Keerukus – võistlusprogrammerija pragmaatika

Marko Tsengov

12. november 2023. a.

1 O?

$O(1)$, $O(n)$, $O(n^2)$, ... – keerukushinnangud, aga kuidas ja milleks?

Koodi puhul on võrdlemisi raske käivitamata määrata, kui kaua aega programmil (halvimate lubatud sisendite korral) aega võiks minna. Samas on meil meetodeid, millega kuluvat aega vähemasti hinnata.

Järgnevalt on toodud neli koodinäidist. Materjali läbides võiks üritada leida iga näidise ajalise keerukuse, lõpuks räägitakse ka kõigist.

```
# A
print(312)

############################
# B
n = int(input())
for i in range(n):
    for j in range(n):
        print(i + j)

############################
# C
n = int(input())
for j in range(5):
    for i in range(n):
        print(i + j)

############################
# D
n = int(input())
a = []
for i in range(n):
    a.append(i**2)
    print(a)
```

1.1 O(1)

Näidises A toodud kood on äärmiselt lihtne – kood prindib ühe täisarvu, mingeid valikuid ega muud keerulist koodil praktiliselt teha pole. Seejuures pole tegelikult tähtis, kui *keerulist* ülesannet koodilõik lahendab, vaid just see, et see alati ligikaudu sama palju tööd teeb. Näiteks ka järgmised funktsioonid täidavad oma töö (mõistliku suurusega sisendite puhul) konstantse ajaga:

```
def f1(x):
    return x

def f2(a, b, c):
    return (a + b + c) / 3
```

Saab väita, et ilmselt on funktsiooni `f1` käitamine märksa kiirem, kui funktsiooni `f2` käitamine, kuid see osutub enamasti üsna tühiseks detailiks.

Kui mingi koodilõigu täitmine võtab pidevalt ligikaudu sama palju aega, võime öelda, et selle *keerukus* on $O(1)$. Seejuures $O(...)$ on matemaatiliselt defineeritud funktsiooni leiduvuse ja ülalt piiramise põhjal, kuid meie kasutuses piisab teada, et see annab mingi tõkke (seose), kui aeglaselt koodilõik antud sisendi korral joosta võib. $O(1)$ tähendab, et muust sõltumata kulub aega alati sama palju.

Tasub tähele panna, et $O(1)$ koodilõikude järjest asetamisel saame samuti $O(1)$ lõigu. Praktikas on vaja teada vaid põhilisi meetodeid, mis töötavad ligikaudu konstantse ajaga. Siin on mõned kasulikumad näited:

- Aritmeetika `+-/*%` põhjal (lisaks bitioperaatorid).
- Omistamine `=`
- Igasorti võrdlused `<`, `==`, ...
- Massivist elemendi võtmine indeksi põhjal (`a[i]`)
- Järjendisse (`list`, `vector<T>`, `ArrayList<T>`) elemendi lisamine* või lõpust eemaldamine.
- Väärtuse sisse lugemine / väljastamine mingist sisendist / väljundist
- Konstantse hulga läbimistega tsükkel (nt. `for i in range(5):`) (tsüklitest kohe lähemalt)

Kokkuvõtteks:

- **Konstantne aeg** – $O(1)$**.**
- **Lihtsaimad tegevused on valdavalt keerukusega** $O(1)$**.**

1.2 O(n)

Tihtipeale aga on vaja käsitleda märkimisväärse, tihti muudetava suurusega andmehulka. Lihtsaim näide on on ilmselt sisendist järjendi lugemine:

```
n = int(input())                        # O(1)

a = list(map(int, input().split()))     # O(n)
# või
a = [int(x) for x in input().split()] # O(n)
# või
a = []                                  # O(1)
for x in input().split():               # O(n)
    a.append(int(x))                    # O(1)
# või muu sarnane
```

