti dve iste particije broja n, jer bi onda i particije broja $n-1$ od koga su nastale bile iste.

Particije dobijene na način 1 i 2 nikako ne mogu biti iste, jer je poslednji (najmanji) deo particije u slučaju 1 uvek jednak 1, a u slučaju 2 uvek veći od 1.

Može se primetiti da ovo nisu sve mogućnosti za dobijanje particije broja n od particije broja $n-1$. Kao što se poslednji deo može povećati za 1 ako je strogo manji od pretposlednjeg, tako se i bilo koji drugi deo particije, a koji je strogo manji od prethodnog, može povećati za 1. Međutim, smanjivanjem *poslednjeg* dela takve particije za 1 (ako je veći od 1), odnosno uklanjanjem broja 1 sa njenog kraja dobija se druga particija broja $n-1$. Dakle, *svaka* particija broja n se može dobiti od *neke* particije broja $n-1$ na neki od dva načina opisana iznad; pri tome je, s obzirom na prethodni pasus, *jedinstveno* određena particija broja $n-1$ od koje se ona dobija.

Time je konačno dokazan induktivni korak: na opisani način se, počevši od svih particija broja $n-1$, dobijaju *sve* particije broja n i to svaka *tačno jednom*. Zajedno sa bazom indukcije (na osnovu koje se dobija da je (1) jedina particija broja 1, što je tačno) ovime je dokazana ispravnost opisanog postupka nabrajanja particija. □

4.2 Iterativni algoritmi za nabrajanje

Prethodno navedeni algoritmi za nabrajanje particija koriste rekurziju. Rekurzija ume da bude memorijski zahtevna, s obzirom na pamćenje stanja svih pozivaoca. Na primer, gornja granica memorijske složenosti algoritma *partitions_rec* je $O(n)$, s obzirom na to da particija broja n može da ima do n sabiraka (sve jedinice), a u svakom pozivu *partitions_rec* dodaje (i na kraju uklanja) po jedan sabirak u trenutnu particiju. Za algoritam *partitions_ind* je složenost još gora – pod pretpostavkom da je ceo skup particija broja $n-1$ već izračunat (prilikom poziva `partitions_ind(n-1)`), zauzeće memorije jednako je prostoru potrebnom za čuvanje *svih* particija broja $n-1$, čiji je samo broj eksponencijalan u odnosu na n. Čak i da poziv `partitions_ind` vraća jednu po jednu particiju (o implementaciji takvog načina rada biće reči u naslovu 5), memorijsko zauzeće tog algoritma može biti i do $O(n^2)$, s obzirom na to da se na steku čuva do n prethodnih particija (po jedna za svaki broj od 1 do n), a kao i u algoritmu *partitions_rec*, particija sastavljena od n jedinica zauzima memorijski prostor $O(n)$.

Jedan način za nabrajanje svih particija bez korišćenja rekurzije, u obrnutom leksikografskom poretku, je sledeći: početi od particije (n), zatim u svakoj iteraciji smanjiti za 1 poslednji sabirak particije veći od 1 (oni su dati u nerastućem poretku, u skladu sa definicijom 1), a jedinice nakon njega zameniti najvećim mogućim brojevima. Dakle, $(\ldots, x, 1, 1, \ldots, 1)$ postaje $(\ldots, x-1, x-1, \ldots, x-1, r)$, gde su svi sabirci osim poslednjeg jednaki $x-1$ i je r ostatak ($r < x$). [1]

Algoritam koji sledi generiše particije datog celog broja u obrnutom leksikografskom poretku:

`revlex_partitions(`n`):`

```
// particije su (a_1, a_2, ..., a_m), m ≤ n
a_0 ← 0, a_1 ← n, m ← 1 // inicijalna particija - (n)
while true:
        print (a_1, a_2, ..., a_m)
        q ← max{k|a_k > 1}
        // q je indeks poslednjeg elementa veceg od 1
        while a_q = 2: // korak 2 → 1,1
                a_q ← 1
                m ← m + 1, a_m ← 1
                print (a_1, a_2, ..., a_m)
                q ← q − 1
        if q = 0: // kriterijum zaustavljanja - (1, 1, ..., 1)
                return
        a_q ← a_q − 1
        r ← m − q + 1
        m ← q + 1
        while r > a_q: // korak x, 1, ..., 1 → x − 1, ..., x − 1, r
                a_m ← a_q, m ← m + 1
                r ← r − a_q
        a_m ← r
```

Primer. Neka je $n = 6$.

- Algoritam počinje sa inicijalnom particijom (6) na početku spoljne `while` petlje. Očigledno je poslednji, pa i jedini, deo veći od 1 (i jedini deo uopšte) $n = 6$, te je $q = 1$.
- Unutrašnja `while` petlja (korak $2 \to 1, 1$) se u potpunosti preskače, s obzirom na to da je $a_1 \neq 2$.
- q ostaje jednak 1; algoritam nastavlja sa izvršavanjem, dobija se $a_1 = 5, r = 1, m = 2$, donja `while` petlja se preskače s obzirom na $r < a_1$ i dobija se $a_2 = 1$.
- U sledećoj iteraciji spoljne petlje ispisuje se dobijena particija $(5, 1)$; kao i u prethodnoj iteraciji, i ovaj put se dobija $q = 1$, preskače se korak $2 \to 1, 1$ i dobija se particija $(4, 2)$.
- U trećoj iteraciji se dobija $q = 2$, s obzirom na $a_2 = 2 > 1$; ispisuje se particija $(4, 1, 1)$, nakon čega se izlazi iz gornje unutrašnje petlje sa $q = 1, m = 3$.
- Pre donje petlje vrednosti promenljivih su $a_1 = 3, r = 3, m = 2$; pošto je $r = a_1$ dobija se $a_2 = 3$, odnosno particija $(3, 3)$.
- U četvrtoj iteraciji spoljne petlje je sada $q = 2$ i preskače se korak $2 \to 1, 1$; dobija se $a_2 = 2, r = 1, m = 3$ i $a_3 = 1$.
- Nakon ispisa particije $(3, 2, 1)$ se u gornjoj unutrašnjoj petlji ispisuje još i particija $(3, 1, 1, 1)$; nakon nje se dobija $a_1 = 2, r = 4, m = 2$;
- Po prvi put se ulazi u donju petlju s obzirom na $r > a_1$; u njoj se dobija $a_2 = 2$, a nakon nje $a_3 = 2$, odnosno particija $(2, 2, 2)$.

- U poslednjoj iteraciji je $q = 3$ i $a_3 = a_2 = a_1 = 2$; q se smanjuje sve do nule, umanjujući jedan po jedan deo particije unazad i dopisujući jedinice na kraj. Dobijaju se particije $(2, 2, 1, 1)$, $(2, 1, 1, 1, 1)$ i konačno $(1, 1, 1, 1, 1, 1)$, nakon čega se usled $q = 0$ završava sa radom.

Time je dobijeno ukupno 11 particija, koliko ih i ima broj 6, u redosledu za koji se može uočiti da je obrnuti leksikografski.

Teorema. *Algoritam* revlex_partitions *generiše sve particije broja n i to u obrnutom leksikografskom poretku.*

Dokaz. U opisu pseudokoda iznad pokazano je da je prva particija koju algoritam ispisuje (n). Poslednja particija koju algoritam ispisuje mora biti $(1, 1, ...1)$, s obzirom na to da nijedan drugi uslov zaustavljanja algoritma ne postoji. Ove dve particije su, očigledno, leksikografski najveća i najmanja particija broja n.

Sada ostaje pokazati da je particija generisana od prethodno ispisane unutar `while` petlje zaista sledeća leksikografski najveća.

