SVEUČILIŠTE U ZAGREBU
PMF – MATEMATIČKI ODJEL

Saša Singer

Složenost algoritama

Predavanja i vježbe

Zagreb, 2005.

Sadržaj

1. Uvod u složenost algoritama

Sadržaj ovog kolegija su algoritmi i njihova složenost (engl. "complexity of algorithms"). Ova dva pojma mogu se precizno definirati u okviru teorije izračunavanja (dio matematičke logike). Dio puta prema preciznoj formulaciji ovih pojmova napravit ćemo na kraju kolegija, kad budemo govorili o NP–potpunosti.

Za početak, zadovoljimo se neformalnim opisom ovih pojmova.

Algoritam = metoda, proces, postupak, tehnika ili pravilo za rješavanje neke klase problema (obično, računalom).

Složenost algoritma = cijena korištenja (izvođenja) tog algoritma za rješavanje jednog od tih problema iz te klase. Obično se mjeri u vremenu izvršavanja, potrebnoj memoriji ili sličnim relevantnim pojmovima (na pr. broj procesora za multiprocesorska računala).

Naš cilj je mjerenje i analiza složenosti algoritama. Zbog toga, oba ova pojma moramo malo detaljnije opisati. Prvo, da vidimo što smatramo algoritmom, a što ne, tj. kad ima smisla govoriti o složenosti. Slijedeći korak je preciznija formulacija pojma složenosti, u obliku korisnom za praktičnu primjenu.

1.1. Što je algoritam?

Algoritam, kao postupak za dobivanje rješenja, sastoji se od **konačnog niza koraka** — tzv. instrukcija, naredbi ili operacija, koje treba **izvršiti** (izvesti) da bi se dobilo rješenje.

Klasični primjeri algoritma su: Euklidov algoritam za nalaženje najveće zajedničke mjere dva prirodna broja (kao prvi formalni algoritam u povijesti), kulinarski recepti za pripremanje jela i pića (kokteli), itd.

Svaki algoritam ima slijedećih pet bitnih osnovnih svojstava (v. [10]):

- ulaz,
- izlaz,

- konačnost,
- definiranost i nedvosmislenost (određenost),
- efikasnost (efektivnost).

Svako od ovih svojstava ima direktne posljedice za formulaciju pojma složenosti algoritma.

1.1.1. Ulaz

Svaki algoritam ima nula ili više, ali konačno mnogo, vrijednosti koje treba zadati na početku, prije nego što počne izvršavanje algoritma. Ulazne vrijednosti definiraju konkretan primjerak — **zadaću** (engl. "instance") problema kojeg treba riješiti.

Svaka ulazna vrijednost uzima se iz nekog unaprijed zadanog skupa dozvoljenih objekata. Kartezijev produkt tih skupova je **domena ili područje definicije** problema. Ako algoritam ima $n > 0$ ulaznih vrijednosti $i_1, \ldots, i_n$, čije dozvoljene vrijednosti su iz skupova $I_1, \ldots, I_n$, onda je

$$I := I_1 \times I_2 \times \cdots \times I_n$$

domena ili područje definicije problema. Konkretna zadaća je $i = (i_1, \ldots, i_n) \in I$ i to je ulaz za svako pojedino izvršenje algoritma.

Na primjer, Euklidov algoritam ima dva ulaza, koji su oba elementi skupa $\mathbb{N}$ prirodnih brojeva.

Algoritmi bez ulaza nisu besmisleni (ni tako rijetki). To je naprosto postupak za rješenje konkretnog problema, tj. jedne fiksne zadaće (problem je jednočlan). Na primjer, u teoriji brojeva ima dosta takvih algoritama, koji provjeravaju određena fiksna svojstva fiksnih (velikih) brojeva.

Za analizu složenosti, zanimljivi su algoritmi koji, barem u principu, imaju beskonačan dozvoljeni skup svih mogućih ulaznih vrijednosti.

1.1.2. Izlaz

1.1.3. Konačnost

1.1.4. Definiranost i nedvosmislenost (određenost)

Izvođenje tog niza naredbi mora biti objektivan proces, koji se (u načelu) mora moći reproducirati. Sličan princip vrijedi, na primjer, za valjane eksperimente u fizici.

1.1.5. Efikasnost (efektivnost)

.

Probleme rješavamo s praktičnim ciljem da dobijemo (izračunamo) rješenje. To znači da izvršavanje algoritma ne smije trajati predugo, tj. moramo ga moći dočekati u razumnom vremenu.

Što je razumno vrijeme, naravno ovisi o problemu i potrebi (može par sekundi, ali i par mjeseci ili godina). Međutim, postoji velika klasa problema za koje je razumno rješenje moguće samo za **male** zadaće.

Primjer s permutacijama to lijepo pokazuje. Nije toliko bitno da li je $n \leq 10$ ili $n \leq 15$ razumna granica za n na ulazu, da bi rješenje bilo iole efikasno. Sasvim je očito da, bar u današnje vrijeme, rješenje za $n = 100$ nećemo dočekati ($100! \approx 10^{157}$), bez obzira na brzinu računala.

1.2. Što je složenost (efikasnost) algoritma?

1.3. Analiza složenosti algoritma

1.4. Asimptotsko ponašanje funkcija

1.5. Rekurentne (rekurzivne) relacije

Analiza složenosti algoritma vrlo često vodi na rekurzivne ili rekurentne relacije (engl. "recurrence relations").

Tipičan primjer su algoritmi konstruirani metodom "podijeli, pa vladaj". Rješenje zadaće x, veličine $|x|$, opisano je (ili dobiva se) preko rješenja nekih **manjih** zadaća istog tipa, iz **istog problema**. Realizacija vodi na rekurzivne algoritme, ili na algoritme s petljama koje realiziraju "rekurzivno" svođenje na manje zadaće.

Za složenost algoritma dobivamo rekurzivnu relaciju izraženu u veličini zadaće. Ta relacija može biti jednadžba, ako radimo preciznu analizu složenosti. No, često nas zanima samo red veličine složenosti, pa pojednostavljujemo neke elemente složenosti, što vodi na nejednadžbe. Obično trebamo gornju ogradu, pazeći da ona ne bude pregruba. Katkad, istovremeno radimo i ocjenu odozdo, s ciljem da preciznije opišemo ponašanje složenosti.

Zbog toga, prvo ćemo ponoviti osnovne činjenice o rješavanju nekih tipova rekurzivnih jednadžbi, a zatim ćemo pokazati neke načine (tehnike) za procjenu ili ocjenu (odozgo) ponašanja rješenja rekurzivnih nejednadžbi. Način izlaganja podešen je osnovnoj primjeni u analizi složenosti, bez namjere da precizno formuliramo sve rezultate.

Uočimo da postupak redukcije na manje zadaće mora biti konačan, da bi bio algoritamski. Drugim riječima, bar za zadani x, skup svih veličina zadaća koje se pojavljuju u izvršenju algoritma je **konačan**, tj. diskretan. Zbog toga promatramao diskretne ili diferencijske rekurzivne relacije.

Uvedimo osnovne oznake. Smatramo da su veličine zadaća $|x|$ prirodni brojevi. Oznaka je $|x| = n \in \mathbb{N}_0$. Ovaj model osigurava konačnost redukcije na manje zadaće, a dozvoljava i promatranje asimptotskog ponašanja složenosti za velike zadaće. Osim toga, ako veličine zadaća zaista mjerimo duljinom kôda (brojem bitova), dobivamo $|x| \in \mathbb{N}_0$.

Složenost koju promatramo je obično vremenska. Oznaka je

$$T(|x|) = T(n) \equiv t_n,$$

jer vrijednosti funkcije T na $\mathbb{N}_0$ možemo interpretirati i kao niz $t_n, n \in \mathbb{N}_0$.

Potpuno općenito, složenost zadaće veličine n ovisi o složenosti **svih** manjih zadaća, tj. oblik relacije je

$$T(n) \equiv t_n \left\{ \begin{matrix} = \\ \leq \end{matrix} \right\} f_n(t_{n-1}, \ldots, t_0), \quad n \in \mathbb{N}. \tag{1.5.1}$$

Funkcija f_n na desnoj strani mora ovisiti o n, zbog broja parametara (dimenzije domene).

Ako znamo sve funkcije $f_n, n \in \mathbb{N}$, uz pretpostavku da su korektno definirane, dovoljno je zadati **početnu vrijednost** t_0, da čitav niz t_n bude jednoznačno određen. Dakle, iz t_0 možemo (barem u principu) izračunati vrijednost t_n, za bilo koji $n \in \mathbb{N}$. Naravno, ako znamo računati vrijednosti svih funkcija f_i, za $i \leq n$.

Algoritamski gledano, to znači da svaku zadaću reduciramo sve dok ne dobijemo trivijalnu zadaću veličine nula. Nju znamo riješiti i to traje t_0 vremena (poznata konstanta). Funkcija f_n kaže koliko traje rješenje zadaće veličine n, u ovisnosti o trajanju rješenja svih manjih zadaća. Prirodno je očekivati da je $t_0 = 0$, ili barem "mala" konstanta.

Napomena 1.5.1. Iz prakse znamo da redukcija veličine zadaće **ne** ide uvijek do $n = 0$. Vrlo često, posebno u hibridnim algoritmima, redukcija ide sve dok ne dobijemo $n < k$, gdje je k unaprijed poznata "najmanja netrivijalna zadaća". Tj. k je konstanta (ne ovisi o n) u algoritmu. Naravno, veličina konstante k može ovisiti o implementaciji.

Smatramo da su sve manje zadaće ($n < k$) trivijalne, u smislu da ih znamo (eksplicitno) riješiti i da im znamo složenosti $t_0, \ldots, t_{k-1}$, kao poznate konstante. Drugim riječima, rekurzivna relacija (1.5.1) vrijedi za $n \geq k$, a za potpunu određenost niza t_n trebamo k početnih uvjeta $t_0, \ldots, t_{k-1}$. ■

Ovaj oblik rekurzivne relacije je preopćenit, da bismo išta konkretno mogli reći o ponašanju niza t_n. Potrebno je uvesti neke dodatne pretpostavke na funkcije f_n, za bilo kakav koristan zaključak. Osim toga, u analizi složenosti rijetko trebamo ovako općeniti oblik relacije. Funkcije f_n su gotovo uvijek relativno jednostavnog oblika, jer želimo kratku informaciju o složenosti, pa nevažne detalje odmah zanemarujemo.

U općem obliku (1.5.1), svaka vrijednost t_n **eksplicitno** ovisi o svim prethodnicima $t_{n-1}, \ldots, t_0$. Drugim riječima, t_n izravno ovisi o cijeloj prethodnoj povijesti niza (od t_0 nadalje). U praksi, ta ovisnost, obično, nije tako izravna, u smislu da zadaću veličine n ne rastavljamo eksplicitno na zadaće **svih** manjih veličina. Uglavnom, t_n izravno ovisi samo o svojim neposrednim prethodnicima. Najčešće, samo o jednom — o t_{n-1}. Druga varijanta je, da ovisi o najviše **fiksnom** broju prethodnika, na primjer, o k njih, gdje je k unaprijed poznat (zadan) i ne ovisi o n. Zbog toga, promatramo upravo takve rekurzivne relacije. Eksplicitna ovisnost u (1.5.1) tada ima bitno jednostavniji oblik. Naravno, ali sada indirektno, t_n i dalje ovisi o svim prethodnim članovima niza.

Počnimo stoga s jednostavnim rekurzivnim relacijama, u kojima svaka sljedeća vrijednost u nizu ovisi samo o **jednoj** prethodnoj. Takve rekurzivne relacije zovemo rekurzivnim relacijama **prvog** reda. Opći oblik rekurzivne (diferencijske) jednadžbe prvog reda je

$$t_n = f_n(t_{n-1}), \quad n \in \mathbb{N}.$$

Bez dodatnih informacija o funkcijama f_n, iako sve imaju istu domenu, i dalje se ništa posebno ne može reći o ponašanju niza t_n, za velike n.

Uzmimo onda najjednostavniji opći oblik funkcija f_n. Pretpostavimo da je f_n linearna (preciznije, afina) funkcija oblika

$$f_n(t) = b_n \cdot t + c_n, \quad n \in \mathbb{N}, \tag{1.5.2}$$

gdju su b_n, c_n zadani, recimo, realni brojevi, za svaki $n \in \mathbb{N}$. Još uvijek nije sasvim očito kako treba riješiti pripadnu rekurzivnu relaciju. Neka vrsta supstitucije, bilo unaprijed (od t_0 prema t_n), bilo unatrag (od t_n, silazno, prema t_0), mora dati rješenje, i ono, očito, ima oblik sume.

Umjesto izravnog "napada" na (1.5.2), u sljedeća dva primjera rješavamo jednostavne posebne slučajeve ove rekurzije, koji su bitna motivacija za kasnije, a zatim rješavamo čtav problem.

Primjer 1.5.1. Najjednostavniji netrivijalni oblik funkcija iz (1.5.2) je $f_n(t) = bt$. Sve funkcije f_n su iste, jer je $b_n = b$ i $c_n = 0$, za svaki $n \in \mathbb{N}$. Rekurzivna relacija ima oblik

$$t_n = bt_{n-1}, \quad n \in \mathbb{N}.$$

Za zadani t_0, uvrštavanje unaprijed daje redom članove niza

$$\begin{aligned} t_1 &= bt_0, \\ t_2 &= bt_1 = b^2 t_0, \\ &\vdots \\ t_n &= bt_{n-1} = b^2 t_{n-2} = \{\text{indukcija}\} = b^n t_0. \end{aligned}$$

Ako želimo biti matematički precizni, onda uvrštavanje **naslućuje** odgovor, a matematička indukcija ga **dokazuje**. ■

Napomena 1.5.2. Mogli smo uzeti i drugačiji oblik funckija f_n, u kojem je $b_n = 0$ i $c_n = c$. Rješenje je, naravno, konstantan niz $t_n = c$, za $n > 0$, (neovisno o t_0). Međutim, ovaj oblik nema algoritamski smisao. To bi odgovaralo slučaju da rješenje bilo koje zadaće ima konstantno trajanje, neovisno o veličini, što je trivijalni granični slučaj (potpuno neinteresantan u praksi). ■

Primjer 1.5.2. Nešto kompliciraniji oblik funkcija iz (1.5.2) je $f_n(t) = b_n t$. Funkcije f_n mogu biti različite, ako su koeficijenti b_n različiti, ali f_n je i dalje linearna (homogena, a ne afina) funkcija, zbog $c_n = 0$, za svaki $n \in \mathbb{N}$. Za zadani t_0, rješenje rekurzivne relacije

$$t_n = b_n t_{n-1}, \quad n \in \mathbb{N}$$

opet izlazi uvrštavanjem unaprijed

$$\begin{aligned}
&t_1 = b_1 t_0,\\
&t_2 = b_2 t_1 = b_2 b_1 t_0,\\
&\quad\vdots\\
&t_n = b_n t_{n-1} = b_n b_{n-1} t_{n-2} = \{\text{indukcija}\} = b_n b_{n-1} \cdots b_1 t_0.
\end{aligned}$$

tj.

$$t_n = \left(\prod_{j=1}^{n} b_j \right) t_0, \quad n \in \mathbb{N}_0.$$

■

Napadnimo sada cijeli problem.

Primjer 1.5.3. Opća linearna, nehomogena rekurzija prvog reda je

$$t_n = b_n t_{n-1} + c_n, \quad n \in \mathbb{N}.$$

Uz zadani t_0, ista strategija uvrštavanja unaprijed daje

$$\begin{aligned}
&t_1 = b_1 t_0 + c_0,\\
&t_2 = b_2 t_1 + c_1 = b_2 b_1 t_0 + b_2 c_1 + c_2,\\
&t_3 = b_3 t_2 + c_2 = b_3 b_2 b_1 t_0 + b_3 b_2 c_1 + b_3 c_2 + c_3,\\
&\quad\vdots
\end{aligned}$$

što ubrzo postaje zamorno, iako vodi do rješenja.

Zanimljivije je vidjeti da standardni pristup linearnim nehomogenim jednadžbama, koji kaže: "opće rješenje nehomogene jednadžbe je zbroj općeg rješenja homogene i partikularnog rješenja nehomogene jednadžbe", vrlo uredno radi i u ovom slučaju.

(A) Prvo uvodimo supstituciju koja ima oblik rješenja pripadne homogene jednadžbe, prema prethodnom primjeru, tj.

$$t_n = b_1 \cdots b_n u_n, \quad n \in \mathbb{N}.$$

Dobivamo novi nepoznati niz u_n, za $n \in \mathbb{N}$. Prethodna relacija će vrijediti i za $n = 0$, ako uzmemo isti početni član, tj. ako je $u_0 = t_0$ (produkt b-ova nestaje, jer je krajnji indeks produkta manji od početnog). Ova zamjena varijabli u polaznoj rekurziji daje

$$b_1 \cdots b_n u_n = b_n b_1 \cdots b_{n-1} u_{n-1} + c_n, \quad n \in \mathbb{N}.$$

Uočimo da su koeficijenti uz u_n na lijevoj strani i uz u_{n-1} na desnoj strani isti, pa podijelimo relaciju s tim koeficijentom $(b_1 \cdots b_n)$. Dobivamo

$$u_n = u_{n-1} + d_n, \quad n \in \mathbb{N}, \tag{1.5.3}$$

uz oznaku

$$d_n = \frac{c_n}{b_1 \cdots b_n}, \quad n \in \mathbb{N}. \tag{1.5.4}$$

Brojevi d_n su poznati, jer se računaju iz poznatih koeficijenata. Za u_n smo dobili bitno jednostavniju rekurziju od polazne za t_n.

(B) Ostaje još riješiti rekurzivnu relaciju (1.5.4) za u_n

$$u_n = u_{n-1} + d_n, \quad n \in \mathbb{N}.$$

No, to je trivijalno, jer se rješenje svodi na zbrajanje prethodnika

$$u_n = u_0 + \sum_{j=1}^{n} d_j, \quad n \in \mathbb{N}.$$

Ova relacija očito vrijedi i za $n = 0$.

(C) Vratimo se na polazni niz t_n, tako da u rješenje za u_n uvrstimo polaznu supstituciju $t_n = b_1 \cdots b_n u_n$. Izlazi

$$t_n = (b_1 \cdots b_n) \cdot \left(t_0 + \sum_{j=1}^{n} d_j \right), \quad n \in \mathbb{N}_0. \tag{1.5.5}$$

Vratimo li d-ove u b-ove i c-ove, relacijom (1.5.4), dobivamo konačno opće rješenje za (1.5.2). Nakon skraćivanja prvog faktora iz (1.5.5) i nazivnika od d_j, rješenje je

$$t_n = b_1 \cdots b_n t_0 + \sum_{j=1}^{n} b_{j+1} \cdots b_n c_j, \quad n \in \mathbb{N}_0. \tag{1.5.6}$$

Ako pažljivo promotrimo ova tri koraka, doći ćemo do zaključka da **nismo** baš radili kao što je najavljeno. Opće rješenje pripadne homogene rekurzije (iz Primjera 1.5.2.) je prvi član desne strane u (1.5.6). Lako se vidi da je drugi član (suma) iz (1.5.6), zapravo partikularno rješenje nehomogene rekurzije (1.5.2), dobiveno iz početnog uvjeta $t_0 = 0$. Njega nismo posebno tražili, već ga samo čitamo iz oblika rješenja.

Opisani put je nešto lakši, jer lagano daje "nehomogeni" dio rješenja, zato što je rekurzija za u_n, također, nehomogena, ali je homogeni dio trivijalan (svi b-ovi su jednaki 1). ■

Zadatak 1.5.1. Riješite rekurzivnu relaciju

$$t_n = 3t_{n-1} + n, \quad n \in \mathbb{N},$$

uz početni uvjet $t_0 = 0$.

Rješenje:

Rekurzija ima oblik (1.5.2), uz $b_n = 3$ i $c_n = n$. To možemo odmah uvrstiti u (1.5.5) ili (1.5.6). Ako ponovimo cijeli postupak rješenja iz prethodnog primjera, supstitucija je

$$t_n = 3^n u_n, \quad n \in \mathbb{N},$$

a rekurzija za u_n je

$$u_n = u_{n-1} + \frac{n}{3^n}, \quad n \in \mathbb{N}.$$

Zbog $u_0 = t_0 = 0$, sumacija daje

$$u_n = \sum_{j=1}^{n} \frac{j}{3^j}, \quad n \in \mathbb{N}.$$

Na kraju dobivamo rješenje

$$t_n = 3^n \sum_{j=1}^{n} \frac{j}{3^j} = \sum_{j=1}^{n} j\, 3^{n-j}, \quad n \in \mathbb{N}_0.$$

■

2. Sortiranje

U ovom poglavlju analizirat ćemo neke standardne algoritme za sortiranje podataka. Promatrat ćemo samo tzv. opće algoritme sortiranja koji, u principu, rade za bilo koje tipove podataka, a koriste **usporedbe** za određivanje korektnog poretka.

