Математичка гимназија

МАТУРСКИ РАД

- Програмирање и програмски језици -

Пробабилистички алгоритми у такмичарском програмирању

Ученик:
Богдан Петровић IVe

Ментор:
Никола Тасић

Београд, јун 2023.

Овом приликом бих се захвалио свом ментору Николи Тасићу на издвојеном времену и корисним саветима током консултација и на часовима рачунарства. Такође бих се захвалио свим професорима који су ми предавали и својим трудом допринели мом даљем развоју.

Садржај

1 Увод

Иако су детерминистички алгоритми углавном довољно ефикасни, постоје одређени проблеми који су преспори или чија имплементација меморијски, или из неких других разлога, није задовољавајућа. Такве проблеме можемо решити употребом рандомизације. Рандомизацијом можемо постићи да се време извршавања програма драстично смањи, или да програм троши мање меморије.

Пробабилистички или рандомизирани алгоритам је алгоритам који у својој логици користи одређени степен случајности. Такав алгоритам обично користи униформно распоређене насумичне бројеве. У пракси то се постиже коришћењем псеудонасумичног генератора бројева. Таква имплементација може одступати од теоријског очекиваног понашања, али то је у већини нама корисних случајева занемарљиво.

Како су проблеми које решавамо рачунарима све сложенији, у протекле две деценије су пробабилистички алгоритми доживели велику експанзију. Ови алгоритми имају велику примену у много индустрија јер убрзавају класичне, детерминистичке алгоритме са вероватноћом тачности која је нама прихватљива. Понекад коришћење пробабилистичких алгоритама не доводи до побољшања у ефикасности рада програма у поређену са детерминистичким, али су оваква решења углавном једноставнија за разумевање и имплементацију па је вредно решавати проблеме на тај начин.

Пробабилистичке алгоритме можемо поделити у три категорије. Монте Карло алгоритми су алгоритми који тачан резултат дају са одређеном вероватноћом, али су увек брзи. Лас Вегас алгоритми су алгоритми који су брзи са одређеном вероватноћом али увек дају тачан резултат. Атлантик Сити алгоритми су на неки начин комбинација претходна два, они и тачност и сложеност алгоритма дају са одређеним вероватноћама. У овом раду, акценат ћемо ставити на најраспрострањеније алгоритме, конкретно Монте Карло и Лас Вегас алгоритме, у задацима са програмерских такмичења.

2 Елементи теорије вероватноће

Ако је скуп $\Omega = \{\omega_1, \omega_2, ..., \omega_n\}$ коначан простор елементарних догађаја неког експеримента, онда се сваки подскуп A скупа Ω зове *случајан догађај* или кратко догађај. За исход ω, тј. елементарни догађај ω, за који важи $\omega \in A \subset \Omega$, кажемо да је повољан исход за догађај A.
Ω још зовемо и сигуран догађај, скуп исхода. $\emptyset$ је немогућ догађај. Догађај $\overline{A} = \Omega \setminus A$ је догађај комплементаран догађају A. $A \cup B$ је унија догађаја A и B, док је $A \cap B$, или скраћено AB, пресек догађаја A и B. Догађаји A и B су дисјунктни, ако важи $AB = \emptyset$.

Вероватноћу неког догађаја A обележавамо са $P(A)$ и важи:

- $P(A) \geq 0$
- $P(\Omega) = 1$
- $P(\overline{A}) = 1 - P(A)$
- $P(\emptyset) = 0$
- За сваки низ (коначан или пребројив) догађаја који су међусобно дисјунктни у паровима важи: $P(\bigcup_i A_i) = \sum_i P(A_i)$

Неке особине вероватноће су:

- $P(A \cup B) = P(A) + P(B) - P(AB)$
- $P(AB) = P(A)P(B)$, ако су догађаји A и B независни.
- $P(A) < P(B)$, ако је $A \subseteq B$.
- $P(\bigcup_i A_i) \leq \sum_i P(A_i)$, ако је $A_1, A_2, ...A_k$ коначан или пребројив низ догађаја.

Пресликавање $X : \Omega \to \mathbb{R}$ задато са $X(\omega) = 1$, за $\omega \in A$ и $X(\omega) = 0$, за $\omega \in \overline{A}$ је случајна променљива, индикатор случајног догађаја A.

$$I_A : \begin{pmatrix} 0 & 1 \\ P(\overline{A}) & P(A) \end{pmatrix}$$

Ако је X дискретна случајна величина са коначним скупом вредности, чија је расподела дата са:

$$X : \begin{pmatrix} x_1 & x_2 & \cdots & x_m \\ p_1 & p_2 & \cdots & p_m \end{pmatrix}$$

онда се математичко очекивање случајне величине X дефинише са:

$$E(X) = \sum_{i=1}^{m} x_i p_i$$

Ако су X и Y случајне величине дефинисане на истом простору вероватноћа и имају коначна математичка очекивања, а a и b су реални бројеви, онда важи:

$$E(aX + bY) = aE(X) + bE(Y)$$

Ова особина назива се линеарност очекиване вредности.
Можемо приметити да је очекивана вредност индикатора догађаја A једнака $P(A)$.