Nakon ispisa particije (`print` u prvoj liniji petlje), prvo se traži q kao indeks poslednjeg elementa većeg od 1. Tokom koraka $2 \to 1, 1$ taj element je jednak 2 dok je nakon tok koraka veći – ali u oba slučaja je poslednji deo particije kome se može smanjiti vrednost. Unutrašnja `while` petlja (na dnu pseudokoda) obezbeđuje da dalji delovi imaju najveće moguće vrednosti, te je, s obzirom na nerastući redosled delova particije, jasno da je dobijena najveća moguća particija (leksikografski) manja od prethodne. □

Ovo je jedan od najjednostavnijih iterativnih načina generisanja svih particija datog prirodnog broja, a pritom je i prilično efikasan (u smislu prosečnog vremena generisanja jedne particije). Na primer, jedan faktor koji utiče na efikasnost je slučaj $x = 2$: u tom slučaju ne mora da se „računa" koliko puta treba dopisati $x-1$ ili koliko je r – ne mora ni da se vrši nikakav poseban „ispis", odnosno izmena celog niza delova particije počevši od x – dovoljno je samo postaviti x na 1 i dopisati još jednu jedinicu na kraj niza! Ovakve particije su, pritom, veoma česte – skoro 80% particija broja 100, na primer. [1]

4.3 Generisanje particija sa tačno m delova

Ako je potrebno naći sve particije datog prirodnog broja, a koje se sastoje od tačno m delova, za to postoji sledeći iterativan algoritam, koji particije daje u leksikografskom poretku gledane unazad – dakle od *najmanjeg* dela ka najvećem – odnosno u „koleks" redosledu.

Ideja algoritma je u svakoj iteraciji naći prvi deo (od m) koji je moguće uvećati za 1 a da se delovi posle njega ne menjaju. Ovo odgovara pomenutom „koleks" redosledu particija – posmatranjem particije unazad, od m-tog dela do prvog, sledeća particija po leksikografskom redosledu bi podrazumevala najduži mogući isti „prefiks" te deo nakon prefiksa povećan za 1. Po ovome, početna particija je $(n - m + 1, 1, 1, ..., 1)$.

Kako pronaći taj deo (neka bude označen sa a_j)? Prvo se može pokušati sa $j = 2$ dok god je to moguće – ako se uveća a_2 očigledno je moguće jedino smanjiti a_1 za 1, a to je moguće jedino ako je $a_2 < a_1 - 1$.

Dalje, za $j \geq 3$, da bi bilo moguće uvećati a_j neophodan (i dovoljan) uslov je $a_j < a_1 - 1$. Traži se prvo takvo j, počevši od $j = 3$ do $j = m$. Pritom se pamti zbir s svih prethodnih delova particije, umanjen za 1 – ovo je preostali zbir koji je potrebno

„raspodeliti" na $(a_1, a_2, ..., a_{j-1})$. Da bi se dobila najmanja takva particija, po colex redosledu, primenjuje se ista tehnika kao za početnu particiju – oduzimanjem od zbira s, redom deliti istu vrednost $a_j + 1$ svim delovima osim a_1, kome se dodeljuje ostatak. (U početnoj particiji je $a_j + 1 = 1$, a ostatak je $n - m + 1$.) Zatim se ceo proces traženja ponavlja od $j = 2$.

Najveća moguća particija sa m delova po koleks redosledu podrazumeva sve delove jednake $\lfloor n/m \rfloor$, uz prvih $n \mod m$ većih za 1. Ovo je upravo jedina particija u kojoj će važiti $a_m \geq a_1 - 1$ i tada algoritam završava sa radom.

Primer. Neka je $n = 10$ i $m = 4$.

- Početak algoritma daje $n - m + 1 = 7$, te je prva dobijena particija $(7, 1, 1, 1)$.
- Za $j = 2$ se zatim dobijaju particije $(6, 2, 1, 1)$, $(5, 3, 1, 1)$ i $(4, 4, 1, 1)$.
- Nakon toga se postavlja $s = 7$ i $j = 3$. Uslov za povećanje a_j je ispunjen: $a_1 = 4, a_3 = 1$.
- j ostaje 3, pa se uvećava a_3 i njena nova vrednost postavlja na a_2, te se dobija ostatak $s = 5$.
- Ovime je dobijena particija $(5, 2, 2, 1)$ nakon čega počinje sledeća iteracija traženja najmanjeg j za uvećanje a_j.
- Sada se za $j = 2$ dobija particija $(4, 3, 2, 1)$.
- Nakon nje je $s = 6$ i $j = 3$ za koje je opet ispunjeno $a_3 < a_1 - 1$. Sledeći ciklus počinje particijom $(3, 3, 3, 1)$.
- Sada u startu nije ispunjen uslov za uvećanje a_2; najmanje j za koje je ispunjen uslov je $j = 4$ čime je $s = 8$.
- Četvrti ciklus počinje particijom $(4, 2, 2, 2)$, i daje samo još particiju $(3, 3, 2, 2)$, za $j = 2$.
- Konačno, ni za jedno $j > 1$ više nije ispunjeno $a_1 - 1 > a_j$ (jer je $a_1 = 3$ i $a_4 = 2$), te algoritam završava sa radom.

Time dobijamo ukupno 9 podela broja 10 na tačno 4 sabirka:

$$(7, 1, 1, 1), (6, 2, 1, 1), (5, 3, 1, 1), (4, 4, 1, 1), (5, 2, 2, 1), (4, 3, 2, 1), (3, 3, 3, 1),$$

$$(4, 2, 2, 2), (3, 3, 2, 2).$$

Ako bi se particije zapisale s desna nalevo, može se primetiti da bi im redosled bio leksikografski – time se potvrđuje „koleks" redosled particija.

Pseudokod datog algoritma bi bio sledeći:

```
colex_partitions(n, m):
a_1 ← n − m + 1
for j in [2, m]: a_j ← 1
a_{m+1} ← −1
while true:
```

```
print (a_1, a_2, ..., a_m)
while a_2 < a_1 - 1: // prepravljanje a_1 i a_2
        a_1 ← a_1 - 1
        a_2 ← a_2 + 1
        print (a_1, a_2, ..., a_m)
j ← 3
s ← a_1 + a_2 - 1
while a_j ≥ a_1 - 1: // nalazenje j
        s ← s + a_j
        j ← j + 1
// U ovom trenutku je s = a_1 + ··· + a_{j-1} - 1
if j > m:
        return
else: // povecavanje a_j
        x ← a_j + 1
        a_j ← x
        j ← j - 1
while j > 1: // prepravljanje do a_j
        a_j ← x
        s ← s - x
        j ← j - 1
a_1 ← s
```

4.4 Računanje broja particija

Razmotrimo sada nekoliko načina da se, umesto svih particija prirodnog broja n, izračuna samo njihov broj – $p(n)$. Postoje različite formule i algoritmi za računanje $p(n)$ u slučaju da nas ne zanimaju konkretne particije; za to se mogu iskoristiti razni identiteti u vezi particija brojeva, od kojih su neki već pokazani u radu.

Osnovni, naivni način za računanje $p(n)$ bi svakako bio isti rekurzivni algoritam za nabrajanje svih particija naveden na početku poglavlja – sa izmenom da vraća 1 pri nađenoj particiji, inače 0. Ovo je klasičan algoritam pretrage sa vraćanjem (*„backtracking"*). Kao i u algoritmu za generisanje (nabrajanje) svih particija, potrebno je voditi računa o najvećem dozvoljenom broju (delu particije) kako bi se ispoštovao redosled delova i izbeglo generisanje istih particija više puta. Jedan odgovarajući pseudokod za to je sledeći:

```
p_rec(n, max):
        if max = 0:
                return 0
        if n = 0:
                return 1
        else if n < 0:
                return 0
        return p_rec(n, max - 1) + p_rec(n - max, max)
```

Ovaj algoritam je, međutim, veoma neefikasan: s obzirom na to da se u ovom slučaju particije broje jedna po jedna (algoritam je ekvivalentan algoritmu `partitions_rec`, ali

računa samo broj dobijenih particija i zanemaruje njihov ispis), broj operacija koje vrši gornji kod je direktno proporcionalan izlaznom broju particija, odnosno $p(n)$. Ovo čini vremensku složenost algoritma `p_rec` eksponencijalnom; za nabrajanje particija bi to bila optimalna složenost, s obzirom na eksponencijalni rast $p(n)$, ali sama vrednost $p(n)$ se može izračunati i bez nabrajanja jedne po jedne particije.