Dosad smo standardno radili analizu složenosti pojedinih algoritama u **najgorem** slučaju, jer se ova analiza, uglavnom, relativno jednostavno radi. Za algoritme sortiranja uspoređivanjem napravit ćemo (po prvi puta) i dva drugačija rezultata o složenosti.

- Izvest ćemo **donju ogradu** za složenost **bilo kojeg** algoritma sortiranja uspoređivanjem;
- Provest ćemo analizu **prosječne** složenosti *quicksort* algoritma, koja opravdava njegovu čestu upotrebu u praksi (recimo, kao standardnog dijela sistemske biblioteke jezika C).

Prije nego što punu pažnju posvetimo algoritmima za sortiranje, možda je zgodno odgovoriti na pitanje čemu to uopće služi.

2.1. Zašto sortiramo?

Odgovor je vrlo jednostavan: "Zato da bismo brže tražili!".

Traženje nekog podatka u **sortiranom** nizu, u najgorem slučaju, **logaritamski** ovisi o duljini niza. Dovoljno je iskoristiti postupak **binarnog** pretraživanja. Za razliku od toga, traženje u **nesortiranom** nizu, i u prosjeku i u najgorem slučaju, **linearno** ovisi o duljini niza, jer se traženje svodi na **sekvencijalno** pretraživanje, odnosno, provjeravanje.

Dakle, ako **često** tražimo podatke u nekom nizu, onda se **isplati** taj niz prvo sortirati, pa onda tražiti.

2.2. Model sortiranja

Sasvim općenito, pretpostavljamo da **svi** objekti koje treba sortirati imaju **istu** strukturu (tip), ali ta struktura može biti složena. Programski gledano, pretpostavljamo da svi podaci pripadaju nekom složenom tipu podataka. Možemo uzeti da taj tip podataka ima oblik **zapisa** (tip **record** u Pascalu, odnosno, struktura u C-u). Dakle, podatak se sastoji iz više rubrika (komponenti), koje mogu biti različitih tipova.

Nadalje, pretpostavljamo da **jedna** od tih komponenti ima **dobro uređeni** tip vrijednosti, tj., za taj tip (ili na tom tipu) definirana je relacija linearnog uređaja koju označavamo s $\leq$. Tu komponentu objekta zovemo "ključ" (engl. "key").

Naš zadatak je: zadani (konačni) niz od n takvih objekata **poredati** uzlazno (ili silazno) po vrijednosti komponente "ključ". Preciznije, neka su zadani zapisi $r_1, r_2, \ldots, r_n$ s pripadnim vrijednostima ključeva

$$k_1, k_2, \ldots, k_n,$$

respektivno, tj., ključ k_i se nalazi u zapisu r_i, za $i = 1, \ldots, n$. Te zapise treba **poredati** (preurediti) u redoslijedu

$$r_{i_1}, r_{i_2}, \ldots, r_{i_n},$$

tako da za pripadne ključeve vrijedi

$$k_{i_1} \leq k_{i_2} \leq \cdots \leq k_{i_n},$$

tj. ključevi su **uzlazno** poredani. U nastavku gledamo samo uzlazno sortiranje, a modifikacije svih algoritama za silazno sortiranje dobivaju se jednostavnom zamjenom relacije $\leq$ u $\geq$, kod svih uspoređivanja ključeva.

Općenito, zapisa s istom vrijednošću ključa može biti i više, tj. ne tražimo da sve vrijednosti ključeva budu različite. Nadalje, dozvoljavamo da takvi zapisi s istim ključevima imaju bilo koji međusobni poredak u uređenom (sortiranom) nizu. Drugim riječima, osim poretka po ključu, nema drugih zahtjeva na poredak zapisa, poput "sekundarnog" ključa i sl.

2.3. Quicksort

Algoritam *quicksort* smo već napravili.

2.4. Prosječna vremenska složenost Quicksorta

Za nalaženje prosječne složenosti moramo uvesti neke pretpostavke o distribuciji ulaza i definirati prosječnu složenost.

Polazne pretpostavke:

- Svi polazni redoslijedi (permutacije) su jednako vjerojatni.
- Nema jednakih ključeva. Ovo je samo zbog pojednostavljenja analize. Naime, isti ključevi samo ubrzavaju algoritam, jer blok istih ključeva ne treba sortirati. Međutim, pojava istih ključeva komplicira prethodnu pretpostavku. Sve polazne permutacije nisu više jednako vjerojatne i trebalo bi dozvoliti permutacije s ponavljanjem.

To znači da, za zadani n, na ulazu imamo točno $n!$ različitih, jednako vjerojatnih permutacija objekata. **Prosječno vrijeme** je srednja vrijednost vremenske složenosti po svim ulaznim permutacijama (porecima).

Za dodatno pojednostavljenje analize, uvodimo još jednu pretpostavku:

- U trenutku kad pozivamo *quicksort*(i, j) na komadu polja $A[i..j]$, za bilo koje i, j, svi mogući redoslijedi objekata $A[i], \ldots, A[j]$ su, također, jednako vjerojatni.

Ovo je, barem naizgled, mnogo jača pretpostavka od prve. Ranija pretpostavka daje ovo isto, ali samo za prvi (vanjski) poziv *quicksort*$(1, n)$.

Opravdanje (umjesto strogog dokaza) za ovu jaču pretpostavku svodi se na: "bilo koji komadi jednako vjerojatnih rasporeda su, također, jednako vjerojatni". Prije poziva *quicksort*(i, j), svi raniji pivotni ključevi v nisu napravili particiju — rez u tom dijelu polja $A[i..j]$. Tj. ključevi u tom dijelu polja su bili ili **svi** strogo manji od v, ili su bili **svi** veći ili jednaki v.

Neka je $T(n)$ prosječno vrijeme potrebno algoritmu *quicksort* za sortiranje polja od n elemenata. Ideja za procjenu $T(n)$ — odozgo ograditi $T(n)$ u rekurzivnom obliku (prema algoritmu).

Ta rekurzija starta s $n = 1$. Za jedan element nema rekurzije u *quicksort* algoritmu. Očito je $T(1) = c_1$ (ili $T(1) \leq c_1$) za neku konstantu $c_1 > 0$.

Pretpostavimo da je $n > 1$. Zbog pretpostavke da su svi ključevi različiti, postoje bar 2 različita ključa. Tada *quicksort* nalazi pivotni ključ i particionira

polje. Taj dio posla (*findpivot* i *partition*) traje najviše linearno u duljini polja (neovisno o rasporedu elemenata u polju)

$$t(n) \leq c_2 n, \tag{2.4.1}$$

za neku konstantu $c_2 > 0$. Aditivni dio vremena $t(n)$ ograđujemo multiplikativno, tj. umjesto $c_2' n + c'$, pišemo samo $c_2 n$, uz $c_2' \leq c_2 \leq c_2' + c'$. Uočimo da ova gornja ograda vrijedi za bilo koji polazni poredak elemenata, pa vrijedi i u prosjeku.

Nakon ovog dijela posla, *quicksort* rekurzivno sortira 2 manja polja — potpolja polaznog polja. Označimo s $T_R(n)$ prosječno trajanje rekurzije na ta 2 potpolja. Tada je

$$T(n) \leq T_R(n) + t(n). \tag{2.4.2}$$

Za primjenu ove ocjene treba još naći prosječno trajanje rekurzije $T_R(n)$. Očito je

$$T_R(n) = \sum_{i=1}^{n-1} p_i \, [\, T(i) + T(n-i) \,], \tag{2.4.3}$$

gdje je p_i vjerojatnost da lijevo potpolje (elementi s ključevima strogo manjim od v) ima točno i elemenata. Zbog različitosti ključeva, sigurno imamo 2 neprazna potpolja, pa za i vrijedi $i \in \{1, \ldots, n-1\}$. Ako neko od dva potpolja ima samo jedan element (pa ga ne treba sortirati), relacija (2.4.3) i dalje vrijedi, bez obzira na to da li se *quicksort* tada rekurzivno zove ili provjerava duljinu radnog polja.

Za nastavak ocjene treba naći vjerojatnosti p_i. Bilo bi lijepo kad bismo odmah mogli zaključiti da su sve veličine lijevog potpolja (od 1 do $n-1$) jednako vjerojatne, tj. da je

$$p_1 = p_2 = \cdots = p_{n-1} = \frac{1}{n-1}. \tag{2.4.4}$$

Međutim, to **ne vrijedi**. Za nalaženje p_i treba precizno analizirati rad procedure *findpivot*.

Pretpostavimo da je lijevo potpolje (nakon particije) duljine točno i, za neki $i \in \{1, \ldots, n-1\}$. U našoj realizaciji, *findpivot* bira strogo većeg od prva 2 (različita) ključa koje nađe, gledano u polaznom poretku i to od manjih prema većim indeksima. Svi ključevi su različiti, pa *findpivot* nalazi različite ključeve na prva dva mjesta i uzima većeg od njih.

Prema našem algoritmu, nakon particije, pivotni element s ključem v mora biti element s najmanjim ključem u desnom potpolju (elementi s ključevima većim ili jednakim od v). Ako je lijevo potpolje duljine i, zbog pretpostavke da su svi ključevi različiti, pivotni element ima poziciju točno $i+1$ u sortiranom poretku ovih n elemenata (u trenutnom potpolju). Drugim riječima, radni dio polja sadrži točno i elemenata s ključevima strogo manjim od pivotnog.

U izboru pivotnog ključa imamo slijedeće dvije mogućnosti.

(A) Pivot je na prvom mjestu (u polaznom poretku), a jedan od i elemenata manjih od njega je na drugom mjestu. Sada koristimo pretpostavku o jednakoj vjerojatnosti svih polaznih poredaka. Vjerojatnost da je bilo koji određeni element (na pr. $i+1$–i po veličini, tj. pivotni) na prvom mjestu je

$$p' = \frac{1}{n},$$

jer imamo n jednako vjerojatnih mogućnosti za taj element. Nadalje, ako je on na prvom mjestu, onda je vjerojatnost da se neki od i manjih od njega, među $n-1$ preostalih, nalazi na drugom mjestu, jednaka

$$p'' = \frac{i}{n-1}.$$

Dakle, vjerojatnost da se ovaj slučaj dogodi u izboru pivota je

$$p_A = p'p'' = \frac{i}{n(n-1)}.$$

(B) Pivot je na drugom mjestu (u polaznom poretku), a jedan od i elemenata manjih od njega (među $n-1$ preostalih) je na prvom mjestu. Potpuno analogno, vjerojatnost da se ovaj slučaj dogodi je ista

$$p_B = p_A = \frac{i}{n(n-1)}.$$

Na kraju, jer su (A) i (B) jedine dvije i to disjunktne mogućnosti da dobijemo lijevo potpolje duljine i, izlazi

$$p_i = p_A + p_B = \frac{2i}{n(n-1)}. \tag{2.4.5}$$

To znači da veće duljine lijevog potpolja imaju veću vjerojatnost, tj. u prosjeku je lijevo potpolje veće od desnog. Pokazat će se da ovaj fenomen ne utječe bitno na konačni rezultat, u usporedbi s jednako vjerojatnim duljinama lijevog potpolja.

Uvrštavanjem p_i iz (2.4.5) u (2.4.3) dobivamo

$$T_R(n) = \sum_{i=1}^{n-1} \frac{2i}{n(n-1)} \left[T(i) + T(n-i) \right]. \tag{2.4.6}$$

Za pojednostavljenje ove formule koristimo slijedeću pomoćnu činjenicu. Neka je f bilo koja realna funkcija definirana na skupu točaka $\{1, \ldots, n-1\}$. Onda vrijedi

$$\sum_{i=1}^{n-1} f(i) = \frac{1}{2} \sum_{i=1}^{n-1} \left[f(i) + f(n-i) \right]. \tag{2.4.7}$$

(Dokaz je trivijalan.) Ako stavimo da je

$$f(i) = \frac{2i}{n(n-1)} \left[T(i) + T(n-i) \right],$$

primjenom prethodne relacije u (2.4.6) dobivamo

$$T_R(n) = \frac{1}{2} \sum_{i=1}^{n-1} \left[\frac{2i}{n(n-1)} \left[T(i) + T(n-i) \right] + \frac{2(n-i)}{n(n-1)} \left[T(n-i) + T(i) \right] \right].$$

Prvo skratimo faktore 1/2 i 2, a zatim uočimo da prvi i drugi član u zagradi imaju iste faktore u uglatim zagradama, pa zbrojimo pripadne faktore ispred tih zagrada

$$\frac{i}{n(n-1)} + \frac{n-i}{n(n-1)} = \frac{n}{n(n-1)} = \frac{1}{n-1}.$$

Tako dobivamo da je

$$T_R(n) = \frac{1}{n-1} \sum_{i=1}^{n-1} \left[T(i) + T(n-i) \right]. \tag{2.4.8}$$

Prije nastavka ocjene, promotrimo pobliže prethodni rezultat. Kad bismo u polaznoj relaciji (2.4.3) uzeli da su sve duljine i lijevog potpolja jednako vjerojatne, tj. uvrstili (2.4.4), dobili bismo upravo prethodnu relaciju. To pokazuje da su jednako vjerojatne duljine lijevog potpolja **podjednako** dobar izbor kao i naš, jer dobivamo istu relaciju za prosječnu složenost. Drugim riječima, izbor većeg od prva dva ključa neće pokvariti konačni rezultat o prosječnoj složenosti.

Iz relacije (2.4.8), primijenom (2.4.7) s $f(i) = T(i)$, izlazi

$$T_R(n) = \frac{2}{n-1} \sum_{i=1}^{n-1} T(i).$$

Uvrštavanjem ove relacije i relacije (2.4.1) u (2.4.2) dobivamo rekurzivnu ocjenu za $T(n)$

$$T(n) \leq \frac{2}{n-1} \sum_{i=1}^{n-1} T(i) + c_2 n. \tag{2.4.9}$$

Ova rekurzija se može i direktno rješavati (pokušajte sami). Na osnovu ranijih rezultata o rekurzijama, očekujemo da za rješenje $T(n)$ vrijedi asimptotska relacija $T(n) = O(n \log n)$.

Teorem 2.4.1. *Neka je $T(n)$ prosječna složenost quicksort algoritma. Postoji konstanta $c > 0$, takva da je*

$$T(n) \leq c\, n \log n, \quad \forall n \geq 2. \tag{2.4.10}$$

Posebno, asimptotski vrijedi $T(n) = O(n \log n)$.

Dokaz:

Pretpostavimo, radi jednostavnosti, da je $\log = \log_2 = \lg$. Konstantu c nalazimo tehnikom konstruktivne indukcije. Dokažimo prvo bazu indukcije za $n = 2$. Rekurzija (2.4.9) tada glasi

$$T(2) \leq 2T(1) + 2c_2.$$

Zbog $T(1) \leq c_1$, izlazi

$$T(2) \leq 2(c_1 + c_2).$$

Tražena relacija (2.4.10) za $n = 2$ ima oblik $T(2) \leq 2c \lg 2 = 2c$. Ako izaberemo c tako da vrijedi

$$2(c_1 + c_2) \leq 2c,$$

ili, ekvivalentno,

$$c \geq c_1 + c_2, \tag{2.4.11}$$

onda vrijedi baza indukcije.

Neka je $n > 2$. Pretpostavimo da je c tako odabran da vrijedi (2.4.11) i da je

$$T(i) \leq c\, i \lg i, \quad i = 2, \ldots, n-1.$$

Uvrštavanjem ovih ocjena i $T(1) \leq c_1$ u rekurziju (2.4.9) dobivamo

$$T(n) \leq \frac{2c}{n-1} \sum_{i=2}^{n-1} i \lg i + \frac{2c_1}{n-1} + c_2 n. \tag{2.4.12}$$

Suma na desnoj strani ove relacije ostaje ista, ako joj dodamo član za $i = 1$, zbog $\lg 1 = 0$. Ovu sumu rastavljamo u dva dijela sličnih duljina, u obliku

$$\sum_{i=2}^{n-1} i \lg i = \sum_{i=1}^{n-1} i \lg i = \sum_{i=1}^{\lfloor n/2 \rfloor} i \lg i + \sum_{i=\lfloor n/2 \rfloor + 1}^{n-1} i \lg i,$$

a zatim koristimo ocjene

$$\begin{aligned} &\lg i \leq \lg \frac{n}{2} = \lg n - 1, \quad i = 1, \ldots, \lfloor n/2 \rfloor, \\ &\lg i < \lg n, \quad i = \lfloor n/2 \rfloor + 1, \ldots, n-1, \end{aligned}$$

Zbrajanjem ovih ocjena dobivamo

$$\begin{aligned} \sum_{i=2}^{n-1} i \lg i &< (\lg n - 1) \sum_{i=1}^{\lfloor n/2 \rfloor} i + \lg n \sum_{i=\lfloor n/2 \rfloor + 1}^{n-1} i \\ &= \lg n \sum_{i=1}^{n-1} i - \sum_{i=1}^{\lfloor n/2 \rfloor} i \\ &= \frac{1}{2} n(n-1) \lg n - \frac{1}{2} \left\lfloor \frac{n}{2} \right\rfloor \left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right), \end{aligned}$$

pa u (2.4.12) izlazi

$$T(n) < \frac{2c}{n-1}\left[\frac{1}{2}\,n(n-1)\lg n - \frac{1}{2}\left\lfloor\frac{n}{2}\right\rfloor\left(\left\lfloor\frac{n}{2}\right\rfloor+1\right)\right] + \frac{2c_1}{n-1} + c_2 n,$$

ili

$$T(n) < c\,n\lg n - \frac{c}{n-1}\left\lfloor\frac{n}{2}\right\rfloor\left(\left\lfloor\frac{n}{2}\right\rfloor+1\right) + \frac{2c_1}{n-1} + c_2 n. \tag{2.4.13}$$

Slijedeća ideja je odabrati c tako da suma zadnja 3 člana na desnoj strani bude manja ili jednaka 0. Uočimo prvo da je

$$\frac{n-1}{2} \le \left\lfloor\frac{n}{2}\right\rfloor < \left\lfloor\frac{n}{2}\right\rfloor + 1,$$

Množenjem izlazi da za $n \ge 2$ vrijedi

$$\frac{(n-1)^2}{4} < \left\lfloor\frac{n}{2}\right\rfloor\left(\left\lfloor\frac{n}{2}\right\rfloor+1\right).$$

Primjenom ove ocjene u (2.4.13) dobivamo

$$\begin{aligned} T(n) &< c\,n\lg n - \frac{c}{n-1}\,\frac{(n-1)^2}{4} + \frac{2c_1}{n-1} + c_2 n \\ &= c\,n\lg n - \frac{c(n-1)}{4} + \frac{2c_1}{n-1} + c_2 n \\ &= c\,n\lg n + \frac{(n-1)}{4}\left[-c + \frac{8c_1}{(n-1)^2} + \frac{4c_2 n}{n-1}\right]. \end{aligned}$$

Na kraju, za svaki $n \ge 2$ vrijedi

$$\frac{1}{(n-1)^2} \le 1, \quad \frac{n}{n-1} \le 2,$$

tako da je

$$T(n) < c\,n\lg n + \frac{(n-1)}{4}\left[-c + 8c_1 + 8c_2\right]. \tag{2.4.14}$$

Ako izaberemo c tako da vrijedi

$$c \ge 8(c_1 + c_2), \tag{2.4.15}$$

onda je izraz u uglatoj zagradi u (2.4.14) manji ili jednak od 0, odakle slijedi željena ocjena

$$T(n) < c\,n\lg n.$$

Usporedbom uvjeta (2.4.11) i (2.4.15) za c, zaključujemo da tvrdnja teorema vrijedi, čim c zadovoljava (2.4.15). Očito je da uz malo truda možemo dobiti i bolji (blaži) uvjet na c, ali to nije posebno bitno za konačni rezultat. ■

Ovaj rezultat o prosječnoj složenosti *quicksort* algoritma vrijedi za odabranu *findpivot* proceduru, koja bira pivotni ključ kao veći od prva dva ključa (uz pretpostavku različitih ključeva).

Postavlja se pitanje da li isti zaključak vrijedi i za drugačije izbore pivotnih ključeva. Odgovor je potvrdan, uz vrlo blage pretpostavke.

Napomena 2.4.1. Izbor pivotnog ključa za koji vrijedi (2.4.4) je pravi, stvarno slučajni izbor pivotnog ključa u svakom koraku algoritma. Tada v s jednakom vjerojatnošću može biti bilo koji po veličini element u radnom polju. Napomena uz relaciju (2.4.8) pokazuje da i za taj izbor vrijedi ista ocjena prosječne složenosti.