Siin tekib kas *varjatud* või silmnähtav tsükkel, mis käivitub n korda. Iga loetud elemendi töötlemist loeme kui $O(1)$ tegevust, sellisel juhul on justkui muutuv arv $O(1)$ koodilõike kokku kleebitud. Seni kasutatud märkimisviisis pole selle tuletamiseks head võimalust, kuid võime enamvähem väita $n \cdot O(1) = O(n \cdot 1) = O(n)$. Praktikas, kui koodilõiku keerukusega $O(x)$ käivitatakse n korda, on keerukus $O(n \cdot x)$, ehk keerukuste korrutis. Seejuures keerukuse leiame algul tsükli (ning ka funktsiooni) sees ning alles seejärel vaatame tsüklit kui kordajat.

Samas oleme nüüd olukorras, kus on kõrvuti kirjutatud $O(n)$ ning $O(1)$ keerukusega lõigud. On õige väita, et selline kood on $O(n+1)$, kuid selgub, et võime alles jätta vaid kiireimini kasvavad liikmed, seega $O(n+1) \sim O(n)$ (tähistus erineb, aga märgime nii, et need on samad keerukused). Samamoodi võime ära jätta konstandiga korrutamise, seeega $O(5n) \sim O(n)$ ning $O(11) \sim O(1)$.

Märkus: funktsiooni kasvamise kiirust me hästi ei defineeri, kuid praktikas piisab võrdluseks „Kui anda funktsioonidele suur argument, siis milline funktsioon annab suurima vastuse?“

Kokkuvõtteks:

- **Lineaarne aeg** – $O(n)$
- **Konstandiga korrutamise ning aeglasemalt kasvavad liikmed saame minema visata** – $O(5n+3) \sim O(n)$

1.3 Keerukamad keerukused

Eelneva põhjal võiksime saada suurema astmega keerukusi, pannes näiteks $O(n)$ keerukusega koodilõigu tsüklisse, mida käivitatakse omakorda n korda:

```
n = int(input())        # O(1)
x = 0                   # O(1)
for i in range(n):      # O(n**2) (O(n * n))
    for j in range(n):  # O(n)
        x += i * j * j  # O(1) (*)
print(x)                # O(1)
```

Kogu koodilõigu keerukus on $O(n^2)$, kui n on väike. Kui n on suurem, tekivad juba probleemid suurte täisarvudega arvutamisel, nii et keerukus on pisut kehvem ($O(n^2 \log n)$?). Samas võiksime lisada ka teise muutuja:

```
n = int(input())          # O(1)
m = int(input())          # O(1)
x = 0                     # O(1)
for i in range(n):        # O(n**2) (O(n * n))
    for j in range(n):    # O(n)
        x += i * j * j    # O(1) (*)
print(x)                  # O(1)

y = 0                     # O(1)
for i in range(n):        # O(nm) (O(n * m))
    for j in range(m):    # O(m)
        y += i * j * j    # O(1) (*)
print(y)                  # O(1)
```

Nii saime teise koodilõigu keerukusega $O(nm)$, kuid esimese koodilõigu keerukus on $O(n^2)$. Selgub, et sõltuvalt n ja m valikust võib kumbki neist teisest kiiremini kasvada, seega peaksid mõlemad lõplikus keerukuses kajastuma – kogukeerukus on $O(n^2 + nm)$. Näiteks keerukuse $O(n^2 + nm + m^2)$ puhul aga saame analüüsiga leida, et nm tegur pole vajalik, ning kirjutada $O(n^2 + m^2)$.

Toodu põhjal ei tohiks olla kuigi raske konstrueerida funktsioone, mille keerukus on $O(n^3)$ või mõni suurem aste. Samas on olemas palju teisigi keerukusi, näiteks on võimalik saavutada $O(2^d)$, kui programm käivitab kaks korda funktsiooni, mis käivitab iga kord kaks korda funktsiooni, mis käivitab ..., kuni sügavuseni d. Seejuures 2^n kasvab nii kiiresti, et iga n aste jääb sellele analüüsis alla: $O(n^2 + n^5 + 2^n) \sim O(2^n)$.