Условна вероватноћа догађаја A у односу на догађај B дефинише се као вероватноћа да је испуњен догађај A ако је испуњен догађај B. Записује се као $P(A|B)$.

$$P(A|B) = \frac{P(AB)}{P(B)}$$

При рачуну са условном веровантноћом често се користи Бајесова формула. За коначно много дисјунктних догађаја A_i, $A_1 \cup A_2 \cup ... \cup A_n = \Omega$:

$$P(A_i|B) = \frac{P(B|A_i)P(A_i)}{\sum_{j=1}^{n} P(B|A_j)P(A_j)} = \frac{P(B|A_i)P(A_i)}{P(B)}$$

Бајесова формула за два случаја A и B своди се на:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

3 Брзо сортирање и брзо селектовање

У овом поглављу ћемо се бавити најкориснијим пробабилистичким алгоритмима у такмичарском програмирању, брзим сортирањем (енг. *quicksort*) и брзим селектовањем (енг. *quickselect*). Брзо сортирање служи за сортирање елемената низа у неки поредак, док брзо селектовање одговара на питање који елемент би се налазио на k-тој позицији у сортираном низу? Из тог разлога ови алгоритми су уско повезани и заснивају се на истим принципима.

3.1 Брзо сортирање

Размотримо идеју сортирања скупа S у неком редоследу. Постоје разни детерминистички алгоритми који решавају овај проблем. Најједноставнији такав алгоритам зове се Бабл сорт, који ради у сложености $O(N^2)$. Неки од напреднијих алгоритама који постижу временску сложеност $O(NlogN)$ су Сортирање спајањем и Хип сорт. Постоје поједини алгоритми који постижу временску сложеност $O(N)$ са незанемарљивим константним фактором, али су њихове примене углавном ограничене на изузетно специјалне случајеве.
Алгоритам брзог сортирања предложио је Британски научник Тони Хор 1959. године. Његов алгоритам постиже просечну временску сложеност $O(NlogN)$, као и горе наведени алгоритми, али се у практичној примени показао као најбржи, поготово на великим скуповима података. То је алгоритам типа подели па владај и не захтева додатну меморију, тј. све ради у месту.

3.1.1 Опис рада алгоритма

Ради лакшег разумевања и доказа сложености алгоритма, претпоставимо да не постоје две исте вредности елемената у низу.
Нека је елемент на позицији i у неком низу означен са a_i. Уколико би сви елементи са његове леве стране били мањи од њега, а сви елементи са десне стране већи, онда би се овај елемент налазио на оној позицији на којој би био и у сортираном низу. Тај елемент ћемо у даљем тексту звати пивот.
Да бисмо сортирали низ користићемо дату идеју на следећи начин. Случајно бирамо пивот елемент. Поредимо га са осталим елементима и одређујемо која би била његова позиција у сортираном низу. Остале елементе постављамо на позиције лево, односно десно у односу на позицију пивот елемента у зависности од поређења вредности пивот елемената са вредностима осталих елемената. Тиме смо постигли одговарајућу позицију пивот елемента. Понављамо исти поступак за подниз лево и десно од пивота, за сваки подниз дужине веће од 1.

3.1.2 Бирање пивота и партиција

Уколико је дужина низа мања од 2, низ је сортиран, па се враћамо из позива партиције.
Нека је низ дужине N индексиран од 0. Пивот елемент се бира насумично, униформно, користећи псеудонасумичан генератор бројева. Размењују се вредности пивота и последњег елемента низа. Бројач b се поставља на вредност 0 и понавља се следећи процес за свако i од 0 до $N-2$.
Уколико важи $a_i < a_{N-1}$, размењују се вредности a_b и a_i и повећава се вредност бројача b за 1.
Након тога, размењују се вредности a_b и a_{N-2}. Тиме је завршен процес бирања пивота и партиције. Пивот се налази после свих елемената са мањом или једнаком вредношћу, а пре свих елемената са већом вреднош ћу. Тиме се постиже да је пивот на својој позицији у сортираном низу.

3.1.3 Подели па владај

Након партиције низа, рекурзивно позивамо исти алгоритам бирања пивота и партиције за подниз који чине елементи лево од пивота и за подниз који чине елементи десно од пивота.

3.1.4 Анализа временске сложености

Алгоритам партиције низа дужине N има сложеност $O(N)$, јер се сваки елемент низа пореди и размењује највише једном.
Означимо са $T(N)$ временску сложеност алгоритма брзог сортирања низа дужине N.