Ovaj rezultat ne iznenađuje. Naime, u idealnom slučaju, oba bi potpolja imala uvijek iste duljine ili maksimalno za 1 različite duljine. Tj. za dobre izbore pivotnih ključeva vrijedi da se maksimum p_i dostiže za srednje vrijednosti $i \approx n/2$, a opada prema rubovima.

Obzirom na to da i u idealnom slučaju dobivamo isti oblik ocjene, možemo zaključiti da izbor većeg od prva dva ključa ne utječe bitno na konačni rezultat o prosječnoj složenosti. ■

2.5. Prosječna složenost Quicksorta (broj usporedbi)

Najlakši uvid u prosječnu složenost *quicksort* algoritma dobivamo tako da promatramo **prosječni broj** elementarnih operacija — uspoređivanja ključeva, odnosno, zamjena (ili kopiranja) elemenata (objekata u polju).

Za pojednostavljenje modela, da ne bismo strogo ovisili o preciznoj implementaciji svih detalja, uzmimo da algoritam *quicksort* ima ovaj opći oblik:

Algoritam *quicksort*

Ulaz: Polje $A[1..n]$ s n elemenata.
Izlaz: Elementi u A sortirani nepadajuće po ključevima.

quicksort$(A, 1, n)$ { vanjski poziv }.

procedure *quicksort*(A, l, r)
  if $l < r$ **then**
    findpivot$(A[l..r], pivot\_index)$
    partition$(A[l..r], pivot\_index, j)$
    quicksort$(A, l, j-1)$
    quicksort$(A, j+1, r)$
  end if

Algoritam *findpivot* bira pivotni ključ u radnom polju $A[l..r]$ i vraća njegov indeks *pivot_index*, a *partition* stavlja taj element na njegovo pravo "sortirano" mjesto s indeksom j i dijeli radno polje u dva potpolja: $A[l..j-1]$ sadrži elemente s ključevima manjim ili jednakim od pivotnog, a $A[j+1..r]$ elemente s ključevima većim ili jednakim od pivotnog.

Uočimo da je nakon toga $A[j]$ na svom mjestu, pa ga **ne treba** uključiti niti u jedno od dva potpolja za nastavak rekurzivnog sortiranja. Oba potpolja su strogo kraća od polaznog, a može se dogoditi da jedno od njih ima "duljinu" jednaku 0.

Za nalaženje prosječne složenosti moramo uvesti neke pretpostavke o distribuciji ulaza i definirati prosječnu složenost.

Polazne pretpostavke:

- Svi polazni redoslijedi ključeva (permutacije) su jednako vjerojatni. Lako se vidi da stvarne vrijednosti ključeva uopće nisu bitne, već samo njihov poredak, tj. ulazna permutacija (obzirom na sortiranu).
- Nema jednakih ključeva. Ovo je samo zbog pojednostavljenja analize. Naime, isti ključevi samo ubrzavaju algoritam, jer blok istih ključeva ne treba sortirati. Međutim, pojava istih ključeva komplicira prethodnu pretpostavku.

Sve polazne permutacije nisu više jednako vjerojatne i trebalo bi dozvoliti permutacije s ponavljanjem.

Obzirom na to da stvarne vrijednosti ključeva nisu bitne, već samo njihov poredak, možemo čak pretpostaviti da su ulazni ključevi upravo svi prirodni brojevi od 1 do n (u nekom poretku, tj. permutaciji).

Uz ove pretpostavke, za zadani n, na ulazu imamo točno $n!$ različitih, jednako vjerojatnih permutacija objekata. **Prosječna složenost** je srednja vrijednost složenosti po svim ulaznim permutacijama objekata.

Za dodatno pojednostavljenje analize, uvodimo još jednu pretpostavku:

- U trenutku kad pozivamo *quicksort*(A, l, r) na komadu polja $A[l..r]$, za bilo koje l, r, svi mogući redoslijedi objekata $A[l], \ldots, A[r]$ su, također, jednako vjerojatni.

Ovo je, barem naizgled, mnogo jača pretpostavka od prve. Ranija pretpostavka daje ovo isto, ali samo za prvi (vanjski) poziv *quicksort*$(A, 1, n)$.

Opravdanje (umjesto strogog dokaza) za ovu jaču pretpostavku svodi se na: "bilo koji komadi jednako vjerojatnih rasporeda su, također, jednako vjerojatni".

Prije poziva *quicksort*(A, l, r), svi raniji pivotni ključevi v nisu napravili particiju — rez u tom dijelu polja $A[l..r]$. Trenutni ključevi u tom dijelu polja su ili bili **svi** manji (ili jednaki) od v, ili bili **svi** veći (ili jednaki) od v, i isto vrijedi za sve ranije pivotne ključeve — znak nejednakosti može biti drugačiji za razne ranije pivotne ključeve, ali taj znak vrijedi za **sve** ključeve u $A[l..r]$ obzirom na dotični raniji pivotni ključ.

Ranije zamjene su mogle promijeniti sadržaj tog dijela polja, ali uz pretpostavku slučajnog polaznog poretka, te zamjene su, također, "slučajne", i to bez obzira na to kako smo izbrali pivotni ključ, tako da opet dobivamo "slučajni" poredak u tom dijelu polja $A[l..r]$. Dakle, sortiranje tog dijela $A[l..r]$ trenutnog polja ponaša se **isto** kao da smo to polje (duljine $r - l + 1$) poslali izvana kao **polazno** polje u *quicksort* algoritam.

Bez obzira na detalje realizacije *quicksort* algoritma, usporedbe ključeva i zamjene elemenata javljaju se samo u prvom "nerekurzivnom" dijelu algoritma — izboru pivotnog ključa i particiji radnog dijela polja. Odmah se vidi da je broj zamjena elemenata uvijek **manji ili jednak** od broja usporedbi ključeva. Dakle, ako želimo gornju ogradu za određenu vrstu složenosti (u ovom slučaju, to je prosječna slovzenost), dovoljno je gledati samo broj usporedbi ključeva. Zato, u daljnjem, složenost mjerimo upravo brojem usporedbi ključeva.

Neka je $C(n)$ prosječan broj usporedbi ključeva u *quicksort* algoritmu za ulazno polje od n elemenata. Po pretpostavci, na ulazu imamo točno $n!$ različitih per-

mutacija objekata (ključeva). Sasvim općenito, po definiciji prosječne složenosti, vrijedi

$$C(n) = \sum_{p \in S_n} P(p)\, C(n,p),$$

gdje je $C(n,p)$ točan broj usporedbi ključeva u algoritmu za ulaznu permutaciju ključeva $p \in S_n$, a $P(p)$ je vjerojatnost da će se upravo ta permutacija p pojaviti na ulazu.

Po našoj pretpostavci, sve te ulazne permutacije su jednako vjerojatne, tj. $P(p) = 1/n!$, za svaki $p \in S_n$, pa je

$$C(n) = \frac{1}{n!} \sum_{p \in S_n} C(n,p). \tag{2.5.1}$$

Dalje postaje komplicirano, jer bi trebalo naći $C(n,p)$, za svaki $p \in S_n$ (točno po algoritmu), uvrstiti to u (2.5.1) i riješiti dobivenu rekurziju.

Umjesto toga, idemo probati napisati jednostavniju rekurziju za $C(n)$, opet prema algoritmu.

Za početak, uočimo da je $C(1) = 0$, jer je polje od jednog elementa već sortirano i nemamo što uspoređivati. Također, zgodno je uzeti i $C(0) = 0$, ako nam zatreba kao dodatni početni uvjet (ako prekid rekurzije realiziramo tako da duljina polja smije biti 0).

Neka je sada $n > 1$. Tada *quicksort* bira pivotni ključ, particionira polje i rekurzivno sortira dva manja potpolja.

Pretpostavimo da se izbor pivotnog ključa realizira na najjednostavniji mogući način — potpuno slučajno se izabere neki element u (radnom) polju, bez ikakvih uspoređivanja. To se dosta često koristi u praksi, a uobičajni izbori su prvi, srednji, ili zadnji element, tj.

$$pivot\_index := l; \quad pivot\_index := (l + r) \textbf{ div } 2; \quad pivot\_index := r.$$

Bez smanjenja općenitosti, možemo uzeti da se bira prvi ključ (tj. $pivot\_index := l$).

Pogledajmo detaljno prvi korak (ili poziv) algoritma, kad radimo na cijelom polju $A[1..n]$. Zbog jednake vjerojatnosti svih ulaznih permutacija i različitosti svih ključeva, svaki ključ iz polja ima **jednaku vjerojatnost** da bude izabran kao pivotni ključ — tj. da se nalazi (recimo) na prvom mjestu u polju. Dakle, vjerojatnost da će točno j-ti po veličini ključ biti izabran kao pivotni, između n mogućih različitih ključeva, je

$$p_{j,n} = \frac{1}{n}, \quad j = 1, \ldots, n. \tag{2.5.2}$$

Onda je

$$C(n) = \sum_{j=1}^{n} p_{j,n} C(n, j), \tag{2.5.3}$$

gdje je $C(n, j)$ prosječan broj usporedbi ključeva u algoritmu, uz uvjet da je pivotni ključ j-ti po veličini.

Treba još naći $C(n, j)$. Pretpostavimo zato da je algoritam izabrao j-ti ključ po veličini kao pivotni. To znači da je pravo "sortirano" mjesto pivotnog elementa upravo $A[j]$. Dakle, *partition* mora staviti taj element na mjesto $A[j]$ i raspodijeliti polje $A[1..n]$ oko tog mjesta j. Nakon toga, rekurzivno pozivamo *quicksort* na poljima $A[1..j-1]$ i $A[j+1..n]$.

Zato je $C(n, j)$ jednak zbroju prosječnog broja usporedbi u *partition* i u dva rekurzivna poziva *quicksort*.

Znamo da *partition* mora usporediti pivotni ključ sa **svim** preostalim ključevima u radnom dijelu polja, tj. imamo barem $n-1$ usporedbi ključeva. Uz malo pažnje pri realizaciji *partition* algoritma, zaista ne trebamo više od $n-1$ usporedbi. U praksi se dosta često dogodi da imamo jednu ili dvije usporedbe više (tj. n ili $n+1$), zbog dodatnih rubnih elemenata za zaustavljenje petlji (Knuth), ili preklapanja indeksa na samom kraju particioniranja. Kao što ćemo kasnije pokazati, tih par usporedbi više neće bitno promijeniti krajnji rezultat o složenosti.

Fali nam još prosječan broj usporedbi u dva rekurzivna poziva. To, naravno, ovisi o raspodjeli elemenata u ulaznim poljima za ta dva poziva. Može se dokazati da su sve permutacije ključeva u poljima $A[1..j-1]$ i $A[j+1..n]$, također, jednako vjerojatne, jer to vrijedi i za $A[1..n]$. Dokaz nije jako težak, pa ga pokušajte napraviti sami (potrebno je nešto kombinatoričkog znanja).

Zaključujemo da je prosječan broj usporedbi u svakom od ta dva rekurzivna poziva, isti kao da smo odgovarajuće potpolje poslali (izvana) kao polazno polje u *quicksort*, uz pretpostavku jednake vjerojatnosti svih mogućih rasporeda ključeva, tj. jednak je C(duljina polja). Dakle, prosječan broj usporedbi za *quicksort*$(A, 1, j-1)$ je $C(j-1)$, a za *quicksort*$(A, j+1, n)$ je $C(n-j)$.

Konačno, zbrajanjem dobivamo da je

$$C(n, j) = (n-1) + C(j-1) + C(n-j), \quad j = 1, \ldots, n. \tag{2.5.4}$$

Kad to uvrstimo u (2.5.3), zajedno s $p_{j,n} = 1/n$, izlazi rekurzija za prosječni broj usporedbi ključeva $C(n)$

$$\begin{aligned} C(n) &= \sum_{j=1}^{n} \frac{1}{n} \left(n - 1 + C(j-1) + C(n-j)\right) \\ &= (n-1) + \frac{1}{n} \sum_{j=1}^{n} (C(j-1) + C(n-j)). \end{aligned} \tag{2.5.5}$$

Zbog

$$\sum_{j=1}^{n} C(n-j) = C(n-1) + C(n-2) + \cdots + C(0) = \sum_{j=1}^{n} C(j-1),$$

relaciju (2.5.5) možemo napisati u obliku

$$C(n) = (n-1) + \frac{2}{n} \sum_{j=1}^{n} C(j-1). \tag{2.5.6}$$

U usporedbi s rekurzijama koje smo dosad rješavali, ova rekurzija izgleda kompliciranije, jer "nova" vrijednost $C(n)$ ovisi o **svim** prethodnim vrijednostima $C(n-1), \ldots, C(0)$, tj. cijela "povijest" prošlih članova niza određuje svaki novi član.

Pa ipak, znak sume možemo jednostavno ukloniti, po ugledu na rješenje jednog od prvih primjera rekurzivnih relacija. Prvo, pomnožimo (2.5.6) s n

$$nC(n) = n(n-1) + 2\sum_{j=1}^{n} C(j-1). \tag{2.5.7}$$

Ako zamijenimo n s $n-1$ u (2.5.7), dobivamo

$$(n-1)C(n-1) = (n-1)(n-2) + 2\sum_{j=1}^{n-1} C(j-1). \tag{2.5.8}$$

Oduzmemo (2.5.8) od (2.5.7)

$$nC(n) - (n-1)C(n-1) = n(n-1) - (n-1)(n-2) + 2C(n-1),$$

preuredimo ovu relaciju u

$$nC(n) = (n+1)C(n-1) + 2(n-1),$$

i sve podijelimo s $n(n+1)$, tako da dobijemo

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2(n-1)}{n(n+1)}. \tag{2.5.9}$$

Ova rekurzija vrijedi za $n \geq 1$, a početni uvjet je $C(0) = 0$. Naravno, možemo startati i s $C(1) = 0$.

Sad uvedemo novi niz $D(n)$ supstitucijom

$$D(n) = \frac{C(n)}{n+1}, \quad n \geq 0.$$

Relaciju (2.5.9) možemo napisati u terminima novog niza D

$$D(n) = D(n-1) + \frac{2(n-1)}{n(n+1)}, \tag{2.5.10}$$

a početni uvjet je očito $D(0) = 0$, a može i $D(1) = 0$. Rješenje rekurzije (2.5.10) dobivamo jednostavnim zbrajanjem "nehomogenih" članova

$$D(n) = 2\sum_{j=1}^{n} \frac{j-1}{j(j+1)}, \quad n \geq 1, \tag{2.5.11}$$

s tim da suma može početi i s $j = 2$ (prvi član je 0).

Na kraju, treba još pojednostavniti desnu stranu u (2.5.11). Uočimo da je

$$\frac{j-1}{j(j+1)} = \frac{j-1}{j} - \frac{j-1}{j+1} = 1 - \frac{1}{j} - 1 + \frac{2}{j+1} = \frac{2}{j+1} - \frac{1}{j}.$$

Onda je

$$\begin{aligned}
\sum_{j=1}^{n} \frac{j-1}{j(j+1)} &= \sum_{j=1}^{n} \frac{2}{j+1} - \sum_{j=1}^{n} \frac{1}{j} \\
&= 2\sum_{j=2}^{n+1} \frac{1}{j} - \sum_{j=1}^{n} \frac{1}{j} \\
&= 2\sum_{j=1}^{n} \frac{1}{j} + \frac{2}{n+1} - 2 - \sum_{j=1}^{n} \frac{1}{j} \\
&= \sum_{j=1}^{n} \frac{1}{j} - \frac{2n}{n+1} \\
&= H_n - \frac{2n}{n+1},
\end{aligned}$$

gdje je H_n n-ti harmonijski broj (n-ta parcijalna suma harmonijskog reda)

$$H_n := \sum_{j=1}^{n} \frac{1}{j}.$$

Red veličine harmonijskih brojeva u funkciji od n možemo lako procijeniti preko aproksimacije integrala funkcije $f(x) = 1/x$ "produljenom" pravokutnom formulom (tj. donjim i gornjim Darbouxovim sumama), s tim da pravokutnici imaju donju i gornju stranicu duljine 1.

Gornju ogradu dobivamo pravokutnicima "ispod" grafa funkcije $1/x$, tako da je

$$H_n = \sum_{j=1}^{n} \frac{1}{j} = 1 + \sum_{j=2}^{n} \frac{1}{j} \leq 1 + \int_{1}^{n} \frac{dx}{x} = \ln n + 1,$$

a donju ogradu pravokutnicima "iznad" $1/x$

$$H_n = \sum_{j=1}^{n} \frac{1}{j} \geq \int_1^{n+1} \frac{dx}{x} = \ln(n+1) > \ln n.$$

Spajanjem ovih ocjena dobivamo

$$\ln n < \ln(n+1) \leq H_n \leq \ln n + 1,$$

odakle odmah slijedi $H_n \in \Theta(\log n)$, ali i mnogo preciznije informacije, poput

$$H_n \sim \ln n, \quad \text{ili} \quad H_n = \ln n + r_n, \quad r_n \in (0, 1].$$

Kad to uvrstimo u (2.5.11), dobivamo red veličine za $D(n)$

$$D(n) = 2H_n - \frac{4n}{n+1} = 2\ln n + 2r_n - \frac{4n}{n+1}.$$

Vodeći član je očito $2\ln n$, pa ovu relaciju možemo napisati u obliku

$$D(n) := 2\ln n - R(n),$$

gdje je $R(n)$ negativni "ostatak"

$$R(n) := \frac{4n}{n+1} - 2r_n = 4 - \frac{4}{n+1} - 2r_n.$$

Znamo da je $R(1) = 0$. Zbog $r_n \in (0, 1]$, odmah vidimo da je $2/3 \leq R(n) < 4$, za svaki $n > 1$, pa je $R(n) \in \Theta(1)$. Precizna ocjena za $D(n)$ je

$$2\ln n - 4 < D(n) \leq 2\ln n - \frac{2}{3}, \quad n > 1, \tag{2.5.12}$$

uz $D(0) = D(1) = 0$.

Na kraju, kad se vratimo na traženi niz $C(n)$, imamo

$$C(n) = (n+1)D(n) \sim 2n\ln n = \frac{2}{\lg e}\, n\lg n = (2\ln 2)\, n\lg n \approx 1.4\, n\lg n.$$

Naravno, relacija (2.5.12) daje i precizniju ocjena, ali to nije osobito bitno. Time smo dokazali sljedeći teorem.

Teorem 2.5.1. *Prosječni broj usporedbi ključeva $C(n)$ u algoritmu quicksort za sortiranje polja od n elemenata (uz pretpostavku slučajnog polaznog poretka elemenata i slučajnog biranja pivotnih ključeva) ima asimptotsko ponašanje*

$$C(n) \in \Theta(n \log n).$$

Dodatno, multiplikativna konstanta u vodećem članu je mala, tako da u bazi 2 vrijedi $C(n) \approx 1.4\, n \lg n$.

Isti rezultat o prosječnoj složenosti *quicksort* algoritma vrijedi i za ponešto drugačije realizacije algoritma (*findpivot*, *partition* i rekurzija), naravno, uz istu pretpostavku o jednakoj vjerojatnosti svih ulaznih poredaka. Pretpostavka o različitosti ključeva služi, u principu, samo za pojednostavljenje izvoda.

Što sve možemo promijeniti bez većih posljedica? Za konačni rezultat, ključne su polazne relacije (2.5.2), (2.5.3) i (2.5.4), pa treba pogledati koje promjene možemo tamo napraviti, i s kojim posljedicama. Usput, bilo bi još zgodno dobiti končnu rekurziju za $C(n)$ koja ima oblik poput (2.5.6), tako da ju možemo riješiti sličnom tehnikom.

Odmah je jasno da relacija (2.5.3)

$$C(n) = \sum_{j=1}^{n} p_{j,n} C(n, j)$$

vrijedi **uvijek**, bez obzira na to s kojom vjerojatnošću $p_{j,n}$ biramo j-ti po veličini pivotni ključ (između n mogućih ključeva). Bitno je samo da sigurno uvijek biramo neki pivotni ključ, tako da je

$$\sum_{j=1}^{n} p_{j,n} = 1. \tag{2.5.13}$$

Dakle, imamo ponešto slobode u izboru $p_{j,n}$.

Za $C(n, j)$, što je prosječan broj usporedbi ključeva u algoritmu, uz uvjet da je pivotni ključ j-ti po veličini, vrijedila je relacija (2.5.4)

$$C(n, j) = (n - 1) + C(j - 1) + C(n - j), \quad j = 1, \ldots, n.$$

Druga dva člana na desnoj strani dolaze zbog jednake vjerojatnosti svih poredaka u manjim poljima koja rekurzivno sortiramo. To je direktna posljedica osnovne pretpostavke o jednakoj vjerojatnosti svih polaznih poredaka. Jedino se duljina jednog od ta dva polja može povećti za 1, ako pivotni element uključimo u to potpolje (što nema puno smisla). Naravno, tada moramo zabraniti da to "manje" potpolje bude iste duljine n kao i polazno, da ne dobijemo beskonačnu rekurziju. Dakle, na desnoj strani se uvijek javljaju vrijednosti oblika $C(k)$, za $k < n$, a zbroj njihovih duljina bi mogao biti i jednak n. To neće bitno promijeniti stvar (vidjeti malo kasnije).