Za računanje broja particija može se, umesto naivne pretrage, koristiti dinamičko programiranje (DP): za svaki ulaz rekurzivne funkcije (dakle, za svako n kao i najveći sabirak k) pamtimo rezultat i ponovo ga koristimo pri sledećem pozivu za iste argumente. Modifikacija koda bi onda izgledala ovako:

```
p_memo(n, max):
        if max = 0:
                return 0
        if n = 0:
                return 1
        else if n < 0:
                return 0
        if T_{n,max} = 0:
                T_{n,max} ← p_dp(n, max - 1) + p_dp(n - max, max)
        return T_{n,max}
```

gde je T globalno dostupna tabela izračunatih rezultata (brojeva particija za date n i max).

Ovo je najjednostavniji način rada algoritma dinamičkog programiranja – memoizacija. Često se, međutim, kod DP algoritama vrednosti tabele iterativno određuju, umesto rekurzivnih poziva (gornji kod je praktično isti kao prvi algoritam pretrage, ali sa „odsecanjem"). Takav kod bi bio malo brži i malo manje memorijski zahtevan usled nepostojanja steka (koji generiše rekurzija), mada je vremenska složenost i dalje istog reda. Novi pseudokod za računanje broja particija metodom DP je sledeći:

```
p_dp(n, max):
        for i in [1, n]:
                for j in [1, max]:
                        if i - j < 0:
                                T_{i,j} ← T_{i,j-1}
                        else:
                                T_{i,j} ← T_{i,j-1} + T_{i-j,j}
        return T_{n,max}
```

Tabela T opet ima isto značenje kao u prethodnom algoritmu; može se primetiti da su izrazi za pojedinačne članove tabele isti kao u prethodnoj rekurzivnoj varijanti algoritma.

Gornji kod vrši u proseku $O(n^2)$ sabiranja, koja su pritom sve sporija s obzirom na rast veličine sabiraka ($p(n)$ raste eksponencijalno u odnosu na n). Međutim, računanjem $p(n)$ pomoću tog algoritma istovremeno se dobija i $p(k)$ za svako $k <= n$.

Iako je vremenska složenost algoritma `p_memo`, odnosno `p_dp`, mnogo bolja od prethodnog naivnog algoritma `p_rec`, memorijska složenost $O(n^2)$, određena veličinom tabele `T`, je prilično velika – npr. za $n = 10^5$ tabela bi zauzela nekoliko desetina gigabajta memorije. Za razliku od toga, memorijska složenost algoritma `p_rec` je $O(n)$, pri čemu se uzima u obzir stek koji nastaje usled rekurzije, a može biti do dubine n.

Postoje još brži načini za računanje broja particija. Naime, $p(n)$ ne mora da se računa

preko $p(n-1), p(n-2), \ldots, p(n-k)$. Otkrivene su matematičke formule koje računaju $p(n)$ pomoću manjeg broja sabiraka ili čak direktno.

Na primer, u poglavlju 3 je pokazano da se, na osnovu Ojlerovih stepenih redova i teoreme o petougaonim brojevima, kao posledica dobija rekurentna relacija za broj particija:

$$p(n) = p(n-1) + p(n-2) - p(n-5) - p(n-7) + p(n-12) + \ldots$$

Ovde se $p(n)$ dobija samo na osnovu vrednosti $p(n-k)$ za nekoliko različitih k; najveći sabirak se više ne uzima u obzir, čime je memorijska složenost smanjena sa $O(n^2)$ na $O(n)$. Vremenska složenost, međutim, nije linearna: broj sabiraka (vrednosti iz tabele) koji se uzima pri računanju $p(n)$ je $O(\sqrt{n})$, s obzirom na stepene x u Ojlerovoj sumi iz koje rekurentna relacija za broj particija direktno sledi. Time ceo algoritam vrši $O(n\sqrt{n})$ sabiranja, odnosno ukupno $O(n^2)$ operacija, što je ipak brže od do sada opisanih algoritama.

Konačno, primenom asimptotske formule Hardija, Ramanudžana i Rademahera (podsetimo se da je to *egzaktna* i *konačna* formula za $p(n)$), pojedinačne vrednosti $p(n)$ se mogu dobiti u optimalnoj vremenskoj složenosti: $O(\sqrt{n})$. [5] Ovo je definitivno optimalna složenost s obzirom na to da $p(n)$ ima u proseku $\sqrt{n}$ cifara, te je toliko operacija potrebno za sam ispis rezultata.

Asimptotska formula predstavlja najbrži način da se izračuna $p(n)$ za pojedinačno n. Prethodna, Mekmahonova formula – rekurentna veza – računanjem $p(n)$ već dobija i vrednosti $p(k)$ za svako $1 \leq k \leq n$ – međutim, s obzirom na optimalnu vremensku složenost koju je moguće postići Hardi-Ramanudžan-Rademaher formulom, čak se i niz tih vrednosti brže može dobiti njenom primenom na svako k pojedinačno ($O(n\sqrt{n})$ umesto $O(n^2)$ ukupno operacija).

5 Programska realizacija i rezutati

Razmotreni su različiti algoritmi za nabrajanje particija, kao i za računanje broja particija datog prirodnog broja – $p(n)$, i procenjena im je vremenska složenost. Koliko je brzo, međutim, u praksi moguće izračunati $p(n)$ primenom ovih algoritama?

Mnoge matematičke operacije (počevši od samog sabiranja prirodnih brojeva) iz pseudokoda zahtevaju određeno vreme za izvršavanje na računaru. Neki problemi implementacije datih algoritama bi bili, na primer, kako predstaviti jednu particiju (koju strukturu podataka koristiti radi maksimalne efikasnosti?) ili kako predstaviti broj particija $p(n)$. Već je ustanovljeno da broj $p(n)$ ima u proseku $\sqrt{n}$ cifara – dakle, standardni numerički tipovi u računaru (koji su ograničeni uglavnom na 32 ili 64 binarne cifre, ili nešto manje značajnih cifara u slučaju decimalnih brojeva) nisu dovoljni za (precizno) predstavljanje broja particija. Potreban je niz veličine $O(\sqrt{n})$, te isto toliko operacija za svako sabiranje dva broja ovih veličina.

Za rad sa velikim brojevima postoje mnogobrojne gotove biblioteke. Za implementaciju algoritama opisanih u ovom radu odabrana je GNU Multi Precision (GMP) biblioteka [6]. Prednosti ove biblioteke su jednostavnost korišćenja (o čemu će biti reči u podnaslovu 5.2) i široka dostupnost – u sistemu korišćenom za kompajliranje i testiranje programa (Arch Linux) je već bila instalirana. GMP biblioteka i zaglavlja za kompajler se lako instaliraju i na Windows platformi.

Za predstavljanje jedne particije se može koristiti niz (promenljive dužine – *dinamički niz* – s obzirom na to da njegova dužina može biti i do n) ili povezana lista. Ispostavlja

se da je za potrebe svih do sada opisanih algoritama dovoljno koristiti niz – u C++ je to klasa `std::vector` – s obzirom na to da nema „umetanja" sabiraka u sredinu ili na početak particije, već isključivo na kraj. Korišćenjem niza umesto povezane liste je zapravo efikasniji pristup elementima – $O(1)$ za pristup bilo kom elementu (*random access*).

Konačno, ostaje pitanje kako predstaviti ulazne vrednosti algoritama. Ovde je osnovni argument sam broj n; s obzirom na veličinu $p(n)$, verovatno nema svrhe računati ovaj broj, a još manje nabrajati particije, za preveliku vrednost broja n (razumna gornja granica bi bila reda veličine 10^7 za primenu Mekmahonove rekurentne formule) – time je dovoljno ovaj argument predstaviti kao standardni neoznačeni ceo broj od 32 bita (u C++ tip `unsigned int`).