Kokkuvõtteks:

- **Suuremad keerukused: väiksem keerukus tsüklis või funktsiooni alamprotseduurina (rekursioon?)**
- **Suurusi, millest keerukus sõltub, võib olla mitu:** $O(nm)$**,** $O(n^2p + q)$**, ...**

1.4 ...milleks?

Mingi (ehkki võib-olla hägune) teooria võiks nüüd olemas olla, kuid üsna mõttetu on seda pähe tuupida, kui sellel rakendust pole. Käivitame järgmise koodi:

```
n = 10000

x = 0
for i in range(n):
    for j in range(n):
        x += i * j * j
print(x)
```

Tulemusest pole suurt lugu, meid huvitab rohkem, kaua aega selle programmi täitmiseks kulub. See programm juhtus ühel korral jooksma pea täpselt 1 sekundi. Oletame aga nüüd, et tegemist

on ülesandega, ning n oleks hoopis 100000 ehk 10^5 – 10 korda suurem. Kas selline lahendus töötaks, kui ajalimiit oleks 5 sekundit? 30 sekundit?

Katsetamine näitab, et sama masina puhul jooksis uus kood 180 sekundit. Seega lahendus ühtegi mõistlikku ajalimiiti nii pigem ära ei mahu. Ehkki ka aeglane lahendus võib kasuks tulla, on võimalik isegi enne koodi kirjutamist hinnata, kui aeglane kood olla võiks. Selleks tuleks sooritada keerukuse analüüs:

```
n = 10000               # O(1), aga jätame n muutujana

x = 0                   # O(1)
for i in range(n):      # O(n**2)
    for j in range(n):  # O(n)
        x += i * j * j  # O(1) (*)
print(x)                # O(1)
```

Näeme, et keerukus on $O(n^2)$. Seega, saame hinnata, et koodil kuluks $n = 10^4$ puhul $(10^4)^2 = 10^8$ ajaühikut, $n = 10^5$ puhul aga $(10^5)^2 = 10^{10}$ ajaühikut, seega 100 korda rohkem aega. Sefa, et 100 korda rohkem aega kuluks, saame ka tegelikult otse sisendaevude suhtest: n kasvas 10 korda, nii et aeg kasvas $10^2 = 100$ korda. Tasub märgata, et hinnang on pea kaks korda mööda, kuid saab siiski suurusjärgu paika – enamasti huvitabki meid suurusjärk. (Ilmselt tekib siin probleem suurte täisarvudega askeldamisega, mis suurendab keerukust.)

Lubasime aga keerukust koodi välja kirjutamata – tegelikult saab valdava osa analüüsist teha ära juba idee tasemel, mõeldes, mis keerukusega algoritme ning milliseid tsükleid kasutatakse.

Siiski vajas selline lähenemine korra koodi jooksutamist, et määrata mingi sisendi suuruse puhul kuluv aeg. Tegelikult saame sisse tuua veel ühe hinnangu: **arvuti suudab ühes sekundis sooritada suurusjärgus 10^7 kuni 10^8 lihtsat, kohati ka 10^9 väga lihtsat (liitmine...) operatsiooni**. Need suurused võivad ajapikku, võistluskeskkonniti ja programmeerimiskeeleti kõikuda, kuid see on valdava osa lahenduste jaoks hea hinnang. Kui võtame operatsioonide arvuks $O(...)$ parameetri, saame juba üsna hästi teada, kas lahendus võiks töötada – antud juhul 10^8 operatsiooni on piiripealne, 10^{10} aga 100 korda üle selle, et ühte sekundisse mahtuda.

Siit saame luua ka tabeli – kui suur võib sisend mingi lahenduse keerukuse puhul olla. Tasub märkida, et need suurused on kõik ligikaudsed ning kõiguvad palju.