Prvi član $n - 1$ na desnoj strani je broj usporedbi ključeva u *partition*. Već smo rekli da on može biti i veći od $n - 1$. Zbog toga, na to mjesto možemo općenito staviti $P(n)$, što bi bila gornja ograda za broj usporedbi u *partition*, uz uvjet da je j-ti po veličini ključ pivotni, ili, naprosto, gornja ograda za broj usporedbi, neovisno o j. To vodi na gornju ogradu za $C(n, j)$, pa u (2.5.3) dobivamo gornju ogradu na $C(n)$. Ništa strašno, naprosto definiramo da je $C(n)$ gornja ograda za prosječan broj usporedbi i, umjesto $\leq$, i dalje svagdje pišemo jednakost. Dakle, u općem slučaju,

imamo

$$C(n,j) = P(n) + C(j-1) + C(n-j), \quad j = 1, \ldots, n, \tag{2.5.14}$$

Kad to uvrstimo u (2.5.3), zbrojimo i sredimo sumu po $C(j)$, bez obzira na vjerojatnosti $p_{j,n}$, dobivamo rekurziju oblika

$$\begin{aligned} C(n) &= \sum_{j=1}^{n} p_{j,n} \left(P(n) + C(j-1) + C(n-j)\right) \\ &= P(n) + \sum_{j=1}^{n} p_{j,n}(C(j-1) + C(n-j)) \\ &= P(n) + \sum_{j=0}^{n-1} q_{j,n} C(j), \end{aligned} \tag{2.5.15}$$

uz

$$q_{j,n} = p_{j+1,n} + p_{n-j,n}, \quad j = 0, \ldots, n-1. \tag{2.5.16}$$

Uočimo da iz (2.5.13) slijedi

$$\sum_{j=1}^{n-1} q_{j,n} = 2, \tag{2.5.17}$$

jer dva puta zbrajamo svaku vjerojatnost $p_{j,n}$.

Kad bismo dozvolili da zbroj duljina potpolja za rekurziju bude jednak n, a ne $n-1$, umjesto (2.5.14), imali bismo

$$C(n,j) = P(n) + C(j) + C(n-j), \quad j = 1, \ldots, n-1,$$

ili

$$C(n,j) = P(n) + C(j-1) + C(n-j+1), \quad j = 2, \ldots, n.$$

Uvrštavanjem u (2.5.3), nakon zbrajanja i sređivanja, opet dobivamo rekurzije oblika (2.5.15)

$$C(n) = P(n) + \sum_{j=0}^{n-1} q_{j,n} C(j),$$

uz malo drugačije veze između koeficijenata $q_{j,n}$ i vjerojatnosti $p_{j,n}$ od onih u (2.5.16). Nađite sami te relacije i pokažite da je $q_{j,n}$ opet zbroj dvije vjerojatnosti $p_{k,n}$, tako da i dalje vrijedi (2.5.17) za zbroj svih koeficijenata $q_{j,n}$.

Dosadašnja razmatranja dovode nas do sljedećeg zaključka. Uz pretpostavku jednake vjerojatnosti svih ulaznih poredaka ključeva, sve razumne implemetacije *quicksort* algoritma, bez obzira na detalje implementacije, vode na rekurziju oblika (2.5.15)

$$C(n) = P(n) + \sum_{j=0}^{n-1} q_{j,n} C(j),$$

gdje je:

- $C(n)$ prosječan broj usporedbi ključeva (ili donja, ili gornja ocjena za taj broj),
- $P(n)$ je broj usporedbi u *partition* dijelu algoritma (ili donja, ili gornja ocjena za taj broj), s tim da možemo uzeti da je uvijek $n-1 \leq P(n) \leq n+1$,
- $q_{j,n}$ je zbroj vjerojatnosti oblika (2.5.16), s tim da vrijedi (2.5.17), tj. zbroj svih koeficijenata $q_{j,n}$ je 2.

Naravno, ponašanje rješenja $C(n)$ ključno ovisi o tome kakvi su koeficijenti $q_{j,n}$, u ovisnosti o j i n.

Kao i obično, opće rješenje rekurzije (2.5.15) nije jednostavno naći, pa se zato ograničavamo na nekoliko karakterističnih slučajeva za koeficijente $q_{j,n}$.

Prije toga, trebamo jedan opći rezultat za rješavanje odgovarajućih rekurzivnih relacija.

Lema 2.5.1. *Bilo koja rekurzivna relacija za niz x_n oblika*

$$a_n x_{n+1} = b_n x_n + g_n, \quad n \geq m, \tag{2.5.18}$$

uz $a_n, b_n \neq 0$, i neki početni uvjet x_m, može se transformirati u "aditivnu" rekurziju oblika

$$y_{n+1} = y_n + h_n, \quad n \geq m,$$

gdje je

$$y_n = \frac{a_m \cdots a_{n-1}}{b_m \cdots b_{n-1}} x_n, \quad h_n = \frac{a_m \cdots a_{n-1}}{b_m \cdots b_{n-1} b_n} g_n,$$

s očitim rješenjem $y_{n+1} = h_n + \cdots + h_m + y_m$, uz $y_m = x_m$.

Dokaz:

Pomnožimo (2.5.18) sa "sumacijskim faktorom" $(a_m \cdots a_{n-1})/(b_m \cdots b_{n-1} b_n)$,

$$\frac{a_m \cdots a_{n-1}}{b_m \cdots b_{n-1} b_n} a_n x_{n+1} = \frac{a_m \cdots a_{n-1}}{b_m \cdots b_{n-1} b_n} b_n x_n + \frac{a_m \cdots a_{n-1}}{b_m \cdots b_{n-1} b_n} g_n, \quad n \geq m.$$

Koristeći definicije za y_n i h_n, ovu relaciju možemo napisati u obliku

$$y_{n+1} = y_n + h_n, \quad n \geq m.$$

Uz početni uvjet $y_m = x_m$ i $h_m = g_m/b_m$, rješenje ove rekurzije je

$$y_{n+1} = h_n + \cdots + h_m + x_m.$$

■

Iz ove leme odmah možemo dobiti i dva bitna rezultata, koja ćemo iskorstiti za daljnju analizu *quicksort* algoritma.

Korolar 2.5.1. *Pretpostavimo da za niz x_n vrijedi rekurzivna relacija oblika*

$$x_n = f_n + \frac{c}{n}\sum_{j=0}^{n-1} x_j, \quad n \geq m, \tag{2.5.19}$$

uz $c > 0$, i neki početni uvjet x_m.

Dokaz:

Pomnožimo (2.5.18 sa "sumacijskim faktorom" $(a_m \cdots a_{n-1})/(b_m \cdots b_{n-1}b_n)$,

$$\frac{a_m \cdots a_{n-1}}{b_m \cdots b_{n-1}}\, a_n x_{n+1} = \frac{a_m \cdots a_{n-1}}{b_m \cdots b_{n-1}}\, b_n x_n + \frac{a_m \cdots a_{n-1}}{b_m \cdots b_{n-1}}\, g_n, \quad n \geq m.$$

Koristeći definicije za y_n i h_n, ovu relaciju možemo napisati u obliku

$$b_n y_{n+1} = b_n y_n + b_n h_n, \quad n \geq m,$$

pa dijeljenjem s b_n dobivamo $y_{n+1} = y_n + h_n$, za $n \geq m$. Uz početni uvjet $y_m = x_m$ i $h_m = g_m/b_m$, rješenje ove rekurzije je

$$y_{n+1} = h_n + \cdots + h_m + x_m.$$

■

Postavlja se pitanje da li isti zaključak vrijedi i za drugačije izbore pivotnih ključeva. Odgovor je potvrdan, uz vrlo blage pretpostavke.

Kad to uvrstimo u (2.5.2), zajedno s $p_{j,n} = 1/n$, izlazi rekurzija za prosječni broj usporedbi ključeva $C(n)$

$$C(n) = \sum_{j=1}^{n} \frac{1}{n}\,(n - 1 + C(j-1) + C(n-j)) = (n-1) + \frac{1}{n}\sum_{j=1}^{n}(C(j-1) + C(n-j)). \tag{2.5.20}$$

$$C(n) = (n-1) + \frac{2}{n}\sum_{j=1}^{n} C(j-1). \tag{2.5.21}$$

Napomena 2.5.1. Izbor pivotnog ključa za koji vrijedi (2.5.2) je pravi, stvarno slučajni izbor pivotnog ključa u svakom koraku algoritma. Tada v s jednakom vjerojatnošću može biti bilo koji po veličini element u radnom polju. Napomena uz relaciju (2.4.8) pokazuje da i za taj izbor vrijedi ista ocjena prosječne složenosti.

Ovaj rezultat ne iznenađuje. Naime, u idealnom slučaju, oba bi potpolja imala uvijek iste duljine ili maksimalno za 1 različite duljine. Tj. za dobre izbore pivotnih

ključeva vrijedi da se maksimum $p_{j,n}$ dostiže za srednje vrijednosti $j \approx n/2$, a opada prema rubovima.

Obzirom na to da i u idealnom slučaju dobivamo isti oblik ocjene, možemo zaključiti da izbor većeg od prva dva ključa ne utječe bitno na konačni rezultat o prosječnoj složenosti. ■

3. Pohlepni algoritmi

Pohlepni algoritmi su, obično, jednostavnog oblika. Koriste se, uglavnom, za rješavanje problema optimizacije, na pr.

- nalaženje minimalnog razapinjućeg stabla grafa,
- nalaženje najkraćeg puta u grafu,
- nalaženje najboljeg redoslijeda izvođenja zadanih poslova.

Apstraktna formulacija takvog problema optimizacije standardno sadrži slijedeće karakteristične elemente.

- Skup C svih mogućih (raspoloživih ili dozvoljenih) **kandidata** — na pr. bridovi grafa ili poslovi koje treba obaviti.
- Funkciju koja provjerava da li određeni skup S izabranih kandidata daje (predstavlja) **rješenje** zadanog problema, ignorirajući (bar zasad) pitanje optimalnosti tog rješenja. Drugim riječima, funkcija *rješenje* daje odgovor da ili ne, tj. ima vrijednosti tipa *boolean*. Na pr. ova funkcija provjerava da li je neki skup bridova put između dva zadana vrha u grafu.
- Funkciju **cilja** (engl. objective function) koja daje vrijednost svakog rješenja problema. To je funkcija koju optimiziramo (minimiziramo ili maksimiziramo) — na pr. ukupna duljina razapinjućeg stabla, duljina puta između dva vrha, ukupno vrijeme potrebno za izvođenje poslova u zadanom poretku.

U ovim terminima, rješavanje problema optimizacije je traženje skupa kandidata koji predstavlja rješenje problema i optimizira vrijednost funkcije cilja. Obično pretpostavljamo da polazni problem ima barem jedno rješenje, tj. da postoji podskup $S \subseteq C$ koji je rješenje problema. Onda je problem optimizacije korektno definiran i, također, ima rješenje.

Vidimo da se rješenje problema svodi na traženje (izbor) pravog podskupa $S \subseteq C$. Broj svih podskupova skupa C je $2^{|C|}$, pa traženje pravog podskupa S može vrlo dugo trajati. Zbog toga se rješenje problema optimizacije ne traži ispitivanjem svih podskupova, ako se to može izbjeći. Obično se traženi skup S konstruira element po element, tj. dodavanjem (prihvaćanjem) i izbacivanjem kandidata, vodeći računa o funkciji cilja.

Pohlepni algoritam je pokušaj **brzog** nalaženja rješenja — izbora kandidata za traženi skup S. U takvom algoritmu imamo još neke dodatne karakteristične objekte.

- Skup onih kandidata koje smo već iskoristili.
- Funkciju koja provjerava da li je određeni skup kandidata **dopustiv** (engl. feasible), tj. da li je moguće taj skup dopuniti tako da dobijemo bar jedno rješenje problema (ne nužno optimalno).
- Funkciju **izbora** (selekcije) koja u svakom trenutku daje najperspektivnijeg, još neiskorištenog kandidata.

U ovim terminima rad pohlepnog algoritma možemo opisati na slijedeći način. Algoritam napreduje korak po korak.

- Na početku je skup S izabranih kandidata prazan ($S = \emptyset$).
- U svakom koraku, skupu S pokušavamo dodati **najboljeg** preostalog (neiskorištenog) kandidata, označimo ga s x. Prijedlog tog kandidata daje funkcija izbora, tj. x je kandidat na kojem se dotiže maksimum vrijednosti $izbor(x)$.
- Ako skup $S \cup \{x\}$ do tada izabranih kandidata, zajedno s upravo predloženim, nije više dopustiv, odbacujemo x. Tog odbačenog kandidata **nikada više ne provjeravamo**, tj. ne vraćamo ga među neiskorištene, već ga prebacujemo u odbačene kandidate. Odavde slijedi da svaki kandidat može biti provjeren samo jednom, što daje brzinu pohlepnom algoritmu.
- Ako je $S \cup \{x\}$ dopustiv, dodajemo kandidata x skupu S, tj. on postaje stvarno izabran (prihvaćen).
- Svaki put kad povećamo skup S, provjeravamo da li je novi S i rješenje problema. Ako je — stajemo, a u protivnom idemo na novi korak — izbor slijedećeg kandidata x, ako takav postoji.

Sasvim općenito, u svakom trenutku, skup C svih kandidata je disjunktna unija slijedeća 3 podskupa.

- C_0 = skup neiskorištenih kandidata,
- S = skup izabranih (prihvaćenih) kandidata,
- D = skup odbačenih kandidata.

Na početku je $C_0 = C$, a $S = D = \emptyset$. Funkcija *izbor* je definirana na skupu C_0.

U pohlepnom algoritmu, skup C_0 se stalno smanjuje, a skupovi S i D rastu. Za pojednostavljenje zapisa algoritma, skup C_0 pamtimo u skupu (varijabli) C, a skup D uopće ne pamtimo, jer te kandidate više ne provjeravamo.

Opći oblik pohlepnog algoritma je

```
procedure greedy ( C : skup ; var S : skup ; var OK : boolean ) ;
        { C je na početku skup svih raspoloživih kandidata. }
        { U skupu S akumuliramo rješenje problema. }
        { OK kaže da li je nađeni S rješenje. }
begin
   S := ∅ ;
   while not rješenje(S) and (C ≠ ∅) do
      begin
      x := element iz C koji maksimizira vrijednost izbor(x) ;
      C := C \ {x} ;
      if dopustiv(S ∪ {x}) then S := S ∪ {x} ;
      end ;
   OK := rješenje(S) ;
end ; { greedy }
```

Vidimo da pohlepni algoritam **ne** mora naći rješenje problema. Čak i ako ga nađe, to rješenje **ne** mora biti optimalno, jer pohlepni algoritam nigdje ne koristi funkciju cilja, već samo funkciju izbora. Da bi ovako nađeno rješenje bilo i optimalno rješenje, treba **dokazati** da pohlepni algoritam korektno rješava problem optimizacije.

Funkcija izbora je najčešće bazirana na funkciji cilja. Te dvije funkcije čak mogu biti identične. Međutim, kao što ćemo vidjeti, ima problema za koje postoji nekoliko prihvatljivih mogućnosti za funkciju izbora (a funkcija cilja je, naravno, ista). Katkad treba odabrati pravu funkciju izbora, da bi pohlepni algoritam radio korektno, u smislu da zaista rješava problem optimizacije. Osim toga, različtim funkcijama izbora možemo dobiti različte korektne algoritme za isti problem.

Lako se vidi zašto ove algoritme zovemo "pohlepnim". U svakom koraku algoritam uzima najbolji zalogaj koji može progutati, ne vodeći računa o budućnosti (ili prošlosti). On nikad ne mijenja odluku — kad je kandidat jednom prihvaćen u rješenje, on tamo i ostaje (bio dobar ili ne). Također, ako je jednom odbačen, to je zauvijek, nikada više neće biti provjeren.

Zbog toga su pohlepni algoritmi **brzi**, jer svakog kandidata provjeravamo najviše jednom (tj. broj provjera je najviše $|C|$). Za uzvrat, u nekim problemima ne dobivamo optimalno rješenje. Čak i tada, ovi algoritmi imaju veliku primjenu. Često se koriste za rješavanje dokazano teških problema, poput TSP, kao brza heuristika koja daje nekakvo rješenje, iako ne optimalno. To rješenje se može iskoristiti kao polazno u iterativnim metodama optimizacije, koje rade na principu poboljšavanja postojećeg rješenja.

U nastavku promatramo neke probleme za koje pohlepni algoritmi daju optimalno rješenje.

3.1. Minimalno razapinjuće stablo

Neka je $G = (V, E)$ povezan, neusmjeren graf sa skupom vrhova V i skupom bridova (grana) E. Svaki brid $e \in E$ ima nenegativnu duljinu $\ell(e)$, tj. zadana je funkcija $\ell : E \to \mathbb{R}_0^+$ duljine ili cijene bridova.

Problem MST (Minimal Spanning Tree) — minimalno razapinjuće stablo.

Treba naći podskup T bridova grafa G tako da svi vrhovi iz V ostanu povezani samo bridovima iz T i da je zbroj duljina bridova u T minimalan, po svim takvim skupovima T. ■

Lako se vidi da je podgraf (V, T) grafa G **stablo**, tj. povezan graf bez ciklusa. Taj graf zovemo minimalno razapinjuće stablo grafa G.

Ovaj problem ima mnogo primjena. Na primjer, ako vrhovi od G predstavljaju gradove, a cijena brida $\{a, b\}$ je cijena izgradnje ceste između gradova a i b, onda minimalno razapinjuće stablo grafa G kaže kako treba projektirati najjeftiniji mogući sustav cesta koji povezuje sve zadane gradove.

Konstruirat ćemo 2 pohlepna algoritma za rješenje ovog problema. U ranijoj terminologiji za pohlepne algoritme, imamo slijedeća značenja.

- Kandidati su bridovi grafa, tj. $C = E$.
- Skup S izabranih bridova je **rješenje** problema, ako bridovi iz S tvore razapinjuće stablo za G, tj. vežu sve vrhove iz V.
- Skup S bridova je **dopustiv**, ako ne sadrži ciklus. Uočimo da ne tražimo da taj skup daje stablo na V, tj. da (V, S) bude povezan graf. Dozvoljavamo da taj graf bude i šuma — nepovezan, a svaka komponenta povezanosti je stablo. Upravo na tom mjestu će se algoritmi razlikovati.
- Funkcija **cilja** je očita — zbroj duljina svih bridova u rješenju (razapinjućem stablu) i treba ju minimizirati.

Funkciju izbora specificiramo kasnije, jer ona razlikuje algoritme.

Uvedimo još dva termina potrebna za dokaz korektnosti pohlepnih algoritama. Dopustiv skup bridova je **obećavajući**, ako se može dopuniti do optimalnog rješenja. Posebno, prazan skup je obećavajući, čim je G povezan. Brid e **dira** dani skup vrhova W, ako točno jedan kraj brida e pripada skupu W. Slijedeća lema omogućava konstrukciju algoritama i dokaz njihove korektnosti.

Lema 3.1.1.

Neka je $G = (V, E)$ povezan, neusmjeren graf i neka je $\ell : E \to \mathbb{R}_0^+$ funkcija cijene (duljine) bridova. Neka je $W \subset V$ pravi podskup skupa vrhova i neka je $T \subseteq E$ obećavajući skup bridova, takav da niti jedan brid iz T ne dira W. Neka je

$e' \in E$ **najkraći** *brid koji dira* W *(ili bilo koji takav, ako ima više najkraćih). Tada je i skup bridova* $T' = T \cup \{e'\}$ *obećavajući.*

Dokaz:

Po pretpostavci, T je obećavajući. Zbog toga, postoji skup bridova $U \subseteq E$, takav da je podgraf (V, U) minimalno razapinjuće stablo grafa G i vrijedi $T \subseteq U$. Ako je $e' \in U$, onda je $T' \subseteq U$ i dokaz je gotov.

Pretpostavimo da je $e' \notin U$ i neka je $U_1 = U \cup \{e'\}$. Podgraf (V, U) je stablo, pa (V, U_1) mora sadržavati točno jedan ciklus. U tom ciklusu, jer brid e' dira W, mora postojati barem još jedan brid $e \neq e'$ koji, također, dira W. Naime, zbog $W \neq V$, taj ciklus mora "ući" i "izaći" iz skupa W s barem 2 brida — jedan je e', a drugi e. U protivnom se ciklus ne bi mogao zatvoriti. Brid e ne pripada skupu T, jer e dira W, a niti jedan brid iz T ne dira W.