Mora se, međutim, voditi računa o nekim uslovima navedenim u pseudokodu određenih algoritama. Neki rekurzivni algoritmi, na primer, kao uslove zaustavljanja imaju $n < 0$ ili $m < 0$ – pošto negativne vrednosti nisu dozvoljene za neoznačene brojeve i izazivaju prekoračenje, mogu se koristiti ili označeni brojevi, ili prepraviti uslovi tako da ne dozvole pomenuto prekoračenje. U implementacijama, koje će biti date u daljem tekstu, izabrano je drugo navedeno rešenje.

Poslednja stavka vezana za implementaciju algoritama za nabrajanje particija je funkcija `visit`, navedena kao `print` u pseudokodovima. Ime „visit" je dato s razlogom: posao algoritma je da generiše sve particije (ili, pak, neki podskup particija) broja n, a na koji će način one biti prikazane krajnjem korisniku je druga stvar. Možda ih je potrebno ispisati u fajl, u neko tekstualno polje (za programe sa grafičkim korisničkim interfejsom), ili za svaku particiju izvršiti neku radnju – za tu svrhu se definiše interfejs funkcije `visit`, čija se konkretna implementacija daje kao ulazni podatak u svaki algoritam.

Interfejs ka funkciji „visit" je definisan kao tip `part_visitor`, koji predstavlja parametrizovani šablon klase `std::function` iz standardne C++ biblioteke. Instance ove klase se mogu pozvati na isti način kao i obične funkcije u jeziku C++ – sintaksa je ista zahvaljujući predefinisanju operatora koje C++ podržava. U ovom slučaju parametri šablona su povratni tip i tipovi argumenata funkcije. Funkcija `visit` prihvata jednu celu particiju (dakle, niz promenljive dužine – `std::vector<unsigned int>` – u kodu definisan kao `part_v`) a ne vraća nikakav rezultat.

Za funkciju `visit` se uzima funkcija `print_partition` definisana u kodu, koja ispisuje particiju u standardnom obliku na standardni izlaz. Ovo je dovoljno za proveru ispravnosti algoritama za nabrajanje navedenih u radu.

Konačno, slede implementacije algoritama iz prethodnog poglavlja rada.

5.1 Implementacija algoritama za nabrajanje particija

Na početku je potrebno definisati sve navedene tipove – za predstavljanje jedne particije prirodnog broja, funkcije `visit` i sl, te uključiti potrebna zaglavlja iz C++ standardne biblioteke. Korišćena zaglavlja su: `vector` (za tip particije), `functional` (za tip visit funkcije), `string` (za funkciju `std::stoi` – konverzija teksta, unešenog u standardni ulaz ili datog kao argument komandne linije, u broj), `iomanip` i `iostream` (za formatirani ispis na standardni izlaz). Definisani su tip particije `part_v`, tip funkcije `part_visitor` i funkcija `print_partition` (slovo ‚v' u `part_v` je od *vector*):

```
typedef std::vector<unsigned int> part_v;
typedef std::function<void(const part_v &)> partv_visitor;
```

Sada sledi implementacija jednog po jednog algoritma. Dosta algoritama (posebno rekurzivnih) implementirano je kao dve funkcije – jedna koja sadrži sve potrebne dodatne ulazne promenljive, i druga koja poziva prvu sa odgovarajućim ulaznim podacima, a prihvata samo n.

Algoritam partitions_rec

Ulazne promenljive ovog algoritma su, pored broja n, i niz prethodnih sabiraka – odnosno deo particije na koji se „dopisuje" svaka generisana particija manjeg broja (novog n), kako bi mogla da se „sastavi" cela particija početnog broja n.

U ovom slučaju je najefikasnije koristiti referencu na particiju – particija je svakako predstavljena kao niz, i nije neophodno da bude kompletna – uostalom, broj n nije deo definicije particije, već ulazna promenljiva algoritma koji treba da ih sastavi. Dakle, na već postojeću, nedovršenu particiju, biće dodat sledeći sabirak, i referenca do te iste particije prosleđena pri svakom sledećem (rekurzivnom) pozivu funkcije. Na kraju se taj dodati sabirak uklanja sa kraja particije, kako bi se ista vratila u stanje u kome je bila na početku funkcije. Ovde se sa particijom postupa kao sa stekom.

Treba još napomenuti da na ovaj način nije potrebno navoditi m kao broj sabiraka u P. Klasa `std::vector` koja se koristi za tip particije već u sebi čuva broj elemenata – a pritom ima i metod za dodavanje elementa *na kraj*, čitanje i uklanjanje *poslednjeg* elementa. Imajući sve to u vidu, sledi kod:

```
void partitions_rec (
    const partv_visitor & visit, unsigned int n, part_v & P)
{ // n, m, P
    if (n == 0) {
        visit(P);
        return;
    }
    unsigned int limit = P.empty() // m = 0
        ? n
        : P.back();                // P_m
    if (limit > n) limit = n;      // min(limit, n)
    for (unsigned int x = 1; x <= limit; x++) {
        P.push_back(x); // P_{m+1} ← x
        partitions_rec(visit, n - x, P); // n - x, m + 1, P
        P.pop_back(); // vracamo poslednje stanje
    }
}
```

U komentarima su navedene veze datog koda sa pseudokodom istog algoritma. Glavna funkcija je `partitions_rec(visit, n)`, koja rekurzivno poziva svoju pomoćnu funkciju sa dodatnim argumentom P – inicijalno praznom particijom. Ovo odgovara pozivu *partitions_rec*$(n, 0, ())$ iz pseudokoda.

Algoritam partitions_ind

Bukvalna implementacija algoritma *partitions_ind* po pseudokodu podrazumeva rekurzivni poziv čitavog algoritma za $n - 1$, a zatim sastavljanje jedne po jedne particije

broja n prolazom kroz niz particija broja $n-1$. Ovakvu implementaciju po principu matematičke indukcije je praktično trivijalno prevesti u iterativni proces.

```
std::vector<part_v> partitions_ind (unsigned int n) {
    std::vector<part_v> P;
    if (n == 0) {
        P.push_back(part_v()); // skup od samo prazne
           particije
        return P;
    }

    const std::vector<part_v> & Prev = partitions_ind(n-1);
    P.reserve(Prev.size());    // alocirati dovoljno
       prostora unapred
    for (auto b = Prev.begin(), e = Prev.end(); b != e; b++)
       {
        // *b = (p_1, p_2, ..., p_m) - jedna particija broja n-1
        unsigned m = b->size();
        part_v P1(*b);   // kopija
        P1.push_back(1); // (p_1, p_2, ..., p_m, 1)
        P.push_back(P1);

        if ((m > 0) && ((m < 2) || ((*b)[m-2] > (*b)[m-1])))
           {
            part_v P2(*b);
            P2[m-1]++;   // (p_1, p_2, ..., p_{m-1}, p_m + 1)
            P.push_back(P2);
        }
    }
    return P;
}
```

Primetimo da su indeksi elemenata u particiji `*b` (trenutno razmatrana particija iz skupa `Prev`) i `P2` (nova generisana particija koja se dodaje u izlazni skup P) umanjeni za 1 – to je zato što u programskom jeziku C++ indeksi u nizu počinju od 0.

Međutim, nije potrebno imati ceo skup particija broja $n-1$ da bi se počeo sastavljati skup particija broja n. Nema potrebe čuvati ceo skup particija broja n – štaviše, za veliko n to može biti i nepraktično (prevelika potrošnja memorije). Kao što je pomenuto, zadatak ovih algoritama je da obrade svaku particiju broja n, *nezavisno*, na bilo koji zadati način – u ovom slučaju, ispisom na ekran. Za takvu obradu nije potrebno čuvati niti vraćati ceo skup particija.

Način rada ovog algoritma je da za svaku particiju broja $n-1$ sastavi jednu ili dve moguće particije broja n, na načine opisane u ideji algoritma u prethodnom poglavlju. Za to je dovoljno „zatražiti" jednu po jednu particiju broja $n-1$, zatim je zapamtiti i iskoristiti za sledeću jednu ili dve particije broja n koje se od algoritma „zatraže" prilikom generisanja particija broja $n+1$.