Keerukus	$n \leqslant ...$
$O(n)$	$10^6,\ (10^8)$
$O(n \log n)$	$4 \cdot 10^5$
$O(n^2)$	$10^3,\ (4 \cdot 10^3)$
$O(n^3)$	10^2
$O(2^n)$	24
$O(n!)$	11

Sama keerukusega lahenduse käivitusaja vähendamist nimetatakse kohati *konstantoptimeerimiseks* (inglise keelest *constant optimization*), enamasti aga nõuavad ülesanded peamiselt hea algoritmi keerukuse saavutamist ning vaid mõningast tuunimist, et ajalimiiti mahtuda.

Eri andmestruktuuride ja algoritmide juurde on enamasti märgitud nende keerukused – neid hästi teades saab üsna hea pildi, mida antud ülesandes teha saab või ei saa.

1.5 Näiteid

Naaseme algul toodud koodilõikude juurde. Lõikude A, B, C keerukused on vastavalt $O(1)$, $O(n^2)$ ning $O(n)$. Toome aga uuesti välja näite D, seekord keerukustega:

```
# D
n = int(input())    # O(1)
a = []              # O(1)
for i in range(n): # O(n * |a|) -> O(n**2)
    a.append(i**2) # O(1)
    print(a)       # O(|a|) (!)
```

D puhul on tegemist kurjema juhtumiga – näiliselt on keerukus $O(n)$, kuid tegelik keerukus on $O(n^2)$. Siin on tegemist *varjatud* tsükliga: `print` funktsiooni täitmiseks tuleb iga element järjendist väljastada, seega tuleb elemente eraldi töödelda. Järelikult kulub selle funktsiooni täitmiseks aega $O(|a|)$, kus $|a|$ on järjendi a suurus. Märgates, et $|a|$ on keskmiselt enam-vähem $\frac{n}{2}$, on tsükli keerukus $O(n \cdot \frac{n}{2}) \sim O(n^2)$.

Vaatame ka teist kurikuulsat komistuskivi: osajadade miinimumi ledmist. Olgu ülesande sisu midagi järgmist:

„Antud on massiiv a, milles on n elementi: $a_0, a_1, ..., a_{n-1}$ $(0 \leqslant a_i \leqslant 10^9)$. Esitatakse q päringut kujul `l r` $(0 \leqslant l \leqslant r < n)$, mille vastuseks tuleb esitada osajada $a_l, a_{l+1}, ..., a_r$ vähim element. $n \leqslant 10^5$, $q \leqslant 10^5$, aega on 1 sekund."

Naiivne lahendus, mis kohe pähe võiks hüpata, on järgnev:

```
n = ...
q = ...
a = [... for _ in range(n)]      # O(n)

for qi in range(q):              # O(q * (r - l)) -> O(qn)
    l, r = ...                   # O(1)

    mi = 10**18                  # O(1)
    for i in range(l, r + 1):    # O((r - l + 1) * 1) = O(r - l)
        mi = min(mi, a[i])       # O(1)

    print(mi)                    # O(1)
```

Näeme, et igale päringule vastamiseks kulub $O(r-l)$ aega, kuid eeldusel, et testid on korralikult genereeritud, leidub teste, kus $r - l$ on pidevalt võimalikult suur väärtus, ehk vaadata tuleb (peaaegu) kõiki elemente. Seega võime võtta selle asemele pessimistlikuma hinnangu $O(n)$, millele teame juba ülempiiri. Kokkuvõttes saame nii keerukuseks $O(nq)$, mis maksimaalseid suuruseid sisse asendades annab $10^5 \cdot 10^5 = 10^{10}$ operatsiooni, mis pigem ei ole sekundisse mahutatav.

Ülesande lahendamiseks tuleks tutvuda andmestruktuuride (lõikude puu, hõre tabel) või teiste lahendusmeetoditega (ruutjuur-dekompositsioon).

2 Logaritm

Tähelepanelikule jälgijale jäi silma mainimata keerukustest $O(\log n)$ ning mainitud $O(n \log n)$. Matemaatiliselt võib logartimi tekkimine tunduda üsna veider, kuid selle tekkimisele leidub üsna hea põhjendus.