Ako izbacimo brid e iz U_1, taj ciklus se prekida. Na taj način dobivamo skup bridova $U' = U_1 \setminus \{e\}$, s tim da je (V, U') razapinjuće stablo grafa G. Zbog $e \notin T$, vrijedi $T \subseteq U'$, pa iz $e' \in U'$ slijedi $T' \subseteq U'$.

Na kraju, po pretpostavci je $\ell(e') \leq \ell(e)$, jer je e' najkraći brid koji dira W. Uočimo da U' dobivamo iz U zamjenom brida e, bridom e'. Tada je

$$\sum_{u \in U'} \ell(u) - \sum_{u \in U} \ell(u) = \ell(e') - \ell(e),$$

odakle slijedi

$$\sum_{u \in U'} \ell(u) \leq \sum_{u \in U} \ell(u).$$

No, (V, U) je minimalno razapinjuće stablo za G. Zbog toga, u prethodnoj relaciji mora vrijediti jednakost, a (V, U') je, također, minimalno razapinjuće stablo za G. Iz $T' \subseteq U'$ slijedi da je T' obećavajući skup bridova. ■

Osnovna razlika između raznih pohlepnih algoritama za rješavanje problema MST je u poretku izbora bridova. Primov algoritam gradi stablo koje raste, a Kruskalov pohlepno slaže šumu koju postepeno spaja u stablo. Oba algoritma paze da dodavanjem brida ne zatvore ciklus.

3.1.1. Primov algoritam

Autori ovog algoritma su Prim (1957.g.) i Dijkstra (1959.g.), a tek kasnije je otkriveno da je Jarník još 1930. godine formulirao sličan postupak.

Algoritam počinje u bilo kom vrhu — korijenu stabla. Na početku je skup W jednočlan, s bilo kojim vrhom kao elementom, a skup T je prazan. U svakom koraku, povećavamo do tada nađeno stablo (W, T), tako da skupu T dodajemo novi

brid koji dira W, a skupu W dodajemo vrh tog brida koji nije bio u W. Na taj način imamo stablo koje raste, sve dok ne poveže sve vrhove iz V.

Funkcija izbora u Primovom algoritmu ima slijedeći oblik. U pojedinom koraku, za trenutni graf (W, T), funkcija izbora bira najkraći brid $e = \{u, v\}$, takav da je

$$u \in V \setminus W \quad \text{i} \quad v \in W.$$

Nakon toga, algoritam dodaje vrh u skupu W i brid $\{u, v\}$ skupu T. Očito je da, u svakom trenutku, bridovi u skupu T tvore minimalno razapinjuće stablo grafa (W, E'), gdje je E' restrikcija od E na skup vrhova $W \subseteq V$. Algoritam se nastavlja sve dok je $W \neq V$. Neformalni zapis algoritma je

```
procedure Prim (
        G = (V, E) : graf ;
        ℓ : E → ℝ₀⁺ : funkcija ;
        var T : skup_bridova ) ;

begin
    { Inicijalizacija. }
  T := ∅ ;
  W := bilo koji element iz V ;

    { Pohlepna petlja. }
  while W ≠ V do
    begin
    nađi brid e = {u, v} najmanje duljine ℓ(e),
          takav da je u ∈ V \ W i v ∈ W ;
    T := T ∪ {e} ;
    W := W ∪ {u} ;
    end ; { while }
end ; { Prim }
```

Teorem 3.1.1.

Primov algoritam radi korektno, tj. za povezan, neusmjeren graf G s funkcijom cijene ℓ, algoritam vraća skup bridova T takav da je (V, T) minimalno razapinjuće stablo grafa G.

Dokaz:

Direktno iz leme 3.1.1., indukcijom po broju vrhova u skupu W. ■

Uočimo da se while petlja izvršava točno $|V| - 1$ puta, jer svaki prolaz dodaje po jedan vrh skupu W.

Napomena 3.1.1. U ovom obliku algoritma, izabrani brid uvijek prihvaćamo, tj. nema odbacivanja i zato se while petlja izvršava točno $|V| - 1$ puta.

Kad bi funkcija izbora dala samo brid najmanje duljine među preostalim bridovima (što je logičnija realizacija), imali bismo mogućnost odbacivanja. Funkcija *dopustiv* treba, prema lemi 3.1.1., prihvatiti samo one bridove $e = \{u, v\}$ za koje je $u \in V \setminus W$ i $v \in W$ (ili obratno). Tada, dodavanjem brida e skupu T, dobivamo minimalno razapinjuće stablo za podgraf od G s vrhovima u $W \cup \{u\}$. ■

Napomena 3.1.2. Graf G s funkcijom ℓ može imati više minimalnih razapinjućih stabala. U Primovom algoritmu se ta mogućnost ogleda tako, da može biti više bridova e najmanje duljine $\ell(e)$, koji diraju trenutni skup vrhova W. ■

Za potpunu specifikaciju algoritma, treba odabrati strukture podataka za prikaz objekata, koje omogućavaju efikasno izvođenje izbora.

Jednu jednostavnu implementaciju Primovog algoritma dobivamo na slijedeći način. Pretpostavimo da su vrhovi grafa G numerirani brojevima od 1 do n, tj. da je $V = \{1, \ldots, n\}$. Bridove grafa i pripadne duljine zadajemo simetričnom matricom L reda n, s elementima

$$L(u, v) = \begin{cases} \ell(\{u, v\}) , & \text{ako je } \{u, v\} \in E \\ \infty , & \text{inače,} \end{cases}$$

uz pretpostavku da su duljine $\ell(\{u, v\})$ bridova nenegativne.

Za realizaciju funkcije izbora, koristimo još 2 polja: *nearest* (najbliži) i *mindist* (najmanja udaljenost), indeksirana vrhovima iz V. Za vrh $i \in V \setminus W$, stavljamo

$$\begin{aligned} \mathit{nearest}[i] &= \text{vrh iz } W \text{ najbliži vrhu } i, \\ \mathit{mindist}[i] &= \text{udaljenost vrhova } i, \mathit{nearest}[i]. \end{aligned}$$

Za vrh $i \in W$ stavljamo $\mathit{mindist}[i] = -1$, tako da skup W ne treba posebno pamtiti. U inicijalizaciji, proizvoljno uzimamo $W = \{1\}$, a elemente $\mathit{nearest}[1]$ i $\mathit{mindist}[1]$ uopće ne koristimo. Na kraju, zbog $|V| = n$, znamo da konačni skup T ima točno $n-1$ bridova (zbog povezanosti grafa G), pa while petlju realiziramo kao for petlju.

Jedini preostali problem je reprezentacija skupa bridova T. Zbog $|T| = n - 1$, za T možemo koristiti i polje bridova. Svaki brid je dvočlani skup, kojeg možemo reprezentirati na nekoliko načina — "pravim" skupom (u Pascalu), poljem od 2 vrha ili recordom (zapisom) s 2 vrha. Za T možemo koristiti i 2 polja vrhova — prvo za jedan vrh brida, a drugo za drugi vrh. Točna reprezentacija za T ovisi o konkretnoj primjeni algoritma i nećemo ju detaljno specificirati.

Također, nećemo vratiti ukupnu duljinu bridova iz T. Dodatak toga u algoritam je trivijalan.

Algoritam 3.1.1. (Primov algoritam za MST)

```
procedure Prim (
        n : integer ;
        L : matrix ; { koristimo L[1..n, 1..n] }
        var T : skup_bridova ) ;
begin
    { Inicijalizacija W := [1], T = [ ]. }
  T := ∅ ; { T akumulira bridove MST. }
  for j := 2 to n do  { samo vrh 1 je u skupu W. }
    begin
    mindist[j] := L[j, 1] ;
    nearest[j] := 1 ;
    end ; { for j }
    { Pohlepna petlja. }
  for i := 1 to n − 1 do  { T sadrži n − 1 bridova. }
    begin
    min := ∞ ;
    for j := 2 to n do
      if (mindist[j] ≥ 0) and (mindist[j] < min) then
        begin
        min := mindist[j] ;
        k := j ;
        end ; { if, for j }
      { k je definiran ako i samo ako je G povezan, inače ne! }
    T := T ∪ { {k, nearest[k]} } ;
    mindist[k] := −1 ; { dodaj vrh k skupu W. }
      { Popravi polja nearest, mindist za novi k u skupu W. }
    if i < n − 1 then
      for j := 2 to n do
        if L[k, j] < mindist[j] then  { j ∉ W }
          begin
          mindist[j] := L[k, j] ;
          nearest[j] := k ;
          end ; { if, for j, if }
    end ; { for i — pohlepna petlja }
end ; { Prim }
```

Zadatak 3.1.1. Beskonačne vrijednosti, obično, ne možemo koristiti u programu. Problem se javlja na dva mjesta: u matrici L i inicijalizaciji varijable *min*. Kako treba popraviti algoritam, ako to želimo izbjeći? ■

Zadatak 3.1.2. Povezanost grafa G je bitna pretpostavka za korektnost algoritma. Ovaj algoritam to ne provjerava. Kako ga treba popraviti da uredno javlja da je G nepovezan? Koje značenje tada ima skup T kojeg vraća algoritam? ■

3.1.2. Složenost Primovog algoritma

Za graf G s n vrhova, pohlepna (vanjska) petlja se izvršava točno $n-1$ puta. U našoj implementaciji, svaki korak — prolaz kroz tu petlju, sadrži još dvije petlje s po $n-1$ prolaza, a sve ostale operacije su elementarne i njihovo trajanje ne ovisi o n. Dakle, svaki korak ima trajanje $O(n)$, a u našem programu vrijedi $\Theta(n)$.

Zaključujemo da za vremensku složenost Primovog algoritma vrijedi

$$T(n) = O(n^2), \tag{3.1.1}$$

a u našem programu je $T(n) = \Theta(n^2)$, gdje je $n = |V|$. Za empirijsko računanje složenosti, pogodno je uzeti kvadratni polinom kao oblik funkcije za $T(n)$.

Preciznija procjena trajanja, naravno, ovisi o tome koliko bridova ima graf G, jer to bitno utječe na broj prolaza kroz "then" blokove u if–ovima. Zbog toga što je G povezan i neusmjeren, očito je

$$n - 1 \leq |E| \leq \frac{1}{2}\, n(n-1).$$

Ako je G "gust" graf, tj. $|E|$ je blizu gornje granice, broj prolaza kroz "then" blokove je velik.

Ako je G "rijedak", tj. blizak stablu, takvih prolaza je vrlo malo. Tada je pogodnije polja *nearest* i *mindist* reprezentirati dinamički, na pr. listom. Posebno to vrijedi za *mindist*, tako da ne pamtimo vrijednosti -1, koje odogovaraju već povezanim vrhovima. Ova modifikacija bitno skraćuje trajanje obje unutarnje petlje.

Detaljnija analiza ovisi o detaljima strukture grafa G — broju bridova, stupnjevima vrhova i sl.

3.1.3. Kruskalov algoritam

Ovaj algoritam formulirao je Kruskal 1956. godine. Kao i kod Primovog algoritma, u skupu bridova T akumuliramo minimalno razapinjuće stablo grafa G.

Međutim, Kruskalov algoritam stalno radi s podgrafom oblika (V, T) grafa G, tj. sa svim vrhovima od G, pa nema skupa W i dodavanja vrhova tom skupu.

Na početku je $T = \emptyset$, a u svakom koraku skupu T pokušavamo dodati neki brid.

U svakom trenutku Kruskalovog algoritma, podgraf (V, T) se sastoji od nekog broja povezanih komponenti, ali sam ne mora biti povezan. Bridovi iz T, sadržani u nekoj komponenti povezanosti grafa (V, T), čine minimalno razapinjuće stablo za tu komponentu (tj. vežu sve vrhove u toj komponenti i to najmanjom duljinom). Drugim riječima, T je obećavajući skup bridova.

Na početku, kada je T prazan, svaki vrh grafa G je zasebna, trivijalna komponenta povezanosti. Takvih komponenti je $|V| = n$. Nakon toga, algoritam spaja po dvije disjunktne komponente povezanosti u jednu, veću komponentu povezanosti, dodavanjem brida između njih. Zbog povezanosti grafa G, algoritam završava u trenutku kad dobijemo samo jednu komponentu povezanosti. Tada je (V, T) minimalno razapinjuće stablo za G.

Ostaje još opisati kako biramo bridove za spajanje komponenti. Na početku, sortiramo sve bridove iz E, uzlazno po duljini. U svakom koraku biramo najkraći preostali brid.

Ako on spaja dva vrha u različitim (disjunktnim) komponentama povezanosti, onda ga dodajemo skupu T. Taj novi brid spaja dvije komponente povezanosti u jednu. Prema lemi 3.1.1., bridovi iz T čine minimalno razapinjuće stablo za tu novu komponentu.

U protivnom, brid odbacujemo, jer spaja dva vrha iz iste komponente povezanosti. Po pretpostavci, bridovi iz T već tvore minimalno razapinjuće stablo za tu komponentu, pa bi dodavanje tog brida u skup T zatvorilo ciklus. Tada bi T prestao biti obećavajući.

Teorem 3.1.2.

Neka je G povezan, neusmjeren graf s funkcijom cijene ℓ. Kruskalov algoritam nalazi skup bridova T takav da je (V, T) minimalno razapinjuće stablo za G.

Dokaz:

Dokaz smo već proveli, opisujući algoritam. Formalno govoreći, koristimo indukciju po broju izabranih bridova do tog trena, pozivanjem na lemu 3.1.1. u svakom koraku. ■

Za razliku od Primovog algoritma, pohlepno biramo bridove, ne vodeći računa o tome da li su oni vezani s prethodno odabranim bridovima ili ne. Pazimo samo na to da ne zatvorimo ciklus. Tj. graf (V, T) u Kruskalovom algoritmu je šuma (nepovezan), a stajemo kad postane stablo (povezan).

Zadatak 3.1.3. Graf može imati više minimalnih razapinjućih stabala. Kako se to odražava u Kruskalovom algoritmu? ■

Zadatak 3.1.4. Što se događa u ovom algoritmu, ako G nije povezan? ■

U Kruskalovom algoritmu moramo sortirati bridove po duljini. Zbog toga je pogodnije graf prikazati nizom (poljem, listom) bridova s pridruženim duljinama, a ne matricom udaljenosti.

Ostatak algoritma sastoji se iz operacija na komponentama povezanosti grafa (V, T). Svaku komponentu povezanosti reprezentiramo skupom vrhova koji pripadaju toj komponenti. Bridove iz T ne treba dodatno pamtiti (osim, naravno, u T). Na takvoj strukturi disjunktnih skupova vrhova trebamo slijedeće dvije operacije.

- *find*(x) — "nađi skup od x", nađi skup vrhova – komponentu povezanosti u kojoj se nalazi vrh x,
- *merge*(A, B) — "spoji A i B", spoji dva disjunktna skupa (komponente povezanosti) A i B u jedan skup (unija disjunktnih skupova).

Potrebna apstraktna struktura (ili tip) podataka su tzv. disjnktni skupovi — particije, nad određenim skupom (univerzumom). U našem slučaju, univerzum je skup V svih vrhova grafa G, a komponente povezanosti grafa (V, T) su particija tog skupa.

Ovu strukturu i algoritme za efikasnu implementaciju operacija *find* i *merge* opisujemo nakon zapisa Kruskalovog algoritma. U tom zapisu, koristimo ove dvije operacije kao potprograme, a kasnije ćemo ih detaljno specificirati. Osim toga, umjesto $n = |V|$, koristimo oznaku $N = |V|$.

```
procedure Kruskal (
        G = (V, E) : graf ;
        ℓ : E → ℝ₀⁺ : funkcija ;
        var T : skup_bridova ) ;
begin
    { Inicijalizacija. }
  sortiraj E uzlazno po duljini ℓ bridova ;
  T := ∅ ;
  N := card(V) ; { = |V| }
  inicijaliziraj N (disjunktnih) skupova tako da svaki sadrži
        točno po jedan (različit) vrh iz V ;
    { Pohlepna petlja. }
  repeat
    {u, v} := najkraći preostali brid (do tada neobrađen) ;
```

```
        { Nađi komponente povezanosti kojima pripadaju vrhovi u i v. }
      ucomp := find(u) ;
      vcomp := find(v) ;
      if ucomp ≠ vcomp then   { prihvati brid {u, v} }
        begin
          { Spoji različite komponente povezanosti u jednu. }
        merge(ucomp, vcomp) ;
        T := T ∪ { {u, v} } ;
        end
      else
        odbaci brid {u, v} ;   { jer bi zatvorio ciklus }
    until card(T) = N − 1 ;
  end ;  { Kruskal }
```

U ovom algoritmu može doći do odbacivanja brida, pa repeat ne možmo pretvoriti u for petlju. Uvjet zaustavljanja $card(T) = N - 1$, odgovara tome da minimalno razapinjuće stablo u povezanom grafu s N vrhova, ima točno $N - 1$ bridova. Ekvivalentno je da je preostala samo jedna komponenta povezanosti.

Zadatak 3.1.5. Ako graf G nije povezan, kako treba modificirati ovaj uvjet da algoritam daje smisleni rezultat? Što je tada rezultat? ■

4. Brza diskretna Fourierova transformacija

Diskretna Fourierova transformacija (skraćeno DFT) ima vrlo veliko područje primjene, posebno u obradi signala i otklanjanju šumova. Upravo zato je **brza** realizacija diskretne Fourierove transformacije (tzv. Fast Fourier Transform, skraćeno FFT) jedan od najvažnijih i najčešće korištenih algoritama uopće. Na njemu su bazirani i mnogi drugi brzi algoritmi u aritmetici i algebri.

Osnovna primjena FFT-a u ovom poglavlju bit će brzi algoritam za množenje kompleksnih polinoma. Zbog toga su definicija DFT-a i sve oznake prilagođene tom cilju.

4.1. Problemi evaluacije i interpolacije

Neka je A bilo koji kompleksni polinom stupnja najviše $n-1$, gdje je $n \in \mathbb{N}$ prirodan broj. Taj polinom možemo napisati u obliku

$$A(z) = \sum_{j=0}^{n-1} a_j z^j,$$

gdje su $a_0, \ldots, a_{n-1} \in \mathbb{C}$ koeficijenti tog polinoma u standardnoj bazi vektorskog prostora svih polinoma stupnja najviše $n-1$ nad poljem kompleksnih brojeva. Dimenzija tog vektorskog prostora je baš n, pa zato kažemo da je A polinom **reda** n. Time izbjegavamo eksplicitno navođenje stupnja (koji može biti i manji od $n-1$).

Prirodni izomorfizam ovog vektorskog prostora i prostora $\mathbb{C}^n$ pokazuje da je polinom A jednoznačno određen vektorom koeficijenata

$$a = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{C}^n.$$

Pretpostavimo sad da smo izabrali točno n različitih kompleksnih točaka

$$z_0, z_1, \ldots, z_{n-1} \in \mathbb{C}$$

i želimo izračunati vrijednosti polinoma A u **svim** tim točkama. Uz oznaku

$$y_k = A(z_k), \quad k = 0, \ldots, n-1,$$

polinomu A, odnosno vektoru koeficijenata a, pridružili smo novi vektor

$$y = (y_0, y_1, \ldots, y_{n-1}) \in \mathbb{C}^n$$

(iz istog n-dimenzionalnog prostora $\mathbb{C}^n$), koji sadrži vrijednosti zadanog polinoma u zadanim točkama.

Ovo pridruživanje $a \mapsto y$ potpuno je zadano izborom točaka $z_0, \ldots, z_{n-1}$ (i to, očito, jednoznačno). Kad raspišemo komponente vektora y u obliku

$$y_k = A(z_k) = \sum_{j=0}^{n-1} a_j z_k^j, \quad k = 0, \ldots, n-1,$$

vidimo da je svaki y_k **linearna kombinacija** koeficijenata a_j, što možemo matrično zapisati u obliku

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} z_0^0 & z_0^1 & \cdots & z_0^{n-1} \\ z_1^0 & z_1^1 & \cdots & z_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n-1}^0 & z_{n-1}^1 & \cdots & z_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}.$$

Matricu ovog sustava označavamo s

$$V_n(z_0, \ldots, z_{n-1}) := \begin{bmatrix} z_0^0 & z_0^1 & \cdots & z_0^{n-1} \\ z_1^0 & z_1^1 & \cdots & z_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n-1}^0 & z_{n-1}^1 & \cdots & z_{n-1}^{n-1} \end{bmatrix}.$$

Odmah vidimo da je ova matrica zapravo Vandermondeova matrica izabranog vektora točaka $z_0, \ldots, z_{n-1}$, pa iz linearne algebre znamo razna svojstva ovih matrica (koja će nam malo kasnije trebati).