Ovakav način rada predstavlja koncept enumeratora: enumerator pamti trenutno stanje (broj n i poslednju generisanu particiju broja n) i sadrži metod koji vraća sledeći element na osnovu trenutnog stanja. Još je potreban način provere da li je enumerator

„iscrpljen” - odnosno da li su generisane sve particije broja n.

Korišćenje enumeratora je mnogo memorijski efikasnije, i za razliku od prethodno navedene naivne implementacije, neophodna je rekurzija – za svaki broj k od 1 do n je potrebno čuvati stanje enumeratora za particije broja k, i za svaku (ili svaku drugu) particiju broja k je potrebno zatražiti sledeću particiju broja $k-1$, što predstavlja rekurzivni poziv funkcije enumeratora. Memorijska složenost ovakve implementacije je $O(n)$ particija, odnosno maksimalno $O(n^2)$ brojeva.

Enumerator, s obzirom na to da ima svoje stanje i da je potrebno čuvati do n enumeratora, svaki sa zasebnim stanjem, u memoriji, ne može da bude prosta funkcija – potrebno je da ima uvid u trenutno stanje, koje se menja prilikom svakog poziva funkcije. U C++ i drugim objektno-orijentisanim programskim jezicima enumerator može da bude predstavljen kao objekat, odnosno klasa koja će se u kodu instancirati n puta, te toj klasi dodeljen metod koji vraća jednu po jednu – sledeću – particiju ili signalizira da su sve particije nabrojane.

Sledi referentna implementacija klase enumeratora particija:

```
class part_ind_enum {
private:
    part_v last;
    part_ind_enum * parent;
    int state; // broj dobijenih particija od jedne manje
       (n-1)

public:
    const unsigned int n;
    part_ind_enum (unsigned int n):
        last(), parent(nullptr), state(0), n(n)
    {}
    ~part_ind_enum () {
        if (parent != nullptr)
            delete parent;
    }

    bool next (part_v & p) {
        if (n == 0) {
            // specijalan slucaj - jedna prazna particija
            if (this->state == 0)
                p.resize(0);
            return (this->state++ == 0);
            // state == 1 => vec smo vratili particiju
        }

        unsigned m;
        switch (this->state) {

        // nismo dobili novu particiju od particije broja
           n - 1
        case 0:
```

```
            if (parent == nullptr)
                // inicijalna particija - instancirati
                   enumerator
                parent = new part_ind_enum(n-1);
            // trazimo sledecu particiju broja n-1
            // neka je to (p_1, p_2, ..., p_m)
            if (!parent->next(last)) {
                // nema vise particija broja n-1, pa time
                   ni n
                this->state = 2;
                // delete parent; parent = nullptr;
                return false;
            }
            m = last.size();
            p = last; // "kopiramo" ceo niz
            p.push_back(1); // (p_1, p_2, ..., p_m, 1)
            if ((m > 0) && ((m < 2) || (last[m-2] >
               last[m-1]))) {
                // moze se dobiti jos jedna nova particija
                   od iste
                this->state = 1;
            }
            // ako ne moze, stanje ostaje 0 i u sledecem
            // pozivu se trazi nova particija broja n-1
            return true;

        // dobili smo jednu novu particiju i mozemo jos jednu
        case 1:
            m = last.size();
            p = last; // prepisujemo particiju
            p[m-1]++; // (p_1, p_2, ..., p_{m-1}, p_m + 1)
            this->state = 0;
            // sada smo zavrsili sa particijom broja n-1
            return true;

        // vec smo prosli sve particije broja n - nema dalje
        default:
            return false;
        }
    }
};
```

Stanje enumeratora se sastoji od broja n, poslednje dobijene particije broja $n-1$ (ne n), kao i broj već vraćenih particija broja n u odnosu na poslednju particiju broja $n-1$. Poslednja promenljiva stanja služi radi određivanja načina na koji treba sastaviti particiju broja n – kao što je spomenuto u opisu algoritma, od nekih particija broja $n-1$ moguće je dobiti dve nove particije broja n.

Glavna funkcija je jednostavna – instancirati enumerator particija broja n, i dok god

nam vraća novu particiju, obraditi je (ispisati na ekran):

```
void partitions_ind2 (part_visitor visit, unsigned int n) {
    part_ind_enum e(n);
    part_v p;
    while (e.next(p))
        visit(p);
}
```

Algoritam revlex_partitions

Ovaj iterativni algoritam ne zahteva pomoćne funkcije i dodatne argumente – poziva se sa `revlex_partitions(visit, n)` i u jednoj petlji generiše sve particije broja n u obrnutom leksikografskom poretku.

Analizom algoritma u prethodnom poglavlju ustanovljeno je da se particije u kojima je jedan ili više delova jednako 2 mogu obraditi na efikasniji način – naime, poslednja dvojka se smanjuje na 1 i dopisuje se jedinica na kraj particije. Ostale particije su oblika $(\ldots, x, 1, 1, \ldots, 1)$ gde je $x > 2$ i prva leksikografski manja particija je oblika $(\ldots, x-1, x-1, \ldots, r)$ za $r < x$. Ovo zahteva posebnu obradu, koja se dešava u unutrašnjoj petlji pri dnu koda.

Imajući u vidu pomenutu optimizaciju za slučaj $x = 2$, kod može biti napisan ovako:

```
void revlex_partitions (part_visitor visit, unsigned int n) {
    part_v a = {n};
    unsigned m = 1;
    do {
        visit(a);
        unsigned q = m;
        while (q && (a[--q] == 1))
            ;
        // q je indeks poslednjeg elementa veceg od 1
        if ((q == 0) && (a[q] == 1))
            // kriterijum zaustavljanja - moguc jedino za
               n = 1
            return;
        // korak 2 -> 1, 1
        for (; a[q] == 2; q--) {
            a[q] = 1;
            m++;
            a.push_back(1);
            visit(a);
            if (q == 0)
                // kriterijum zaustavljanja - (1, 1, ..., 1)
                return;
        }
        a[q]--;
        unsigned r = m - q;
        m = q + 1;
        // korak x, 1, ..., 1 -> x - 1, ..., x - 1, r
```

```
        while (r > a[q]) {
            a[m++] = a[q];
            r -= a[q];
        }
        a.resize(m + 1);
        a[m++] = r;
    } while (true);
}
```

Kao i u prethodnom algoritmu, indeksi elemenata u nizu `a` su za jedan manji nego u pseudokodu, s obzirom na indeksiranje nizova počevši od 0 umesto 1. `for` petlja u sredini koda obrađuje particije sa sabirkom 2 u sebi. Ostatak koda odgovara pseudokodu algoritma *revlex_partitions*.

Particije sa tačno m delova – algoritam colex_partitions

Kao i `revlex_partitions`, ovaj algoritam je iterativni i nisu potrebne nikakve dodatne funkcije ili ulazne vrednosti sem m i n – u glavnoj funkciji je ceo algoritam. Sem nekih vrednosti koje predstavljaju indekse (pa su time za jedan manje u C++ kodu) kod u potpunosti odgovara pseudokodu algoritma:

```
void colex_partitions (partv_visitor visit, unsigned int n,
   unsigned int m) {
    part_v a(m, 1); // m jedinica; sve particije imaju tacno
       m delova
    a[0] = n - m + 1; // pocetna particija (n-m+1,1,...,1)
    do {
        visit(a);
        // specijalan slucaj m=1 - jedina particija je (n)
        if (m == 1)
            return;
        // prepravljanje a_1 i a_2
        while (a[1] < a[0] - 1) {
            a[0]--;
            a[1]++;
            visit(a);
        }
        // za m = 2 petlja iznad daje sve trazene particije
        if (m == 2)
            return;
        // obrada od treceg dela na dalje
        unsigned int j = 2;
        unsigned int s = a[0] + a[1] - 1;
        unsigned int x;
        // nalazenje j
        while (a[j] >= a[0] - 1) {
            s += a[j];
            j++;
            if (j == m) // a_m >= a_1 - 1 => kraj algoritma
```

```
                return;
            } // U ovom trenutku je s = a_1 + ... + a_{j-1} - 1
            // povecavanje a[j]
            x = a[j] + 1;
            a[j] = x;
            j--;
            // prepravljanje do a[j]
            while (j > 0) {
                a[j] = x;
                s -= x;
                j--;
            }
            a[0] = s;
        } while (true);
}
```

Svi posebni slučajevi i koraci iz pseudokoda su obeleženi odgovarajućim komentarima u kodu.