2.1 Kahendotsing

Kahendotsing on „jaga ja valitse“ (enam-vähem) tüüpi algoritmide musternäide, näidates ühtlasi struktureeritud andmetest saadavat kasu. Vaatame näidet, kus meil on vaja antud mittekahanevalt järjestatud järjendist a leida element 17 (mis asub vähimal indeksil):

$$a = [3, 5, 7, 7, 11, 13, 13, 17, 19, 23, 29, 31]$$

Programmeerimiskeeltes mõningaselt vilunud võivad pakkuda selle lahendamiseks `a.index(17)` või `std::find(a.begin(), a.end(), 17)`, kuid nende funktsioonide keskmine keerukus on $O(n)$, kus $n = |a|$. Sel on ka hea põhjus – need funktsioonid ei eelda, et järjend oleks järjestatud.

Teostame kahendotsingu, kus igal sammul jaotame vaadeldava järjendi osa kaheks, millest otsustame, kumba edasi kontrollida. Algul vaatame kogu järjendit, mille jaotame võimalikult võrdselt pooleks (jätame vajadusel rohkem paremale):

$$[3, 5, 7, 7, 11, \mathbf{13}, 13, 17, 19, 23, 29, 31]$$

Nüüd vaatame, kas vasakpoolse lõigu viimane (suurim) element on otsitavast arvust 17 suurem või sellega võrdne. Antud juhul 13 seda pole, seega pole ka ükski temast vasakul asuv element piisavalt suur. Seega peame uurima edasi ainult teist järjendi poolt, kus rakendame sama taktikat

$$[3, 5, 7, 7, 11, 13, 13, 17, \mathbf{19}, 23, 29, 31]$$

Seekord on $19 \geqslant 17$, seega on kõik sobivad (vähima indeksiga) kandidaadid just vasakus harus, parempoolset võime ignoreerida:

$$[3, 5, 7, 7, 11, 13, \mathbf{13}, 17, 19, 23, 29, 31]$$

Nüüd $13 < 17$, seega jätkame parempoolses harus:

$$[3, 5, 7, 7, 11, 13, 13, \mathbf{17}, 19, 23, 29, 31]$$

Et $17 \leqslant 17$, jätkame vasakpoolse haruga. Nii aga on alles jäänud vaid üks element – viisakas on üle kontrollida, kas see on ka otsitud element (on võimalik, et alles jääb üks element ka nii, et seda pole kunagi otsitava vastu kontrollitud). Antud juhul on seal tõepoolest 17, nii et leidsime, et vähim sobiv indeks on 7. Seejuures on märkimisväärne, et tehtud sai vaid 5 võrdlust, ehk vähem, kui on elemente 17-ni. Kas nii leidmine on aga alati kiire?

Otsimise peamine mõte oli, et otsinguala muutub iga kord ligikaudu kaks korda väiksemaks. Kui järjendi pikkus pole kahe aste, võib nii otsides ala muutuda ka pisut vähem väiksemaks, kuid seda võime praktiliselt ignoreerida (või alustada järjendist, mille pikkus on vähim kahe aste, mis on vähemalt n). Seega on vaja vastust järgnevale küsimusele: **„Mitu korda peab arvu n poolitama, et järelejäänud arv oleks 1?“** (arvu asemel on päriselt osajärjendi pikkus).

Kui eeldame, et $n = 2^m$, on vastus üsna ilmne – tegemist on peaaegu kahendlogaritmi definitsiooniga. Teisisõnu on vaja $\log_2 n$ korda poolitada. Praktikas töötab sama valem (üles ümardades) ka arvude puhul, mis pole kahe astmed. Järelikult on kahendotsingu keerukus $O(\log_2 n)$.