Za početak, uočimo da je pridruživanje $a \mapsto y$ **linearna** transformacija (ili operator) iz $\mathbb{C}^n$ u $\mathbb{C}^n$, a matrica tog operatora je upravo $V_n(z_0, \ldots, z_{n-1})$.

Algoritamski gledano, za zadani vektor točaka $z_0, \ldots, z_{n-1}$, zanimljivo je gledati **dva** problema.

4.1.1. Problem evaluacije (izračunavanja)

Prvi problem je tzv. **problem evaluacije** ili **izračunavanja**, u kojem za zadani vektor a treba izračunati pripadni y. Dakle, točke smatramo fiksnima, ulaz je

vektor a koeficijenata polinoma, a izlaz vektor y vrijednosti tog polinoma u zadanim točkama. Sasvim općenito, ovaj problem se svodi na **množenje matrice i vektora**

$$y = V_n(z_0, \ldots, z_{n-1}) \cdot a,$$

kad uređene n-torke iz $\mathbb{C}^n$ zapišemo matrično kao stupce. Standardnim algoritmom (skalarnim produktima), ovaj problem možemo riješiti u $\Theta(n^2)$ kompleksnih aritmetičkih operacija. Preciznije, treba nam $n(n-1)$ množenja i isto toliko zbrajanja.

Alternativni algoritam dobivamo tako da "zaboravimo" na matrice i vektore, i cijeli problem interpretiramo kao n nezavisnih računanja vrijednosti polinoma u točki (ulaz su polinom A i točka z_k). Znamo da je Hornerova shema optimalan algoritam za **jedan** takav problem, a složenost je (najviše) $n-1$ množenja i isto toliko zbrajanja (jer je polinom stupnja najviše $n-1$). Kad to ponovimo n puta, dobivamo istu složenost kao i za množenje matrice i vektora, tj. **kvadratnu** u n.

Uočite da optimalnost Hornerove sheme za jednu točku **ne povlači** optimalnost primjene istog algoritma n puta za n točaka, jer točke mogu biti specijalne, a i polinom možemo ponešto "pripremiti" za računanje u puno točaka. Dakle, bar za posebne izbore točaka, možemo se nadati da postoje i brži algoritmi za problem evaluacije.

4.1.2. Problem interpolacije

Drugi problem je obrat (ili inverz) prvoga. Za zadani vektor y vrijednosti polinoma u zadanim točkama, treba naći vektor a koeficijenata tog polinoma. Drugim riječima, treba naći polinom A koji u zadanim točkama ima zadane vrijednosti, pa je prirodno da se ovaj problem zove **problem interpolacije** (prisjetite se interpolacije polinomima iz numeričke matematike).

Matrično–vektorski gledano, treba riješiti linearni sustav

$$y = V_n(z_0, \ldots, z_{n-1}) \cdot a,$$

jer a tražimo, a y znamo.

Pogledajmo prvo kad ovaj sustav ima jedinstveno rješenje (tako da ga ima smisla računati). Matrica sustava je Vandermondeova, pa je pripadna determinanta, naravno, Vandermodeova, a za nju znamo da vrijedi

$$\det(V_n(z_0, \ldots, z_{n-1})) = \prod_{0 \le \ell < k \le n-1} (z_k - z_\ell).$$

Vidimo da je pretpostavka o različitosti izabranih točaka, $z_k \neq z_\ell$, za $k \neq \ell$, nužan i dovoljan uvjet da matrica sustava bude regularna, tj. da sustav ima jedinstveno rješenje.

Sad kad smo to ustanovili, možemo raspravljati o složenosti računanja rješenja ovog linearnog sustava. Sasvim općenito, možemo koristiti Gaussove eliminacije (ili LU faktorizaciju), kao da je matrica sustava bilo koja regularna matrica. Taj postupak ima složenost $\Theta(n^3)$ kompleksnih aritmetičkih operacija, tj. **kubnu** u n, što je bitno sporije od prvog problema.

Međutim, naš sustav ima matricu vrlo specijalne strukture (Vandermondeovu), pa očekujemo da se rješenje može naći i brže. Zaista, nije teško konstruirati algoritme koji imaju **kvadratnu** složenost u n (kao i za prvi problem).

Zadatak 4.1.1. U numeričkoj matematici standardno se rade Lagrangeova i Newtonova formula za interpolaciju. Međutim, pripadni oblici interpolacionog polinoma koriste prikaze polinoma u malo drugačijim bazama od standardne. Sastavite pripadne algoritme za nalaženje koeficijenata polinoma u odgovrajućima bazama i analizirajte njihovu složenost. Nađite pripadne baze i sastavite algoritme za prijelaz iz tih baza u standardnu. Kolika je složenost tako dobivenih "kombiniranih" algoritama za problem interpolacije u standardnoj bazi? ■

Zadatak 4.1.2. Znamo da je "**zabranjeno**" koristiti Cramerovo pravilo kao algoritam za rješenje općeg linearnog sustava, bez obzira na to da li potrebne determinante računamo rekurzivno (Laplaceov razvoj) ili svođenjem na trokutastu formu (kolike su pripadne složenosti?).

Da li isto vrijedi i za specijalne sustave s Vandermondeovom matricom, jer se determinanta Vandermondeove matrice može mnogo brže izračunati? Pokušajte sastaviti što je moguće efikasniji algoritam za problem interpolacije na bazi Cramerovog pravila, pazeći na efikasno računanje potrebnih determinanti, tako da pamtite "zajedničke faktore" (slično kao kod efikasne realizacije Lagrangeove interpolacije). Kolika je najmanja složenost koju možete postići? ■

Na kraju ove opće priče o problemu interpolacije, kao i kod prvog problema, očekujemo da za posebne izbore točaka postoje i bitno efikasniji algoritmi interpolacije od kvadratnih u n. Jedan takav izbor točaka su tzv. n-ti korijeni iz jedinice.

4.2. Definicija diskretne Fourierove transformacije

Neka je n prirodan broj. Osnovni ili glavni n-ti korijen iz jedinice je kompleksan broj

$$\omega_n := e^{2\pi i/n} = \cos\left(\frac{2\pi}{n}\right) + i \sin\left(\frac{2\pi}{n}\right). \tag{4.2.1}$$

Očito je $\omega_n^n = e^{2\pi i} = 1$, što opravdava naziv. Svi n-ti korijeni iz jedinice su točke

$$\omega_n^k = e^{2k\pi i/n} = \cos\left(\frac{2k\pi}{n}\right) + i\sin\left(\frac{2k\pi}{n}\right), \quad k = 0, \ldots, n-1.$$

Ključno svojstvo ovih točaka je da one čine **grupu** obzirom na **množenje** kompleksnih brojeva (vidjeti malo niže). Raspoređene su u vrhovima pravilnog n-terokuta na jediničnoj kružnici u kompleksnoj ravnini, s tim da je jedan od vrhova smješten na realnoj osi u točki 1.

Definicija 4.2.1. Diskretna Fourierova transformacija *reda n je pridruživanje $a \mapsto y$ za vektor točaka svih n-tih korijena iz jedinice*

$$z_k = \omega_n^k, \quad k = 0, \ldots, n-1.$$

Pripadnu transformaciju označavamo s $y = \mathrm{DFT}_n(a)$, a pripadnu Vandermondeovu matricu s $V_n := V_n(\omega_n^0, \ldots, \omega_n^{n-1})$.

Inverzna diskretna Fourierova transformacija *reda n je obratno pridruživanje $y \mapsto a$ za vektor svih n-tih korijena iz jedinice. Pripadnu transformaciju označavamo s $a = \mathrm{DFT}_n^{-1}(y)$.*

Računanje diskretne Fourierove transformacije reda n za zadani vektor a je problem evaluacije ili izračunavanja u svim n-tim korijenima iz jedinice. Uočite da po definiciji vrijedi

$$y = \mathrm{DFT}_n(a) \iff y = V_n a.$$

Analogno, računanje inverzne transformacije reda n za zadani vektor y je problem interpolacije u svim n-tim korijenima iz jedinice, što možemo zapisati kao

$$a = \mathrm{DFT}_n^{-1}(y) \iff y = V_n a \iff a = V_n^{-1} y.$$

Malo kasnije ćemo pokazati da se V_n^{-1} može vrlo jednostavno izraziti preko V_n, što nam omogućava da problem interpolacije svedemo na problem evaluacije s malo drugačijom Vandermondeovom matricom. Praktično, to znači da za oba problema možemo koristiti **isti** algoritam (uz minimalne modifikacije).

Za početak, navedimo osnovna svojstva n-tih korijena iz jedinice koja ćemo kasnije trebati.

Teorem 4.2.1. *Za svaki $n \in \mathbb{N}$, skup $\{\omega_n^k \mid k = 0, \ldots, n-1\}$ svih n-tih korijena iz jedinice je grupa obzirom na* **množenje**. *Ta grupa je* **izomorfna** *aditivnoj grupi $(\mathbb{Z}_n, +_n)$ ostataka modulo n.*

Dokaz:

Ova tvrdnja se obično dokazuje kao zadatak u algebri grupa. Iako je dokaz

jednostavan, prođimo kroz osnovne korake zato da ponovimo "formule" za operacije s n-tim korijenima iz jedinice.

Prvo uočimo da vrijedi $\omega_n^n = \omega_n^0 = 1$, što je neutral obzirom na množenje. Zbog toga, za svaki cijeli broj $k \in \mathbb{Z}$ vrijedi

$$\omega_n^k = \omega_n^{k \bmod n},$$

čime dobivamo vezu između eksponenata i ostataka modulo n.

Nadalje, kod množenja potencija iste baze, eksponenti se zbrajaju,

$$\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n},$$

odakle slijede sva potrebna svojstva grupe i spomenuti izomorfizam.

Na kraju, za inverz osnovnog korijena iz jedinice vrijedi

$$\omega_n^{-1} = \omega_n^{n-1} = \overline{\omega}_n,$$

($\overline{\omega}_n$ je kompleksno konjugirani ω_n), odakle potenciranjem slijedi

$$(\omega_n^k)^{-1} = \omega_n^{-k} = \omega_n^{n-k} = \overline{\omega}_n^k = \overline{(\omega_n^k)}, \quad k = 0, \ldots, n-1.$$

Isto vrijedi i za bilo koji $k \in \mathbb{Z}$. Dakle, inverz bilo kojeg korijena iz jedinice je kompleksno konjugirani broj (što vrijedi i za inverz bilo kojeg broja na jediničnoj kružnici). ■

Naš prvi cilj je konstrukcija **brzog** algoritma za računanje $y = \mathrm{DFT}_n(a)$. Očita ideja za to je primjena strategije "podijeli–pa–vladaj", posebno nakon prethodnog teorema. Za realizaciju trebamo još određene veze između n-tih korijena iz jedinice za **razne** n, kao podlogu za rekurzivni algoritam. Napomenimo da svi dokazi idu direktno iz definicione relacije (4.2.1) za ω_n.

Lema 4.2.1. (Lema "kraćenja") *Za bilo koje prirodne brojeve $n, d \in \mathbb{N}$ i $k = 0, \ldots, n-1$, vrijedi*

$$\omega_{dn}^{dk} = \omega_n^k.$$

Tvrdnja vrijedi i za bilo koji $k \in \mathbb{Z}$, s tim da je na kraju $\omega_n^k = \omega_n^{k \bmod n}$.

Dokaz:

Po definiciji je $\omega_{dn} = e^{2\pi i/(dn)}$, pa za $k \in \mathbb{Z}$ imamo

$$\omega_{dn}^{dk} = \left(e^{2\pi i/(dn)}\right)^{dk} = e^{2dk\pi i/(dn)} = e^{2k\pi i/n} = \omega_n^k. \tag{4.2.2}$$

■

Posebno, ako je $n \in \mathbb{N}$ paran broj, onda iz (4.2.2) s $d = n/2$ izlazi

$$\omega_n^{n/2} = \omega_2 = -1. \tag{4.2.3}$$

Potpuno opravdanje za primjenu strategije "podijeli–pa–vladaj" s **faktorom** 2 proizlazi iz sljedećeg rezultata.

Lema 4.2.2. (Lema "raspolavljanja") *Ako je n* **paran** *prirodan broj, onda su kvadrati svih n-tih korijena iz jedinice upravo svi $n/2$-ti korijeni iz jedinice. Preciznije, vrijedi*

$$\left((\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2 \right) = \left(\omega_{n/2}^0, \omega_{n/2}^1, \ldots, \omega_{n/2}^{n/2-1}, \omega_{n/2}^0, \omega_{n/2}^1, \ldots, \omega_{n/2}^{n/2-1} \right),$$

tj. vektor **kvadrata** *n-tih korijena iz jedinice sadrži točno 2 kopije vektora $n/2$-tih korijena iz jedinice (jednu za drugom).*

Dokaz:

Iz leme "kraćenja" s $d = 2$ dobivamo $(\omega_n^k)^2 = \omega_{n/2}^k$, za bilo koji cijeli broj k. Kad to napravimo redom za $k = 0, \ldots, n-1$, svaki $n/2$-ti korijen iz jedinice dobivamo točno 2 puta, s "razmakom" od $n/2$ u pripadnim k-ovima. Naime, za $k = 0, \ldots, n/2 - 1$, vrijedi

$$(\omega_n^{k+n/2})^2 = \omega_n^{2k+n} = \omega_n^{2k}\omega_n^n = \omega_n^{2k} = \omega_{n/2}^k, \tag{4.2.4}$$

pa ω_n^k i $\omega_n^{k+n/2}$ imaju isti kvadrat. ■

Razlog zašto gledamo baš kvadrate n-tih korijena iz jedinice bit će jasan iz algoritma. Ako gledamo stvar "bez kvadrata" (što će nam, također, zatrebati), onda prethodnu relaciju možemo, koristeći (4.2.3), napisati u obliku

$$\omega_n^{k+n/2} = \omega_n^k \omega_n^{n/2} = -\omega_n^k, \quad k = 0, \ldots, n/2 - 1. \tag{4.2.5}$$

Drugim riječima, za parni n, u vektoru n-tih korijena iz jedinice, druga polovina ima **suprotan predznak** od prve (centralna simetrija obzirom na ishodište).

4.3. Brza Fourierova transformacija (FFT)

Sad konačno možemo napraviti **brzi** algoritam za računanje DFT_n poznat pod nazivom **brza Fourierova transformacija** (engl. Fast Fourier Transform, ili, skraćeno, FFT). Što se naziva tiče, obično se podrazumijeva da je riječ o algoritmu za **diskretnu** transformaciju, pa se riječ "diskretna" ispušta. Katkad se u literaturi mogu naći i "produljeni" nazivi s kraticom DFFT, ili FDFT (puno rjeđe, iako ispravnije).

FFT algoritam dobro koristi posebna svojstva n-tih korijena iz jedinice, tako da se $y = \mathrm{DFT}_n(a)$ može izračunati sa samo $\Theta(n \log n)$ kompleksnih aritmetičkih operacija, što je bitno brže od $\Theta(n^2)$ operacija, koliko treba direktnim pristupom iz prvog odjeljka.

Podsjetimo, cilj nam je izračunati vrijednosti polinoma

$$A(z) = \sum_{j=0}^{n-1} a_j z^j,$$

reda n, u svim n-tim korijenima iz jedinice $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$. Polinom je zadan redom n i vektorom koeficijenata $a = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{C}^n$. Usput, uređene n-torke iz $\mathbb{C}^n$ interpretiramo i kao "prave" vektore–stupce s matričnim zapisom $a = [a_0\ a_1\ \ldots\ a_{n-1}]^T$.

Za početak, bez smanjenja općenitosti, možemo pretpostaviti da je red n neka potencija broja 2, tj. da je $n = 2^m$, za neki $m \in \mathbb{N}_0$. Ako originalni red zadanog polinoma A nije tog oblika, uvijek možemo "umjetno" povećati red polinoma, dodavanjem vodećih članova s koeficijentima jednakim nula, sve dok red ne postane potencija od 2. To produljenje vektora koeficijenata odgovara "uzimanju" istog polinoma iz "većeg" vektorskog prostora. Ovim postupkom n se može povećati, ali sigurno za faktor manji od 2.

Tražene vrijednosti — komponente vektora y možemo napisati u obliku

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j (\omega_n^k)^j = \sum_{j=0}^{n-1} a_j \omega_n^{kj}, \quad k = 0, \ldots, n-1. \tag{4.3.1}$$

Od ovog zapisa trenutno nema neke koristi, ali ubrzo će biti.

Osnovni oblik FFT-a dobivamo rastavom polinoma A

$$A(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3 + \cdots + a_{n-2} z^{n-2} + a_{n-1} z^{n-1}$$

na **parne** i **neparne** dijelove, prema indeksu koeficijenata (ili eksponentima od z)

$$A(z) = (a_0 + a_2 z^2 + \cdots + a_{n-2} z^{n-2}) + z \cdot (a_1 + a_3 z^2 + \cdots + a_{n-1} z^{n-2}).$$

Eksponenti od z u zagradama su **parni** brojevi, pa u zagradama možemo napraviti supstituciju $z^2 \mapsto z$. U tu svrhu definiramo dva nova polinoma (tzv. "parni" i "neparni" polinom za A), reda $n/2$

$$A^{[0]}(z) = a_0 + a_2 z + \cdots + a_{n-2} z^{n/2-1},$$
$$A^{[1]}(z) = a_1 + a_3 z + \cdots + a_{n-1} z^{n/2-1},$$

tako da je očito

$$A(z) = A^{[0]}(z^2) + z \cdot A^{[1]}(z^2). \tag{4.3.2}$$

Označimo s $a^{[0]}$ i $a^{[1]}$ pripadne vektore koeficijenata ovih polinoma

$$a^{[0]} = (a_0, a_2, \ldots, a_{n-2}),$$
$$a^{[1]} = (a_1, a_3, \ldots, a_{n-1}).$$

Oba niza su, očito, iz $\mathbb{C}^{n/2}$. Dakle, problem reda n svodimo na 2 problema reda $n/2$. Broj 2 ovdje ima ulogu faktora za strategiju "podijeli–pa–vladaj", a relaciju

$$n = 2 \cdot \left(\frac{n}{2}\right)$$

možemo interpretirati u obliku

$$\text{stari red} = \text{faktor} \cdot \text{novi red}.$$

Vidimo da $a^{[0]}$ sadrži one koeficijente iz a čiji indeksi imaju zadnju znamenku jednaku 0 u bazi 2 (tj. indeks daje ostatak 0 pri dijeljenju s 2), a $a^{[1]}$ sadrži one koeficijente od a čiji indeksi imaju zadnju znamenku jednaku 1 u bazi 2 (odnosno, daju ostatak 1 pri dijeljenju s 2).

Time smo uspostavili vezu između gornjeg indeksa "manjih" polinoma (ili nizova), s ostatkom kojeg daju pripadni koeficijenti pri dijeljenju s faktorom (bazom) rastava. Napomenimo da **potpuno isti** princip rastava možemo iskoristiti za bilo koji prirodan broj n koji ima rastav u dva netrivijalna faktora (tj. nije prost). U tom smislu, pretpostavka da je $n = 2^m$ predstavlja samo najjednostavniji slučaj!

Princip rastava je jasan, ali još nismo gotovi s algoritmom. Treba uspostaviti vezu između $\mathrm{DFT}_n(a)$ i $\mathrm{DFT}_{n/2}$ "polovnih" nizova $a^{[0]}$ i $a^{[1]}$. Neka su $y^{[0]}$ i $y^{[1]}$ pripadne diskretne Fourierove transformacije polovnog reda za $a^{[0]}$ i $a^{[1]}$

$$\begin{aligned} y^{[0]} &= \mathrm{DFT}_{n/2}(a^{[0]}), \\ y^{[1]} &= \mathrm{DFT}_{n/2}(a^{[1]}). \end{aligned}$$

Po definiciji "polovnih" transformacija, reda $n/2$, vrijedi (pogledajte (4.3.1) za $n/2$)

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k), \\ y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k), \end{aligned} \quad k = 0, \ldots, n/2 - 1. \tag{4.3.3}$$

Naš zadatak je izračunati komponente vektora $y = \mathrm{DFT}_n(a)$

$$y_k = A(\omega_n^k), \quad k = 0, \ldots, n-1.$$

Zato uvrstimo točku $z = \omega_n^k$ u rastav (4.3.2) polinoma A. Dobivamo

$$y_k = A(\omega_n^k) = A^{[0]}\Big((\omega_n^k)^2\Big) + \omega_n^k \cdot A^{[1]}\Big((\omega_n^k)^2\Big), \quad k = 0, \ldots, n-1.$$

Na desnoj strani pojavljuju se **kvadrati** n-tih korijena iz jedinice. Lema 4.2.2. (lema "raspolavljanja") kaže da vektor **kvadrata** n-tih korijena iz jedinice sadrži točno 2 kopije vektora $n/2$-tih korijena iz jedinice, jednu za drugom. Zbog toga, prethodnu relaciju pišemo posebno za prvu i posebno za drugu polovinu komponenti vektora y.