Program „particije"

Zajedno sa potrebnim C++ zaglavljima, definicijama tipova i funkcijom za ispis sa početka ovog podnaslova, to čini kompletan kod programa za nabrajanje particija prirodnog broja.

Program za nabrajanje particija sadrži navedene implementacije svih algoritama, te prihvata, kao argumente komandne linije, redni broj algoritma i broj n, a za algoritme gde je to potrebno (`colex_partitions`) i njihove dodatne argumente (broj m). U prilogu `nabrajanje.cpp` je ceo C++ kod programa, uključujući `main` funkciju.

Za potrebe merenja vremena izvršavanja algoritma za nabrajanje particija, može se pročitati sistemsko vreme odmah pre i nakon `switch` bloka koji, u odnosu na sve unete argumente, poziva odgovarajuću funkciju – algoritam. Za potrebe testiranja efikasnosti, takođe, funkcija za ispis particija `PRINT` se može zameniti praznom funkcijom `NO_OP` (definisanom u `main`), kako se ne bi računalo dodatno vreme potrebno operativnom sistemu za ispis podataka na ekran ili u fajl, već da u vreme izvršavanja ulazi samo proces dobijanja particija.

U slučaju računanja vremena bez ispisa particija, potrebno je izbeći optimizacije kompilatora koje bi mogle da izbace neka izračunavanja čiji se rezultati, usled `NO_OP` funkcije, ne koriste. Prvi način bi bio isključiti sve optimizacije, pomoću `gcc` opcije `-O0`; ovo se ispostavilo kao veoma neefikasno rešenje. Moderni kompilatori umeju da ubrzaju izvršavanje (korisnog) koda i po nekoliko puta, što i jeste bio slučaj za program `particije` – dobijen je nekoliko puta sporiji kod u odnosu na korišćenje `-O2`. Međutim, najvećim delom je upravo usporen „koristan" kod – poređenjem vremena izvršavanja svih algoritama bez ispisa i optimizacija, sa vremenom izvršavanja istih algoritama *sa* ispisom (`PRINT` funkcija umesto `NO_OP`) i optimizacijama (`-O2`), pokazalo se da je razlika vremena izvršavanja dva algoritma, za istu ulaznu vrednost n, nekoliko puta veća u programu bez optimizacija.

Zbog toga je primenjen drugi način: funkcija `NO_OP` je naterana da ipak na neki način „obradi" particiju, a da to ne bude ispis na ekran. Obrada u novoj funkciji se sastoji od prolaska kroz sve delove particije i ažuriranja jedne globalne promenljive na način koji

zavisi od njihovih vrednosti; vrednost te promenljive se na kraju izvršavanja programa ispisuje na ekran. Ovaj način se ispostavlja kao ispravan, i uz `-O2` opciju optimizacije, primećuje se malo duže vreme izvršavanja usled ove jednostavne obrade. Poređenje vremena izvršavanja nove verzije bez ispisa i verzije sa ispisom, kad se uzme u obzir da bi vreme samog ispisa trebalo biti isto u svim algoritmima, ovaj put ima smisla.

Definicija makroa `TIMING` u fazi kompilacije koda (opcija `-DTIMING`) aktivira funkciju `NO_OP`.

5.2 Implementacija algoritama za računanje broja particija

Za razliku od algoritama za nabrajanje particija, ovde je problem implementacije malo jednostavniji – na primer, ne razmatraju se načini predstavljanja jedne particije niti bilo kakve obrade dobijenih particija (funkcija „visit" iz prethodnog podnaslova). Međutim, potreban je rad sa velikim brojevima, s obzirom na veličinu rezultata; u slučaju nabrajanja particija nije računat i njihov broj, već su one samo ispisivane na ekran. Svako sabiranje velikih brojeva oduzima više vremena – to više za računar nije jedna operacija sabiranja brojeva, već $O(\sqrt{n})$ operacija.

U slučaju da nas zanima samo nekoliko poslednjih cifara broja $p(n)$, ili ostatak pri deljenju $p(n)$ nekim brojem, rezultat može da bude predstavljen jednim običnim numeričkim tipom – 64-bitnim celim brojem. U ovom slučaju treba voditi računa kad je potrebno primeniti ostatak pri deljenju; svi algoritmi su, međutim, mnogo efikasniji jer je svako sabiranje, oduzimanje, poređenje, pa i računanje ostatka pri deljenju jedna prosta operacija. Ovaj način rada algoritma može da se iskoristi za analizu potrebnog broja sabiranja pri računanju $p(n)$.

U ovom radu je kod za računanje broja particija napisan i testiran na oba načina, i upoređeni su rezultati. Za delioca je izabran broj 10^{18} – dakle, računa se poslednjih 18 cifara $p(n)$ u bržem načinu rada.

Za rad sa velikim brojevima korišćen je C++ interfejs GMP biblioteke – klasa `mpz_class`, koja predstavlja cele brojeve. Rad sa ovom klasom je skoro trivijalan – definisani su svi standardni matematički operatori (od čega su implementiranim algoritmima potrebni samo poređenje, sabiranje i – za algoritam `p_pent` – oduzimanje), kao i operacije unosa i ispisa. Time je objekte tipa `mpz_class` moguće koristiti na potpuno isti način kao standardne, primitivne celobrojne tipove u C++.

Zapravo, u slučaju rada sa velikim brojevima (umesto sa standardnim 64-bitnim celobrojnim tipom po modulu 10^{18}), dovoljno je samo tip `int` zameniti navedenim tipom `mpz_class` u kodu. To je i urađeno – za tip rezultata definisan je tip `result_t`, koji je u slučaju računanja poslednjih cifara broja $p(n)$ neoznačeni 64-bitni broj (`unsigned long long`), a inače tip klase velikog broja `mpz_class` iz GMP biblioteke.

Implementacija prvog algoritam za računanje $p(n)$ – naivnog rekurzivnog algoritma `p_rec` – skoro je ista kao njegov pseudokod:

```
result_t p_rec(unsigned int n, unsigned int max) {
    if (n == 0)
        return result_t(1);
    else if (max == 0)
        return result_t(0);
    else if (n < max)
        return p_rec(n, n); // n, max - 1
```

```
    return p_rec(n, max - 1) + p_rec(n - max, max);
}

result_t p_rec(unsigned int n) {
    return p_rec(n, n);
}
```

Ono što se u ovom kodu razlikuje u odnosu na pseudokod je uslov $n < max$: kao što je napomenuto na početku ovog poglavlja, pošto se u celom programu koriste neoznačeni celi brojevi, n ne može da bude negativno. Zbog toga je potrebno izbeći rekurzivni poziv za $n < 0$ – a to se postiže izostavljanjem poziva `p_rec(n - max, max)` za $n - max < 0$, čime je povratna vrednost samo `p_rec(n, max - 1)`.

Sledi kod algoritma `p_memo` – grananje sa odsecanjem kao osnovna varijanta algoritma dinamičkog programiranja (memoizacija):

```
result_t p_memo(unsigned int n, unsigned int max,
   vector<vector<result_t> > & T) {
    if (n == 0)
        return result_t(1);
    else if (max == 0)
        return result_t(0);
    if (T[n - 1][max - 1] == result_t(0)) {
        if (n < max)
            T[n - 1][max - 1] = p_memo(n, n, T); // n, max -
               1
        else
            T[n - 1][max - 1] = p_memo(n, max - 1, T)
                + p_memo(n - max, max, T);
    }
    return T[n - 1][max - 1];
}

result_t p_memo(unsigned int n) {
    vector<vector<result_t> > T(n, vector<result_t>(n,
       result_t(0)));
    result_t p = p_memo(n, n, T);
    return p;
}
```

Implementacija funkcije `p_memo(n,max,T)` odgovara pseudokodu, uz istu napomenu u vezi sa uslovom $n < max$ kao kod `p_rec`. S obzirom na to da se dimenzije tabele T znaju $(n \times n)$, ona se alocira na početku koda, u „glavnoj" funkciji `p_memo(n)`, te prosleđuje rekurzivnoj funkciji `p_memo(n,max,T)`, koja je popunjava. Tabela T se kao referenca prenosi kroz sve rekurzivne pozive funkcije `p_memo`, tako da se svaki put čita i menja ista tabela, isti podaci u memoriji – to je pomenuta „globalno dostupna tabela" iz opisa algoritma `p_memo`.