Logaritmilise keerukusega algoritmi ajavõitu ei tasu alahinnata. Kahendlogaritm on programmeerimises nii levinud, et tihtipeale asendatakse $\log_2 x$ lihtsalt kirjutisega $\log x$ (lisaks selgub, et keerukushinnangus pole logaritmi alusest vahet). Kasutame ka teadmist, et $10^3 = 1000 \approx 1024 = 2^{10}$, seega $\log 10^3 \approx \log 2^{10} = 10$. Sellisel juhul võtab kahendotsing suurest massiivist pikkusega 10^6 kõigest $\log 10^6 = \log(10^3)^2 \approx 2 \cdot 10 = 20$ võrdlust. Kui aga nõutakse leida mingit kahendotsitavat tingimust rahuldav arv vahemikust $[0, 10^{18})$, siis on vaja $\log 10^{18} = \log(10^3)^6 \approx 6 \cdot 10 = 60$ tingimuse kontrolli. See annab voli kasutada lahendamisel näiteks vastuse kahendotsimise taktikat, kui on vaja leida suurim / vähim võimalik väärtus ning on võimalik kontrollida, kas vähemalt nii väike / suur vastus leidub.

Viimaks – sorteeritud järjendist väärtuse otsimiseks on Pythonis ja C++-is täiesti võimekus olemas, vastavalt `bisect` paketi ning `std::lower_bound` (ja seotud funktsioonide) näol (lisaks vaadata kogu `algorithms` teek).

2.2 Logaritmi kasutusalad

Logaritmi võib leida mitmete andmestruktuuride keerukustes, mis kasutavad mõnd sorti (tasakaalustatud) puud (kahendpuud) andmete hoidmiseks. Üks märkimisväärsemaid keerukusi on aga sorteerimisel – iga endast lugu pidav sorteerimisalgoritm on halvima juhu keerukusega $O(n \log n)$ ($O(n)$ keerukust lubavad sorteerimisalgoritmid üritavad valdavalt $\log n$ osa eri viisidel konstanti peita), mis tähendab, et järjendi sorteerimine on minimaalse ajakuluga tegevus, ent, nagu oli näha näiteks kahendotsingu algoritmist, võimaldab äärmiselt suuri ajalisi võite.

3 Lahenduse optimeerimine

Kui esialgne analüüs näitab, et mälja mõeldud lahendus ajaraamidesse ei mahu (ning ehk on ka server „Ajalimiit ületatud“ vastanud), tasub mõelda algoritmi ümber töötamisele. Tihtipeale saab korduvat tööd seda eir järjekorras või teise lähenemisega tehes väga palju vähendada: näiteks võib mitmekordse lahutamise asemel korra jagada või pika järjendi läbi vaatamise asemel läbi vaadata ainult hädavajaliku osa.

Mitmeid optimeerimismeetodeid iseloomustab algul (või lõpus) mõningase lisatöö tegemine, et hiljem (või algul) mõnd ülesande osa palju kiiremini lahendada. Järgnevalt kirjeldame üht lihtsaimat, kuid suure potentsiaaliga optimeerimismeetodit.

3.1 Prefikssumma

Idee on lihtne: arvutame algse järjendi põhjal uue järjendi, kus indeksil i on summa elementidest indeksitel 0 kuni i. Ilmselgelt saame peale seda $O(1)$ ajas leida, mis on mingi prefiksi (indeksilt 0 algava osajärjendi) summa. Samas aga võime ka märgata järgnevat fakti suvalise indeksite lõigu $0 \leqslant l \leqslant r < n$ kohta:

Alamjärjendi algusindeksiga l ning lõpuindeksiga r elementide summa ning alamjärjendi algusindeksiga 0 ning lõpuindeksiga $l-1$ elementide summa on võrdne sellise alamjärjendi summaga, mille algusindeks on 0 ning lõpuindeks r.

Või ehk sama kuivalt matemaatiliselt:

$$\sum_{i=0}^{l-1} a_i + \sum_{i=l}^{r} a_i = \sum_{i=0}^{r} a_i$$

Mida see praktikas tähendab: kui me võtame lõigu indeksist l indeksini r ning teise lõigu, mis ulatub järjendi algusest selle lõiguni, siis need lõigud kokku on sama, kui lõik algusest indeksini r. Lisaks muidugi kasutasime eelnevalt summat.