Za prvu ili "gornju" polovinu n-tih korijena iz jedinice je

$$(\omega_n^k)^2 = \omega_n^{2k} = \omega_{n/2}^k, \quad k = 0, \ldots, n/2 - 1,$$

pa za prvu polovinu komponenti y_k vrijedi

$$y_k = A(\omega_n^k) = A^{[0]}(\omega_{n/2}^k) + \omega_n^k \cdot A^{[1]}(\omega_{n/2}^k), \quad k = 0, \ldots, n/2 - 1.$$

Prema (4.3.3), desnu stranu možemo napisati preko "polovnih" transformacija u obliku

$$y_k = y_k^{[0]} + \omega_n^k \cdot y_k^{[1]}, \quad k = 0, \ldots, n/2 - 1. \tag{4.3.4}$$

Analogno, za drugu ili "donju" polovinu n-tih korijena iz jedinice, iz (4.2.4) čitamo da je

$$(\omega_n^{k+n/2})^2 = \omega_n^{2k} = \omega_{n/2}^k, \quad k = 0, \ldots, n/2 - 1,$$

(jer ω_n^k i $\omega_n^{k+n/2}$ imaju isti kvadrat), pa za drugu polovinu komponenti $y_{k+n/2}$ vrijedi

$$y_{k+n/2} = A(\omega_n^{k+n/2}) = A^{[0]}(\omega_{n/2}^k) + \omega_n^{k+n/2} \cdot A^{[1]}(\omega_{n/2}^k), \quad k = 0, \ldots, n/2 - 1.$$

Prema (4.3.3), desnu stranu opet možemo napisati preko "polovnih" transformacija u obliku

$$y_{k+n/2} = y_k^{[0]} + \omega_n^{k+n/2} \cdot y_k^{[1]}, \quad k = 0, \ldots, n/2 - 1.$$

Na kraju, iz (4.2.5) vidimo da je $\omega_n^{k+n/2} = -\omega_n^k$, za $k = 0, \ldots, n/2 - 1$, pa je

$$y_{k+n/2} = y_k^{[0]} - \omega_n^k \cdot y_k^{[1]}, \quad k = 0, \ldots, n/2 - 1. \tag{4.3.5}$$

Usporedbom relacija (4.3.4) i (4.3.5) zaključujemo da se one razlikuju **samo** u **predznaku** drugog člana. Ako izračunamo drugi član kao pomoćnu vrijednost i zapamtimo ga, onda štedimo po jedno (kompleksno) množenje za svaku od (recimo, drugih) pola komponenti. Najbolje je odmah izračunati odgovarajuće komponente y_k i $y_{k+n/2}$ iz prve i druge polovine vektora y, tako da za pamćenje onda ne trebamo polje, već jednu jedinu varijablu t. Pripadni algoritam je

$$\left.\begin{aligned} t &= \omega_n^k \cdot y_k^{[1]}, \\ y_k &= y_k^{[0]} + t, \\ y_{k+n/2} &= y_k^{[0]} - t, \end{aligned}\right\} \quad \text{za } k = 0, \ldots, n/2 - 1. \tag{4.3.6}$$

Rezultate primjene strategije "podijeli–pa–vladaj" za računanje $y = \mathrm{DFT}_n(a)$ možemo sažeto izraziti na sljedeći način.

- Našli smo jednostavnu vezu između DFT_n polaznog niza a i $\mathrm{DFT}_{n/2}$ podnizova $a^{[0]}$ i $a^{[1]}$ (gdje je $a^{[0]}$ "parni" podniz i $a^{[1]}$ "neparni" podniz od a);
- Dodatno, pažljivom formulacijom algoritma (upotrebom pomoćne varijable t), još štedimo **polovinu** kompleksnih množenja.

Nije loše, zar ne?

Na osnovu formula (4.3.6), nije teško napisati **rekurzivni** FFT algoritam za računanje $y = \mathrm{DFT}_n(a)$, u slučaju da je $n = 2^m$, za neki $m \in \mathbb{N}_0$. Uz tu pretpostavku, uvijek možemo napraviti raspolavljanje radne duljine niza, sve dok ne

dobijemo trivijalan niz, duljine 1. Lako se vidi da je $\mathrm{DFT}_1(a) = a$, za svaki $a = (a_0)$. Usput, ovaj rezultat **ne ovisi** o točki u kojoj računamo, jer je polinom zapravo **konstanta**, pa se $\omega_1 = 1$ nigdje ne koristi.

Pojednostavljeni zapis algoritma u funkcijskom obliku, nalik na C ili Matlab, ali bez eksplicitnog navođenja tipova objekata, je:

Algoritam 4.3.1. (Rekurzivni FFT za $y = \mathrm{DFT}_n(a)$)

function FFT(a);
{ FFT: Ulaz a je kompleksni vektor. Vraća $y = \mathrm{DFT}_n(a)$.
Pretpostavka: $n = 2^m$, $m \in \mathbb{N}_0$, **ne provjerava se**! }
$n :=$ *length*(a); { *length* vraća duljinu vektora. }
if $n = 1$ **then**
return a { $y = a$ za $n = 1$. }
else
$a^{[0]} := (a_0, a_2, \ldots, a_{n-2})$;
$a^{[1]} := (a_1, a_3, \ldots, a_{n-1})$;
$y^{[0]} := \mathrm{FFT}(a^{[0]})$;
$y^{[1]} := \mathrm{FFT}(a^{[1]})$;
$\omega_n := \exp(2\pi i/n)$;
$\omega := 1$; { vidjeti napomenu iza algoritma. }
for $k := 0$ **to** n div $2 - 1$ **do**
$t := \omega_n^k \cdot y_k^{[1]}$; { $= \omega \cdot y_k^{[1]}$; }
$y_k := y_k^{[0]} + t$;
$y_{k+n/2} := y_k^{[0]} - t$;
$\omega := \omega \cdot \omega_n$; { v. napomenu. }
endfor;
return y; { kompleksni vektor duljine n. }

Formule (4.3.6) koriste vrijednosti potencija ω_n^k, za $k = 0, \ldots, n/2 - 1$, kod računanja t. Ove vrijednosti možemo akumulirati u varijabli ω, kao što je napisano u algoritmu, tako da startamo s 1 (za $k = 0$), a zatim svaku novu potenciju dobivamo množenjem stare s ω_n.

Međutim, u praksi se FFT **istog** reda n vrlo često koristi više (ili mnogo puta), za različite nizove a (svi su iste duljine). Tada se **ne** isplati svaki puta računati sve ove potencije od ω_n. Umjesto toga, potrebne potencije se unaprijed **jednom** izračunaju i spreme u tablicu (niz, polje) *omega*, duljine $n/2$, tako da je

$$omega[k] = \omega_n^k, \quad k = 0, \ldots, n/2 - 1.$$

Potrebne vrijednosti se zatim koriste **čitanjem** iz te tablice (tzv. "table–lookup").

Zadatak 4.3.1. U rekurzivnom FFT-u duljina niza varira, ovisno o razini (dubini) rekurzivnog poziva. Uzmimo da je **polazni** n fiksan (zadan) i da niz *omega* sadrži unaprijed izračunatu tablicu potencija ω_n^k, za $k = 0, \ldots, n/2 - 1$, na gore opisani način. Preuredite gornji algoritam tako da indeksiranjem u ovom nizu korektno koristi **sve** potrebne potencije **svih** potrebnih korijena iz jedinice. Vanjski (fiskni) n i niz *omega* je zgodno smatrati globalnim podacima (ili ih treba zadati kao dodatne ulazne parametre). ■

I prije analize složenosti ovog algoritma, možemo uočiti da se on sigurno može ubrzati (bar ponešto). Ubrzanje je istog tipa kao i kod većine "prvih varijanti" rekurzivnih algoritama (sjetite se Hanojskih tornjeva).

Na dnu rekurzije, za $n = 1$, naš algoritam praktički **ne radi ništa**, samo vraća ulazni niz. Dakle, imamo tipični primjer za "uđi–izađi". To bismo riješili tako da "vanjski" FFT algoritam testira trivijalni slučaj $n = 1$ (koji je, ionako, potpuno nezanimljiv u praksi), a u rekurzivnom ("unutarnjem") dijelu algoritma treba napraviti "dno" rekurzije za slučaj $n = 2$. Probajte!

Međutim, ni to se ne isplati raditi za "pravu" praktičnu upotrebu. Rekurzivni algoritam je **vrlo zgodan** za osnovnu analizu složenosti, pa ćemo ga točno zato i iskoristiti.

S druge strane, napomenimo odmah da se rekurzivna varijanta FFT algoritma vrlo **rijetko** koristi u praksi. Kad se algoritam "raspakira" u tzv. iterativnu varijantu (v. kasnije), dobiva se nešto brži algoritam (vremenska složenost je istog reda veličine, ali je multiplikativna konstanta nešto manja).

Uostalom, budimo do kraja pošteni. Vjerojatno se uopće **ne isplati** pisati vlastitu varijantu FFT algoritma u nekom programskom jeziku. Treba koristiti **gotove** potprograme iz raznih biblioteka, po mogućnosti, optimizirane za računalo (procesor) na kojem se radi. Na primjer, Intelov "Math Kernel Library" (skraćeno, MKL) sadrži hrpu standardnih FFT algoritama, a realizacija je, naravno, specijalno prilagođena (i brza) na Intelovim procesorima.

4.3.1. Složenost FFT za $n = 2^m$

Za početak, napravit ćemo grubu procjenu vremenske složenosti rekurzivnog FFT algoritma, uz pretpostavku **sekvencijalnog** izvršavanja svih naredbi (naročito rekurzivnih poziva).

Ulazni podatak za FFT je kompleksni vektor a, duljine $n = 2^m$. Dakle, složenost, sasvim općenito, ovisi o a i n.

Cijeli algoritam radi neke operacije na kompleksnim vektorima. Osim rekurzivnih poziva, u algoritmu imamo sljedeće operacije:

- kopiranje vektora — priprema podnizova i, eventualno, kopiranje parametara na "stacku" (stogu) za rekurziju,
- kompleksne aritmetičke operacije na komponentama vektora,
- cjelobrojne aritmetičke operacije za indeksiranje komponenti i kontrolu petlji.

Kompleksan broj možemo prikazati kao par realnih (na pr., realni i imaginarni dio broja), a zatim sve kompleksne aritmetičke operacije možemo realizirati preko realnih.

Pretpostavimo sad da za realne i cijele brojeve koristimo odgovarajuće **standardne** tipove podataka podržane arhitekturom računala (recimo, `integer` i `real` u Pascalu, odnosno, `int` i `double` u C-u).

Onda sve aritmetičke operacije (cjelobrojne i realne, odnosno, kompleksne) postaju **elementarne**. Trajanje svake takve operacije je između neke dvije **konstante**, tj. $\Theta(1)$, i **ne ovisi** o **vrijednostima** brojeva u operaciji (operandima i rezultatu). Poptuno isto vrijedi i za operacije pristupa, odnosno, dodjeljivanja (ili kopiranja) vrijednosti odgovarajućih tipova.

Dakle, sve operacije na pojedinačnim komponentama vektora i indeksima u FFT-u su elementarne i ne ovise o vrijednostima brojeva, odnosno, indeksa. Zbog toga, složenost ovisi samo o n (broju komponenti, tj. duljini vektora), a ne ovisi o a (vrijednostima komponenti vektora).

Neka je $T(n)$ vremenska složenost rekurzivnog FFT algoritma za ulazni vektor duljine n, uz standardnu pretpostavku da je $n = 2^m$, za neki $m \in \mathbb{N}_0$. Tada $T(n)$ možemo napisati u obliku

$$T(n) = T_{\text{rekurzivno}}(n) + T_{\text{ostalo}}(n), \tag{4.3.7}$$

gdje je $T_{\text{rekurzivno}}(n)$ trajanje rekurzivnih poziva "polovnih" FFT-ova, a $T_{\text{ostalo}}(n)$ trajanje svih ostalih operacija u algoritmu.

Rekurzivni dio algoritma ima 2 poziva FFT-a na nizovima duljine $n/2$. Ako ih radimo **sekvencijalno**, onda je ukupno trajanje

$$T_{\text{rekurzivno}}(n) = 2T(n/2).$$

Ostatak algoritma na početku sadrži 2 kopiranja vektora duljine $n/2$, a zatim petlju koja se izvršava $n/2$ puta. Kopiranja vektora (sekvencijalno) sigurno možemo napraviti u vremenu koje je najviše **linearno** u n. Reorganizacijom algoritma, taj dio se možda može i izbjeći, tako da radimo na **jednom** globalnom polju, uz pažljivo indeksiranje. Međutim, sekvencijalno izvođenje petlje, koja sadrži samo nekoliko elementarnih operacija, **sigurno** traje **linearno** u n (ili $n/2$, svejedno je). Dakle, uz sekvencijalno izvršavanje, očito vrijedi

$$T_{\text{ostalo}}(n) \in \Theta(n).$$

Kad sve to uvrstimo u (4.3.7), dobivamo

$$T(n) \in 2T(n/2) + \Theta(n).$$

Ako napišemo rekurzije za donju i gornju ogradu za $T(n)$, obje imaju isti oblik

$$T(n) = 2T(n/2) + cn,$$

samo s različitim konstantama c, koje dobivamo iz donje i gornje ograde za $\Theta(n)$. Za početni uvjet očito vrijedi $T(1) \in \Theta(1)$. Ovakve rekurzije smo već rješavali (pogledajte odjeljak o rekurzijama), pa zaključujemo da za $T(n)$ vrijedi

$$T(n) \in \Theta(n \log n), \quad \text{uvjetno, kad je } n = 2^m.$$

Time smo dokazali sljedeći teorem.

Teorem 4.3.1. *Vremenska složenost sekvencijalnog izvršavanja FFT algoritma za vektor duljine $n = 2^m$ je*

$$T(n) \in \Theta(n \log n),$$

uz pretpostavku da su sve operacije na komponentama vektora elementarne.

Zanimljivo je da se vrlo jednostavno može procijeniti i vremenska složenost za **paralelno** izvršavanje FFT algoritma. Pretpostavimo da na raspolaganju imamo (barem) $n/2$ procesora "slične" snage kao i onaj za sekvencijalno izvršavanje, tako da su iste operacije kao i ranije, elementarne i za **svaki** od ovih procesora. Dodatno, radi jednostavnosti modela, uzmimo da je memorija **zajednička** za sve procesore, tj. što jedan procesor tamo spremi, svi ostali mogu (brzo) pročitati. To znači da su dostupi, dodjeljivanja i kopiranja pojedinačnih vrijednosti opet elementarne operacije.

Rekurzivni dio algoritma ima 2 poziva FFT-a na nizovima duljine $n/2$. Ta dva poziva su **nezavisna**, u smislu da rade na nezavisnim podacima, tj. jedan ne ovisi o rezultatima drugog. Dakle, smijemo ih napraviti **paralelno** — istovremeno, pa je ukupno trajanje (uz pretpostavku da oba imaju isto trajanje)

$$T_{\text{rekurzivno}}(n) = T(n/2).$$

Uočite uštedu za faktor 2 obzirom na sekvencijalno izvršavanje.

U ostatku algoritma, kopiranje vektora se trivijalno paralelizira, tako da svaki procesor kopira svoju komponentu (jer imamo barem $n/2$ procesora, a toliko komponenti treba iskopirati). Dakle, paralelna kopiranja vektora sigurno možemo napraviti u (najviše) **konstantnom** vremenu.

Ostaje vidjeti da li možemo paralelizirati petlju koja se izvršava $n/2$ puta. Pojedini prolazi kroz tu petlju rade na potpuno **nezavisnim** podacima, pa ih je trivijalno paralelizirati. Jedini stvarno sekvencijalni dio u toj petlji je akumulacija

potencija od ω_n. Ako taj dio realiziramo čitanjem potrebne potencije iz tablice u zajedničkoj memoriji, onda svaki od $n/2$ procesora može paralelno (u isto vrijeme) izvršiti svoje operacije iz petlje za svoju **fiksnu** vrijednost od k. Tih nekoliko elementarnih operacija po procesoru očito traje **konstantno** vrijeme. Dakle, za ovakvo paralelno izvršavanje vrijedi

$$T_{\text{ostalo}}(n) \in \Theta(1).$$

Usput, ako su svi procesori "isti", onda svakome od njih treba isto vrijeme da obavi svoj dio posla, što opravdava isto trajanje svih rekurzivnih poziva za vektore iste duljine. Kad to uvrstimo u (4.3.7), dobivamo

$$T(n) \in T(n/2) + \Theta(1).$$

Rekurzije za donju i gornju ogradu za $T(n)$ sad imaju oblik

$$T(n) = T(n/2) + c,$$

samo s različitim konstantama c, koje dobivamo iz donje i gornje ograde za $\Theta(1)$. Za početni uvjet očito i dalje vrijedi $T(1) \in \Theta(1)$. Odmah se vidi da za $T(n)$ vrijedi

$$T(n) \in \Theta(\log n), \quad \text{uvjetno, kad je } n = 2^m.$$

Time smo dokazali sljedeći teorem.

Teorem 4.3.2. *Vremenska složenost paralelnog izvršavanja FFT algoritma za vektor duljine $n = 2^m$ na (barem) $n/2$ procesora je*

$$T(n) \in \Theta(\log n),$$

uz pretpostavku da su sve operacije na komponentama vektora elementarne.

Zadatak 4.3.2. Ovaj rezultat, zasad, **ne uključuje** vrijeme potrebno za pripremu tablice potrebnih potencija od ω_n u zajedničkoj memoriji. Uz pretpostavku da imamo istih $n/2$ procesora na raspolaganju, pokažite kako se takva tablica može **paralelno** izračunati u vremenu $\Theta(\log n)$. To dokazuje da prethodni teorem vrijedi i kad pripremu tablice uračunamo u ukupno paralelno trajanje algoritma. ■

Vidimo da se FFT može gotovo **idealno** paralelizirati, jer paralelno trajanje odgovara sekvencijalnom podijeljenom s brojem procesora. Uostalom, barem u ovom pojednostavljenom modelu sa zajedničkom memorijom, očito je da **svi** procesori **svo** vrijeme rade, tj. nema "praznog hoda". Isto vrijedi i za druge, realnije modele paralelnog računanja, što je vrlo bitno i korisno u praksi.

Znamo da je vremenska složenost relativno neprecizan pojam. Standardno je određena najviše do na multiplikativnu konstantu koja mjeri "frekvenciju" računala (tj. ta konstanta je obrnuto proporcionalna brzini računala).

Najjednostavniji i, po rezultatu, najprecizniji uvid u složenost FFT-a dobivamo **brojanjem** kompleksnih aritmetičkih operacija u algoritmu. Posebno brojimo aditivne i multiplikativne operacije, s tim da oduzimanje smatramo ekvivalentnim zbrajanju. Kod multiplikativnih operacija nema problema, jer ionako imamo samo množenja. Brojevi pojedinih vrsta operacija, očito, ovise samo i isključivo o n.

Neka je $A(n)$ broj aditivnih, a $M(n)$ broj multiplikativnih kompleksnih aritmetičkih operacija u FFT algoritmu za zadani $n = 2^m$. Aritmetičke operacije se eksplicitno pojavljuju samo unutar petlje po k. U svakom prolazu kroz tu petlju, za fiksni k, imamo točno 2 zbrajanja. Za brojanje množenja, pretpostavit ćemo da potencije od ω_n računamo unaprijed i koristimo indeksiranjem u tablici. Onda imamo točno 1 množenje (s ω_n^k). U protivnom, ovako kako je napisano, imamo 2 množenja, pa finalni $M(n)$ treba naprosto pomnožiti s 2.

To znači da u $n/2$ prolaza kroz petlju imamo ukupno n zbrajanja i $n/2$ množenja. Tome, naravno, treba dodati operacije u rekurzivnim pozivima FFT-a. Dakle, za $A(n)$ i $M(n)$ vrijede rekurzivne relacije

$$\begin{aligned} A(n) &= 2A(n/2) + n, \\ M(n) &= 2M(n/2) + n/2. \end{aligned}$$

Znamo da je **lokalni** n na svim razinama rekurzije uvijek neka potencija od 2, tj. lokalni n prolazi vrijednosti oblika 2^ℓ, za $\ell = m, m-1, \ldots, 1, 0$. Na dnu rekurzije, za $n = 1$, očito vrijede početni uvjeti $A(1) = M(1) = 0$, jer se ništa ne računa.

Ove dvije rekurzije se lako rješavaju već poznatim tehnikama (probajte!). Za razliku od inače, ovdje ćemo ih riješiti direktno, supstitucijom unatrag.