Na sličan način, po pseudokodu algoritma `p_dp`, implementira se i iterativna verzija DP algoritma za $p(n)$:

```
result_t p_dp(unsigned int n, unsigned int max) {
```

```
    vector<vector<result_t> > T(n + 1,
        vector<result_t>(n + 1, result_t(0)));

    for (unsigned i = 0; i <= n; i++)
        T[0][i] = result_t(1);

    for (unsigned i = 1; i <= n; i++)
        for (unsigned j = 1; j <= max; j++) {
            T[i][j] = T[i][j - 1];
            if (i >= j)
                T[i][j] += T[i - j][j];
        }

    return T[n][max];
}

result_t p_dp(unsigned int n) {
    return p_dp(n, n);
}
```

Glavna funkcija ovaj put ne vrši nikakvu pripremu, niti se tabela prosleđuje funkciji `p_dp(n,max)` – ona samo poziva funkciju sa argumentom $max = n$, koja priprema tabelu. Tabela je, takođe, proširena – u slučaju $i - j = 0$ iz pseudokoda (uslov koji se proverava je $i - j < 0$, a ne $i - j \leq 0$) uzima se vrednost $T_{0,j}$, te je potrebno dodati „nultu" vrstu matrici. S obzirom na dosadašnje algoritme (uključujući i rekurzivnu verziju ovog algoritma – `p_memo`), jasno je da je $T_{0,j} = 1$ za svako $j > 0$.

Konačno, implementirana je funkcija za računanje $p(n)$ pomoću rekurentne formule date u naslovu 3.1 – `p_pent`:

```
result_t p_pent(unsigned int n) {
    vector<result_t> T(n + 1, result_t(0));
    T[0] = result_t(1);

    for (unsigned i = 1; i <= n; i++) {
        int k = 1;
        int nk = i - 1;
        char s = 1; // (-1)^{k-1}
        // p(i) = p(i-1) + p(i-2) - p(i-5) - p(i-7) + p(i-12) + ...
        while (nk >= 0) {
            if (s) T[i] += T[nk]; // p(i - g_k)
            else   T[i] -= T[nk];

            nk -= k;
            if (nk < 0) // zaustavljamo dalji racun; p(n) = 0
               za n < 0
                break;

            if (s) T[i] += T[nk]; // p(i - g_{-k})
            else   T[i] -= T[nk];
```

```
            nk -= (k << 1) | 1; // drugi nacin zapisa 2k+1
            k++;
            s ^= 1;
        }
    }

    return T[n];
}
```

Funkcija u iteracijama računa $p(i)$ za svako i redom od 1 do n; postavlja se pitanje načina izbora sabiraka koji ulaze u zbir $p(i)$.

Petougaoni brojevi su 1, 2, 5, 7, 12... U opštem slučaju, petougaoni broj sa indeksom k (za ceo broj k – pozitivan ili negativan) je $(3k^2 - k)/2$. Razlika petougaonih brojeva sa indeksima k i $-k$ je tačno k. Razlika k-tog i $k+1$-vog petougaonog broja se može izračunati na sledeći način:

$$\frac{3(k+1)^2 - k - 1}{2} - \frac{3k^2 - k}{2} = \frac{3(2k+1) - 1}{2} = 3k + 1$$

Dakle, počevši od $n_k = i-1$ i $k = 1$, uzastopnim smanjivanjem n_k prvo za k a zatim za još $2k+1$ dobijaju se svi petougaoni brojevi, sa alternirajućim indeksima (1, -1, 2, -2, itd). Za svaki dobijeni broj n_k, dok god je $n_k \geq 0$, sabrati rezultat sa $(-1)^{k-1}p(n_k)$; promenljiva `s` u kodu se menja sa svakim uvećanjem k čime određuje znak činioca $(-1)^{k-1}$.

Program „brpart"

Program za računanje $p(n)$, dat u prilogu `brpart.cpp`, sadrži navedene implementacije svih algoritama, te prihvata, na isti način kao i program „particije", redni broj algoritma i broj n kao argumente komandne linije.

Isto kao u programu „particije", i ovde se može dodati uvid u sistemsko vreme odmah iznad i ispod `switch` bloka, te njihova razlika uzeti za vreme izvršavanja algoritma.

Za potrebe sabiranja velikih brojeva i dobijanja *tačnih* vrednosti $p(n)$, korišćen je tip `mpz_class` iz GMP biblioteke.

5.3 Rezultati izvršavanja

Program „brpart", sa navedenim implementacijama algoritama za izračunavanje $p(n)$, pokrenut je sa različitim vrednostima broja n kao ulaznim parametrima i svakim algoritmom po jednom. Referentna konfiguracija na kojoj su dobijeni rezultati izvršavanja je:

- Intel Core i7 procesor (brzine 2,5 GHz)
- 64-bitni Linux operativni sistem (kernel 5.x)
- GNU GCC / G++ 10.1

Implementacije algoritama nisu paralelizovane, odnosno pisane za rad sa više niti, te se koristi samo jedan od 4 logička procesora.

Vrednosti broja n su izabrane tako da pokažu povećanje vremena izvršavanja za sva 4 implementirana algoritma. Uzeto je vremensko ograničenje od 20 sekundi za svaki algoritam. Slede prvo vremena izvršavanja algoritama za `MODULO =` 10^{18} i dobijeni odgovori u tabeli 2.

n	p_rec	p_memo	p_dp	p_pent	$p(n)(mod 10^{18})$
50	0.017808	0.000049	0.000043	0.000022	204226
60	0.047895	0.000060	0.000057	0.000021	966467
70	0.182087	0.000064	0.000070	0.000027	4087968
80	0.752749	0.000077	0.000078	0.000026	15796476
90	2.867295	0.000093	0.000097	0.000027	56634173
100	10.188389	0.000111	0.000118	0.000025	190569292
500	–	0.001964	0.002399	0.000080	165032574323995027
1000	–	0.008496	0.010708	0.000187	622473692149727991
2000	–	0.035548	0.047305	0.000495	799959512200344166
3000	–	0.082746	0.108373	0.000920	462742231683423624
4000	–	0.153005	0.197423	0.001378	150910234889093895
5000	–	0.250842	0.309941	0.001910	203824652329144349
6000	–	0.357753	0.448999	0.002499	925072405349246129
7000	–	0.490328	0.615618	0.003141	394374450791229199
8000	–	0.653668	0.819246	0.003846	450990388402844164
9000	–	0.835475	1.061314	0.004581	790335590435626459
10000	–	1.044905	1.343772	0.005367	556052906916435144
12000	–	1.543749	2.034761	0.007005	251809128853180610
15000	–	2.474299	3.360063	0.009835	246029351672130266
20000	–	4.839875	6.669134	0.015048	166735959242113097
40000	–	**MEM**	**MEM**	0.042918	082673857997469797
60000	–	–	–	0.078682	234313575586067751
80000	–	–	–	0.120999	816485579800085553
100000	–	–	–	0.168909	158600569421098519
200000	–	–	–	0.477701	417376966116043766
300000	–	–	–	0.878207	825597038266403824
400000	–	–	–	1.359575	204985841890352489

Tabela 2: Rezultati izvršavanja `brpart_mod`

Pošto se ovde radi sa ,,običnim" brojevima (numeričkim tipovima u C), za koje postoje jednostavne procesorske instrukcije, vreme izvršavanja u tabeli 2 je proporcionalno broju *sabiranja* potrebnih za računanje $p(n)$. Dakle, za algoritam *p_pent* na primer, kao što je navedeno u analizi složenosti algoritma u naslovu 4.4, vremenska složenost je $O(n\sqrt{n})$. Za veće vrednosti n upravo se može primetiti rast vremena izvršavanja u skladu sa navedenom složenošću.