Nüüd võib märgata, et me teame *iga* sellise lõigu summat, mis algab indeksist 0. Kui see lõik on tühi, siis on summa 0. Seega, kui natuke matemaatilist kuju ümber avaldada:

$$\sum_{i=l}^{r} a_i = \sum_{i=0}^{r} a_i - \sum_{i=0}^{l-1} a_i$$

Järelikult saame me nüüd leida *suvalise* lõigu summa, kasutades kõigest kahte prefikssumma väärtust, ehk $O(1)$ ajas. See on märkimisväärselt parem kui $O(r-l)$ (mis on tihtipeale $O(n)$) keerukusega naiivselt summa leidmine.

Samas tasub märgata, et sel lähenemisel on ka piir: liitmise puhul nõudsime, et olemas on ka lahutamine, mis on üsna mõistlik. Samas, tahtes rakendada analoogset algoritmi teistele seostele, võib juhtuda, et ei leidu ühest võimalust „mingit summat maha lahutada“ (formaalselt operatsiooni *pöörata*). Nii juhtub näiteks, kui tahta hoopis korrutist ning abiks võetud lõikudesse satub arv 0. Sarnane probleem tekib, kui on vaja leida osajärjendi miinimumi või maksimumi.

3.2 Prefikssumma duaal

Kui me saime suvalise alamjärjendi summa $O(1)$ ajas leida, võib mõttesse kippuda küsimus, kas nii saaks ka suvalist alamjärjendit $O(1)$ ajas *muuta*. Selgub, et vähemalt mõnd tüüpi operatsioonide puhul on see täiesti võimalik.

Olgu meie (ala-) ülesandeks liita $O(1)$ ajas mingile lõigu indeksitega l kuni r elementidele väärtus k (igaühele), ning peale selle operatsiooni mingi hulk kordi kordamist väljastada $O(n)$ ajas tulemus. Naiivse lähenemisega saaksime $O(r-l)$ lahenduse liitmiseks, kuid saame ka paremini.

Kujutame ette järjendit, kuhu salvestame, mis väärtus tuleb igale algsele järjendi väärtusele liita, et saada lõplik järjend. Ka selle järjendi naiivne tekitamine võtaks iga liitmise jaoks $O(r-l)$ aega, kuid võime seada ühe huvitava tingimuse: genereerime selle järjendi peale liitmisi prefikssumma abil. Kas me saame üldse sellise algse järjendi moodustada ning kui kulukas see on?

Nagu selgub, siis veelgi uuema järjendi pidamine on ootamatult lihtne. Eeldame hetkeks, et me tõepoolest saame ülesande nii lahendada. Sellisel juhul, kui tahame lõigule l kuni r lisada väärtuse k, ei tohiks me muuta väärtusi enne indeksit l. Samas peaksime indeksile l lisama väärtuse k, et sealne väärtus k võrra kasvaks. Peaksime garanteerima ka, et igale järgnevale indeksile (kuni r) seesama väärtus liidetaks, kuid prefikssumma definitsiooni kohaselt kandub liidetud väärtus juba edasi, seega seda uuesti liitma ei pea! Samas aga muutuksid nii elemendid peale indeksit r. Seda on lihtne parandada – tühistame varem liidetud k mõju, liites indeksile $r + 1$ väärtuse $-k$. Kokkuvõttes sellel indeksil ning edasi on muudatuse mõju 0. Nii saigi järjend (peale prefikssumma rakendamist) soovitud moel muudetud, muutes kõigest kahel indeksil olevat väärtust.

Toodud ülesande lahendamiseks piisab seega liitmistehteid realiseerida prefikssumma eelsel järjendil, peale liitmist leida prefikssumma ning liita muudatused algse järjendi väärtustele.