Uzmimo da je $n = 2^m$ fiksan i računamo $A(n) = A(2^m)$. Rekurzija kaže da je

$$A(2^\ell) = 2A(2^{\ell-1}) + 2^\ell, \quad \ell = m, \ldots, 1,$$

i $A(1) = 0$. Supstitucija unatrag daje

$$\begin{aligned} A(2^m) &= 2A(2^{m-1}) + 2^m \\ &= \text{supstitucija} = 2\Big(2A(2^{m-2}) + 2^{m-1}\Big) + 2^m \\ &= 2^2 A(2^{m-2}) + 2 \cdot 2^{m-1} + 2^m = 2^2 A(2^{m-2}) + 2 \cdot 2^m \\ &= \text{supstitucija} = 2^2\Big(2A(2^{m-3}) + 2^{m-2}\Big) + 2 \cdot 2^m \\ &= 2^3 A(2^{m-3}) + 2^2 \cdot 2^{m-2} + 2 \cdot 2^m = 2^3 A(2^{m-3}) + 3 \cdot 2^m, \end{aligned}$$

i tako redom. Opći korak supstitucije za neki $k = 0, \ldots, m-1$, ima oblik

$$\begin{aligned} A(2^m) &= 2^k A(2^{m-k}) + k \cdot 2^m, \\ &= \text{supstitucija} = 2^k\Big(2A(2^{m-k-1}) + 2^{m-k}\Big) + k \cdot 2^m \\ &= 2^{k+1} A(2^{m-k-1}) + 2^k \cdot 2^{m-k} + k \cdot 2^m = 2^{k+1} A(2^{m-k-1}) + (k+1) \cdot 2^m. \end{aligned}$$

Obzirom na to da polazna relacija očito vrijedi za $k = 0$, ovaj korak je ekvivalentan indukciji unatrag, pa smo dokazali da vrijedi

$$A(n) = A(2^m) = 2^k A(2^{m-k}) + k \cdot 2^m = 2^k A(n/2^k) + k \cdot n,$$

za sve $k = 0, \ldots, m$. Kad ovu relaciju pročitamo za $k = m$, dobijemo

$$A(n) = A(2^m) = 2^m A(1) + m \cdot 2^m = nA(1) + n \lg n = n \lg n,$$

jer je $A(1) = 0$. Dakle, dokazali smo da je $A(n) = n \lg n$, za svaki $n = 2^m$.

Alternativni dokaz iste činjenice možemo dobiti i tzv. tehnikom "zbrajanja". Napišimo redom (unatrag) sve potrebne rekurzivne relacije za $A(2^\ell)$, $\ell = m, \ldots, 1$.

$$\begin{aligned}
A(2^m) &= 2A(2^{m-1}) + 2^m \\
A(2^{m-1}) &= 2A(2^{m-2}) + 2^{m-1} \\
\vdots \quad &= \quad \vdots \\
A(2^\ell) &= 2A(2^{\ell-1}) + 2^\ell \\
\vdots \quad &= \quad \vdots \\
A(2^2) &= 2A(2) + 2^2 \\
A(2) &= 2A(1) + 2.
\end{aligned}$$

Počev od druge jednadžbe, redom, svaku jednadžbu pomnožimo faktorom $2^{m-\ell}$, tako da na lijevoj strani dobijemo isti faktor kao i na desnoj strani u prethodnoj jednadžbi (nakon množenja). Nakon toga, popis jednadžbi ima oblik

$$\begin{aligned}
A(2^m) &= 2A(2^{m-1}) + 2^m \\
2A(2^{m-1}) &= 4A(2^{m-2}) + 2 \cdot 2^{m-1} \\
\vdots \quad &= \quad \vdots \\
2^{m-\ell} A(2^\ell) &= 2^{m-\ell+1} A(2^{\ell-1}) + 2^{m-\ell} \cdot 2^\ell \\
\vdots \quad &= \quad \vdots \\
2^{m-2} A(2^2) &= 2^{m-1} A(2) + 2^{m-2} \cdot 2^2 \\
2^{m-1} A(2) &= 2^m A(1) + 2^{m-1} \cdot 2.
\end{aligned}$$

Zbrojimo sve ove jednadžbe (ima ih točno $m = \lg n$) i odmah skratimo iste članove na lijevoj i desnoj strani zbroja. Dobivamo

$$A(2^m) = 2^m A(1) + 2^m + 2 \cdot 2^{m-1} + \cdots + 2^{m-\ell} \cdot 2^\ell + \cdots + 2^{m-2} \cdot 2^2 + 2^{m-1} \cdot 2.$$

Uvrstimo $A(1) = 0$ i uočimo da je $2^{m-\ell} \cdot 2^\ell = 2^m = n$, te da takvih članova ima točno m. Na kraju ostaje

$$A(n) = A(2^m) = m \cdot 2^m = n \lg n.$$

Iz algoritma ili rekurzije odmah se vidi da je broj množenja $M(n)$ jednak polovini broja zbrajanja $A(n)$, jer na svaka 2 zbrajanja imamo 1 množenje. Dobivene rezultate možemo formulirati na sljedeći način.

Teorem 4.3.3. (Aritmetička složenost FFT) *FFT algoritam za vektor duljine* $n = 2^m$ *treba*

$$A(n) = n \lg n \quad i \quad M(n) = \frac{1}{2} n \lg n$$

aditivnih i multiplikativnih kompleksnih aritmetičkih operacija (respektivno).

Strogo govoreći, rezultat za multiplikativne operacije vrijedi uz pretpostavku da je tablica potencija od ω_n izračunata unaprijed (izvan FFT-a), a operacije za to **nisu** uračunate u $M(n)$. Ako računanje ove tablice uključujemo u FFT algoritam, onda $M(n)$ treba povećati za $M_1(n) = n/2 - 1$, jer toliko množenja treba za računanje te tablice.

Sada imamo efikasan (tj. brzi) algoritam za rješenje problema evaluacije ili izračunavanja u svim n-tim korijenima iz jedinice.

4.4. Inverzna transformacija

Obratni problem interpolacije u svim n-tim korijenima iz jedinice ili inverz diskretne Fourierove transformacije $a = \mathrm{DFT}_n^{-1}(y)$ ekvivalentan je računanju vektora $a = V_n^{-1} y$. Kao što ćemo pokazati, rješavanje linearnog sustava $y = V_n a$ **nije** potrebno, jer se V_n^{-1} može jednostavno izraziti preko V_n. Za računanje inverza V_n^{-1} koristimo sljedeću lemu o n-tim korijenima iz jedinice.

Lema 4.4.1. (Lema "zbrajanja") *Za bilo koji prirodan broj* $n \in \mathbb{N}$ *i cijeli broj* $k \in \mathbb{Z}$ *vrijedi*

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = \begin{cases} 0, & \text{ako } k \textbf{ nije } \text{djeljiv s } n, \\ n, & \text{ako je } k \text{ djeljiv s } n. \end{cases}$$

Dokaz:

Ako je k djeljiv s n, tj. $k = \ell n$ za neki $\ell \in \mathbb{Z}$, onda iz leme "kraćenja" s $d = n$ dobivamo $\omega_n^k = \omega_1^\ell = 1$, jer je $\omega_1 = 1$. Dakle, svaki član u sumi je jednak 1, a ima ih n, pa je suma jednaka n.

Ako k nije djeljiv s n, onda je $k \bmod n \neq 0$, pa je $\omega_n^k = \omega_n^{k \bmod n} \neq 1$. Tražena suma je geometrijska suma s faktorom $q = \omega_n^k \neq 1$, pa je

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} = \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} = \frac{1^k - 1}{\omega_n^k - 1} = 0,$$

što dokazuje tvrdnju. ■

Opća Vandermondeova matrica za vektor točaka $(z_0, \ldots, z_{n-1})$ ima oblik

$$V_n(z_0, \ldots, z_{n-1}) = \begin{bmatrix} z_0^0 & z_0^1 & \cdots & z_0^{n-1} \\ z_1^0 & z_1^1 & \cdots & z_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n-1}^0 & z_{n-1}^1 & \cdots & z_{n-1}^{n-1} \end{bmatrix}.$$

Ako se dogovorimo da retke i stupce matrice brojimo (tj. indeksiramo) počev od 0 (a ne od 1), kao što smo to dosad radili i za vektore, onda za element na presjeku k-tog retka i j-tog stupca opće Vandermondeove matrice vrijedi

$$[\, V_n(z_0, \ldots, z_{n-1}) \,]_{kj} = z_k^j, \quad k, j = 0, \ldots, n-1.$$

Pogledajmo sad kako izgleda Vandermondeova matrica V_n za vektor n-tih korijena iz jedinice, $z_k = \omega_n^k$, za $k = 0, \ldots, n-1$. Očito je

$$[\, V_n \,]_{kj} = \omega_n^{kj}, \quad k, j = 0, \ldots, n-1.$$

Cijela matrica ima oblik

$$V_n = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix}.$$

Odmah vidimo da je V_n kompleksna **simetrična** matrica, tj. vrijedi $V_n^T = V_n$, ali nije Hermitska (tj. $V_n^* \neq V_n$). Simetričnost znači da su k-ti redak i k-ti stupac isti (do na transponiranje), kad ih gledamo kao vektore iz $\mathbb{C}^n$.

Za lakše pisanje, vektor koji odgovara k-tom stupcu matrice V_n označavamo s $v_{(k)} \in \mathbb{C}^n$, tako da je

$$v_{(k)} = (\omega_n^{k\cdot 0}, \omega_n^{k\cdot 1}, \ldots, \omega_n^{k\cdot(n-1)}), \quad k = 0, \ldots, n-1,$$

odnosno, po elementima vrijedi

$$(v_{(k)})_j = [\, V_n \,]_{kj} = \omega_n^{kj}, \quad j, k = 0, \ldots, n-1.$$

Iz relacije $[\, V_n \,]_{kj} = \omega_n^{kj}$ dobivamo još i zgodnu interpretaciju za sumu iz leme zbrajanja

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = \sum_{j=0}^{n-1} \omega_n^{kj} = \sum_{j=0}^{n-1} [\, V_n \,]_{kj} = \sum_{j=0}^{n-1} (v_{(k)})_j.$$

Ta suma je zbroj svih elemenata k-tog retka (ili k-tog stupca) matrice V_n. Ako još uočimo da 0-ti redak (stupac) sadrži sve jedinice, ovu sumu možemo interpretirati i kao **skalarni produkt** k-tog i 0-tog retka (stupca).

Prisjetimo se definicije **kompleksnog skalarnog produkta** dvaju vektora $x, y \in \mathbb{C}^n$, s tim da komponente brojimo od nule

$$\langle x, y \rangle := \sum_{j=0}^{n-1} x_j \bar{y}_j = y^* x.$$

Zadnja formula (matrično množenje) vrijedi kad vektore smatramo jednostupčanim matricama.

Vratimo se sad na sumu iz leme zbrajanja i iskoristimo rezultat te leme, s tim da je $k \in \{0, \ldots, n-1\}$. Dobivamo

$$\langle v_{(k)}, v_{(0)} \rangle = \sum_{j=0}^{n-1} \omega_n^{kj} \cdot \omega_n^{0 \cdot j} = \sum_{j=0}^{n-1} (\omega_n^k)^j = \begin{cases} 0, & \text{ako } k \in \{1, \ldots, n-1\}, \\ n, & \text{ako je } k = 0. \end{cases}$$

To znači da su svi stupci matrice V_n, osim 0-tog, **ortogonalni** (ili okomiti) na 0-ti stupac.

Za skalarni produkt bilo koja dva stupca $v_{(k)}$ i $v_{(\ell)}$ matrice V_n, s tim da je $k, \ell \in \{0, \ldots, n-1\}$, dobivamo

$$\langle v_{(k)}, v_{(\ell)} \rangle = \sum_{j=0}^{n-1} \omega_n^{kj} \cdot \overline{(\omega_n^{\ell j})} = \sum_{j=0}^{n-1} \omega_n^{kj} \cdot \overline{\omega}_n^{\ell j}.$$

Sad iskoristimo činjenicu da je $\overline{\omega}_n = \omega_n^{-1}$, pa slijedi

$$\langle v_{(k)}, v_{(\ell)} \rangle = \sum_{j=0}^{n-1} \omega_n^{kj} \cdot \omega_n^{-\ell j}. = \sum_{j=0}^{n-1} (\omega_n^{k-\ell})^j.$$

Prema lemi zbrajanja, rezultat ovisi o tome da li je $k - \ell$ djeljiv s n ili ne. No, zbog $k, \ell \in \{0, \ldots, n-1\}$, razlika $k - \ell$ može biti djeljiva s n ako i samo ako je $k = \ell$. Dakle, na kraju izlazi

$$\langle v_{(k)}, v_{(\ell)} \rangle = \begin{cases} 0, & \text{ako } k \neq \ell, \\ n, & \text{ako je } k = \ell. \end{cases}$$

Što znači ovaj rezultat? Bilo koja dva različita stupca matrice V_n su međusobno **ortogonalna** (okomita). To je standardno svojstvo **unitarnih** matrica (odnosno, ortogonalnih matrica, u realnom slučaju). Međutim, norme stupaca **nisu** jednake 1, kao kod unitarnih (odnosno, ortogonalnih) matrica, već vrijedi

$$\|v_{(k)}\| = \sqrt{\langle v_{(k)}, v_{(\ell)} \rangle} = \sqrt{n}, \quad k = 0, \ldots, n.$$

Svi stupci imaju istu normu, koju možemo "izlučiti" ispred matrice, pa onda dobivamo unitarnu matricu. Dakle, V_n je **skalarni višekratnik** neke **unitarne** matrice, s faktorom $\sqrt{n}$.

Ako tu unitarnu matricu označimo s U_n, onda je

$$V_n = \sqrt{n}\, U_n, \quad \text{ili} \quad U_n = \frac{1}{\sqrt{n}}\, V_n.$$

Naš rezultat o skalarnim produktima stupaca matrice V_n, zapravo, dokazuje da je $V_n^* V_n = nI_n$, gdje je I_n jedinična matrica reda n. U terminima matrice U_n, taj rezultat glasi $U_n^* U_n = I_n$, a to baš znači da je U_n unitarna. Uočite da je isti rezultat za obratni produkt trivijalan, jer su V_n i U_n regularne kvadratne matrice.

Nakon ovog, zaista nije teško izračunati inverz V_n^{-1}. Za unitarne matrice vrijedi $U_n^{-1} = U_n^*$, pa je

$$V_n^{-1} = (\sqrt{n}\, U_n)^{-1} = \frac{1}{\sqrt{n}}\, U_n^{-1} = \frac{1}{\sqrt{n}}\, U_n^* = \frac{1}{\sqrt{n}}\, \frac{1}{\sqrt{n}}\, V_n^* = \frac{1}{n}\, V_n^*.$$

Uostalom, dovoljno je $V_n^* V_n = nI_n$ podijeliti s n i napisati u obliku

$$\left(\frac{1}{n}\, V_n^*\right) V_n = I_n,$$

pa odmah čitamo inverz.

Ako se sjetimo da je V_n simetrična, onda se V_n^* dobiva samo konjugiranjem elemenata matrice V_n. Dodatno, zbog $\omega_n^{-1} = \omega_n^{n-1} = \overline{\omega}_n$, po elementima vrijedi

$$[\, V_n^* \,]_{kj} = \overline{(\omega_n^{kj})} = \overline{\omega}_n^{kj} = (\omega_n^{-1})^{kj} = \omega_n^{-kj}, \quad k, j = 0, \ldots, n-1.$$

Dakle, V_n^* je, također, Vandermondeova matrica i to u vektoru **recipročnih** ili **konjugiranih** svih n-tih korijena iz jedinice. Na kraju, uočimo da zbog "skaliranja" s $1/n$, V_n^{-1} više **nije** Vandermondeova matrica (ali to i nije bitno).

Teorem 4.4.1. (Inverz DFT**)** *Za inverz V_n^{-1} Vandermondeove matrice V_n diskretne Fourierove transfomacije vrijedi relacija*

$$V_n^{-1} = \frac{1}{n}\, V_n^*.$$

Po elementima, ta relacija ima oblik

$$[\, V_n^{-1} \,]_{kj} = \frac{1}{n}\, \omega_n^{-kj}, \quad k, j = 0, \ldots, n-1.$$

Dokaz:

Sve ove činjenice smo već dokazali. Naravno, ako "zaboravimo" svu prethodnu raspravu, dokaz se može provesti i direktno, provjerom relacije $V_n^{-1} V_n = I_n$ po elementima, uz izravni poziv na lemu zbrajanja.

Pogledajmo element na presjeku ℓ-tog retka i k-tog stupca u produktu $V_n^{-1}V_n$, za $k, \ell \in \{0, \ldots, n-1\}$.

$$[\, V_n^{-1}V_n \,]_{\ell k} = \sum_{j=0}^{n-1} [\, V_n^{-1} \,]_{\ell j} [\, V_n \,]_{jk} = \sum_{j=0}^{n-1} \frac{1}{n}\, \omega_n^{-\ell j} \cdot \omega_n^{jk} = \frac{1}{n} \sum_{j=0}^{n-1} (\omega_n^{k-\ell})^j$$
$$= \begin{cases} 0, & \text{ako } k \neq \ell, \\ 1, & \text{ako je } k = \ell. \end{cases}$$

Dakle, vrijedi $[\, V_n^{-1}V_n \,]_{\ell k} = \delta_{\ell k}$ (Kroneckerov simbol), pa je $V_n^{-1}V_n = I_n$. ■

4.5. Svojstva DFT

Za konstrukciju brzog algoritma za množenje polinoma trebamo nekoliko osnovnih svojstava ove transformacije.

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{k \cdot j},$$

$$u_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{k \cdot j},$$

$$v_k = B(\omega_n^k) = \sum_{j=0}^{n-1} b_j \omega_n^{k \cdot j},$$

$$w_k = C(\omega_n^k) = \sum_{j=0}^{n-1} C_j \omega_n^{k \cdot j},$$

Literatura

[1] KEY = `Aho-Hopcroft-Ullmann-1974`

A. V. Aho, J. E. Hopcroft, and J. D. Ullmann, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, Massachusetts, 1974.

[2] KEY = `Aho-Hopcroft-Ullmann-1983`

——, *Data Structures and Algorithms*, Addison–Wesley, Reading, Massachusetts, 1983. (Reprinted with corrections April 1987.).

[3] KEY = `Alsuwaiyel-1999`

M. H. Alsuwaiyel, *Algorithms — Design Techniques and Analysis*, World Scientific, Singapore, 2003.

[4] KEY = `Brassard-Bratley-1988`

G. Brassard and P. Bratley, *Algorithmics*, Prentice–Hall International, Englewood Cliffs, New Jersey, 1988.

[5] KEY = `Cormen-Leiserson-Rivest-1990`

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 1990.

[6] KEY = `Horowitz-Sahni-1976`

E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Pitman, London, 1976.

[7] KEY = `Horowitz-Sahni-1978`

——, *Fundamentals of Computer Algorithms*, Pitman, London, 1978.

[8] KEY = `Johnsonbaugh-Schaefer-2004`

R. Johnsonbaugh and M. Schaefer, *Algorithms*, Pearson Education International, Upper Saddle River, New Jersey, 2004.

[9] KEY = `Kleinberg-Tardos-2006`

J. Kleinberg and É. Tardos, *Algorithm Design*, Pearson Education, Inc., Boston, Massachusetts, 1st ed., 2006.

[10] KEY = `Knuth-1973`

D. E. Knuth, *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, Addison–Wesley, Boston, Massachusetts, 3rd ed., 1997.

[11] KEY = `Knuth-1981`

——, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, Addison–Wesley, Boston, Massachusetts, 3rd ed., 1998.

[12] KEY = `Knuth-1973a`

——, *The Art of Computer Programming, Volume 3, Sorting and Searching*, Addison–Wesley, Boston, Massachusetts, 2nd ed., 1998.

[13] KEY = `Knuth-2011`

——, *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms, Part 1*, Addison–Wesley, Upper Saddle River, New Jersey, 1st ed., 2011.

[14] KEY = `Kronsjoe-1979`

L. I. Kronsjö, *Algorithms: Thier Complexity and Efficiency*, John Wiley & Sons, Chichester, 1979.

[15] KEY = `McHugh-1990`

J. A. McHugh, *Algorithmic Graph Theory*, Prentice–Hall International, Englewood Cliffs, New Jersey, 1990.

[16] KEY = `Wilf-1986`

H. S. Wilf, *Algorithms and Complexity*, Prentice–Hall, Englewood Cliffs, New Jersey, 1986.

[17] KEY = `Wirth-1973`

N. Wirth, *Systematic Programming: An Introduction*, Prentice–Hall, Englewood Cliffs, New Jersey, 1973.

[18] KEY = `Wirth-1976`

——, *Algorithms + Data Structures = Programs*, Prentice–Hall, Englewood Cliffs, New Jersey, 1976.