Takođe, u skladu sa napomenom memorijske složenosti algoritama *p_memo* i *p_dp*, može se primetiti da za $n = 40000$ program ostaje bez dovoljno memorije za dalji rad (to je označeno sa `MEM` u tabeli). Za razliku od vremenskog, nije dato nikakvo memorijsko ograničenje za rad programa; prosečna raspoloživa memorija na sistemu na kome je pokretan program je oko 6 GB.

U tabeli 3 slede rezultati izvršavanja istih algoritama, za iste vrednosti n, ali sa sabi-

ranjem velikih brojeva.

n	p_rec	p_memo	p_dp	p_pent	$\lfloor \log_{10} p(n) \rfloor + 1$
50	0.055658	0.000294	0.000346	0.000055	6
60	0.282887	0.000402	0.000505	0.000050	6
70	1.277305	0.000526	0.000781	0.000054	7
80	5.223301	0.000658	0.001004	0.000057	8
90	19.711351	0.000815	0.001254	0.000056	8
100	–	0.001019	0.001603	0.000062	9
500	–	0.024612	0.033189	0.000213	22
1000	–	0.100399	0.123831	0.000505	32
2000	–	0.422182	0.562729	0.001362	46
3000	–	1.041432	1.404720	0.002470	57
4000	–	2.027429	2.673574	0.003842	67
5000	–	3.200835	4.309158	0.005268	75
6000	–	4.815521	6.535038	0.006955	82
7000	–	6.751607	9.479108	0.008852	89
8000	–	9.453041	13.199062	0.010870	95
9000	–	12.614708	18.729710	0.013135	101
10000	–	16.062296	**MEM**	0.015536	107
12000	–	**MEM**	–	0.020822	118
15000	–	–	–	0.029714	132
20000	–	–	–	0.047572	153
40000	–	–	–	0.160320	218
60000	–	–	–	0.372817	268
80000	–	–	–	0.650407	310
100000	–	–	–	1.001912	347
200000	–	–	–	3.594049	493
300000	–	–	–	8.089843	604
400000	–	–	–	14.642015	699

Tabela 3: Rezultati izvršavanja `brpart` – poslednja kolona predstavlja broj cifara $p(n)$

S obzirom na rad sa velikim brojevima u ovom slučaju, za svako sabiranje dva broja slične veličine (neka je to k cifara) potrebno je $O(k)$ operacija. Time je složenost algoritama sada veća nego u prethodnom slučaju, sa računanjem poslednjih 18 cifara, i kao i tada, i sada se za veće vrednosti n (od 100 pa nadalje) može primetiti adekvatan rast vremena izvršavanja sa povećanjem n – npr za *p_pent* algoritam se to slaže sa $O(n^2)$. Iz istog razloga je i zauzeće memorije veće, jer međurezultati (tabela `T` u *p_memo* i *p_dp* i niz u *p_pent*) više nisu konstantne veličine; algoritam *p_memo* ostaje bez raspoložive memorije već za $n = 12000$, a *p_dp* već za $n = 10000$.

Na kraju, u tabeli 4 dati su rezultati izvršavanja programa za nabrajanje svih particija broja n, za nekoliko različitih vrednosti n. Primetimo da za $n = 80$ najsporiji algoritam, `partitions_ind`, za samo malo premašuje vremensko ograničenje.

Efikasnost svih razmatranih algoritama za nabrajanje bi trebalo da bude približno ista, s obzirom na istu vremensku složenost nabrajanja particija. Ono što doprinosi razlici u efikasnosti algoritama je prosečno vreme generisanja jedne particije.

Na primer, može se primetiti da je algoritam `revlex_partitions` dosta efikasniji od

n	partitions_rec	partitions_ind	partitions_ind2	revlex_partitions
50	0.011344	0.193040	0.021847	0.003661
60	0.061235	1.026248	0.108868	0.021297
70	0.253802	4.648430	0.483252	0.090144
80	1.059915	**20.852948**	1.996243	0.363885
90	4.026481	–	7.658187	1.372239
100	14.348034	–	–	4.874400

Tabela 4: Rezultati izvršavanja programa `particije`

ostalih, što se može očekivati s obzirom na to da je algoritam potpuno iterativan i da je pritom optimizovan slučaj $x = 2$, koji se dešava u većini particija. Ovo doprinosi veoma dobrom prosečnom vremenu generisanja jedne particije.

Sa druge strane, algoritam `partitions_ind`, iako dobar primer za jedan od načina nabrajanja particija, pokazuje se kao prilično neefikasan, čemu doprinosi više faktora. Ovaj algoritam je takođe memorijski najzahtevniji – čak i u svojoj optimalnoj verziji zahteva više memorije od ostalih algoritama razmatranih u ovom radu. Rekurzija kroz enumeratore, dubine do n, čini prosečno vreme generisanja svake particije dužim, s obzirom na to da svaki rekurzivni poziv podrazumeva operacije sa stekom. Osnovna verzija algoritma `partitions_ind` međutim, iako praktično iterativna, još je manje efikasna zbog eksponencijalne količine memorije sa kojom radi. Pošto se sve particije broja $n - 1$ generišu odjednom, pre prelaska na generisanje particija broja n, nije moguća izmena „trenutne" particije sa kojom se radi (kao u slučaju optimizovane verzije sa enumeratorima), već se svaka particija dobija kopiranjem prethodne, uz povremenu potrebu alokacije memorije. Alokacija i rad sa većim količinama memorije loše utiču na vreme izvršavanja.

6 Zaključak

Particija prirodnog broja je njegovo predstavljanje u obliku zbira nekoliko manjih brojeva. Particije prirodnih brojeva, kao i particije skupova, neke su od klasa iz poznate dvanaestorke kombinatornih problema – raspoređivanja kuglica u kutije. Particije brojeva imaju veliki značaj u kombinatorici kao i u teoriji brojeva. Najznačajnija dostignuća u ovoj oblasti postigli su Ojler u XVIII veku, kao i Ramanudžan, Hardi i posle njih Rademaher. Njihova dostignuća su omogućila efikasno računanje ukupnog broja particija datog prirodnog broja n, $p(n)$, što je i pokazano u radu implementacijom različitih algoritama za računanje $p(n)$ i poređenjem vremena izvršavanja.

Osim računanja broja particija, istraženi su i različiti algoritmi za nabrajanje svih particija i njihove prednosti i mane. Značaj algoritama za nabrajanje je mogućnost rešavanja različitih problema iz teorije particija – na primer, služeći se pseudokodovima i idejama nekih algoritama navedenih u radu, mogu se rešiti problemi poput nalaženja „sledeće" particije date particije, slučajno odabrane particije, particije sa određenim svojstvima, a u kombinaciji sa algoritmima za računanje $p(n)$ i problemi poput nalaženja particije sa određenim „indeksom" (u smislu leksikografskog ili nekog drugog sortiranja particija).

Literatura

[1] Donald E. Knuth: *The Art of Computer Programming*, vol. 4a. Addison-Wesley, 2011

[2] George E. Andrews: *The Theory of Partitions*. Encyclopedia of Mathematics and Its Applications, vol. 2, Addison-Wesley, 1976

[3] Scott Ahlgren, Ken Ono: *Addition and Counting: the arithmetic of partitions*. Notices of the American Mathematical Society, vol. 48/9, 2001: 978-984

[4] Herbert S. Wilf: *Lectures on Integer Partitions*, Uni. Of Pennsylvania

[5] Johansson, F. (2012). *,,Efficient implementation of the Hardy-Ramanujan-Rademacher formula"*. LMS Journal of Computation and Mathematics. 15: 341–59.

[6] The GNU Multiple Precision Arithmetic Library. `https://gmplib.org/`