Када је елемент који је изабран као пивот на првој или последњој позицији у низу, може се показати да рекурентна једначина сложености има овакву форму:

$$T(N) = T(N-1) + O(N)$$

из тога добијамо:

$$T(N) = O(N^2)$$

Када је елемент који је изабран као пивот медијана у низу, може се показати да рекурентна једначина сложености има следећу форму:

$$T(N) = 2T\left(\frac{N-1}{2}\right) + O(N)$$

из тога добијамо:

$$T(N) = O(NlogN)$$

Просечан случај анализирамо на следећи начин:

Теорема 1. *Очекивани број поређења елемената у рандомизираном брзом сортирању је највише* $2NlnN$.

Доказ. Величину која нас занима (укупан број поређења), представићемо као суму једноставнијих, случајних величина и онда њих појединачно анализирати.
Величина X_{ij} узима вредност 1 уколико алгоритам пореди i-ти и j-ти најмањи елемент, у супротном узима вредност 0. Нека је X укупан број поређења. Како никад не поредимо неки пар индекса више од једанпут имамо:

$$X = \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} X_{ij}$$

$$E[X] = E[\sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} X_{ij}]$$

због линеарности очекиване вредности:

$$E[X] = \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} E[X_{ij}]$$

Размотримо X_{ij} када је $i < j$. Замислимо да је низ већ сортиран. Нека је a_i i-ти најмањи елемент у низу и a_j j-ти најмањи елемент у низу. Ако је индекс пивота p и $i < p < j$, a_i и a_j се неће поредити. Ако је $p < i$ или $p > j$, a_i и a_j ће бити у истом низу у следећем алгоритму партиције, па поново бирамо пивот. Ако је $p = i$ или $p = j$, ти елементи се пореде.
Можемо замислити овај процес као игру пикада. Бацамо стрелицу униформно, насумично у низ. Ако погодимо лево од i или десно од j бацамо поново. Ако погодимо индексе i или j, X_{ij} постаје 1 и игра се завршава. Ако погодимо између та два елемента, X_{ij} постаје 0 и игра се завршава. У сваком кораку вероватноћа да је $X_{ij} = 1$ једнака је $\frac{2}{j-i+1}$.
Другим речима, елемент са индексом i поредиће се са елементом на $i+1$ са вероватноћом 1, са елементом на $i+2$ са вероватноћом $\frac{2}{3}$, са елементом на $i+3$ са вероватноћом $\frac{2}{4}$...
На основу претходног закључка можемо извести израз за очекивану вредност:

$$E[X] = \sum_{i=0}^{N-1} 2(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{N-i})$$

Величина $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{N}$, означена са H_N, јесте N-ти хармонијски број и налази се у интервалу $[lnN, lnN + 1]$. Из тога следи:

$$E[X] < 2N(H_N - 1) \leq 2NlnN$$

Како је укупан број поређења директно пропорционалан времену извршавања алгоритма, добија се просечна временска сложеност брзог сортирања $O(NlogN)$ □

3.2 Брзо селектовање

Размотримо идеју налажења k-тог најмањег елемента у низу дужине N. Очигледно решење било би да за сваки елемент прођемо кроз низ и бројимо колико елемената има мању вредност од њега што се може постићи у временској сложености $O(N^2)$. Боље решење било би да сортирамо низ

неким брзим алгоритмом, нпр. брзим сортирањем и онда само издвојимо k-ти елемент. Овај приступ захтева сортирање које ради у сложености $O(NlogN)$.
Алгоритам брзог селектовања постиже просечну сложеност $O(N)$, док у најгорем случају ради у $O(N^2)$.

3.2.1 Опис рада алгоритма

Као и код брзог сортирања, ради лакшег разумевања и анализе претпоставимо да нема истих вредности елемената у низу.
Користимо исту идеју о пивоту као и код брзог сортирања. Да бисмо нашли k-ти најмањи елемент користимо ту идеју на следећ начин. Случајно бирамо пивот елемент. Њега поредимо са осталим елементима и одређујемо позицију на којој би био у сортираном низу. Све елементе који имају мању вредност од пивота премештамо лево, а све елементе који имању већу вредност од пивота премештамо десно од пивота. Тиме смо постигли да се пивот налази на правом месту у сортираном низу.
Ако се пивот после алгоритма партиције налази на позицији k, то је k-ти најмањи елемент. Враћа се вредност пивота и завршава се рад алгоритма.
Ако се пивот налази на позицији $i > k$, k-ти најмањи елемент се сигурно налази лево од пивота, па се понавља алгоритам за све елементе лево од пивота.
Ако се пивот налази на позицији $i < k$, k-ти најмањи елемент се сигурно налази десно од пивота, па се понавља алгоритам за све елементе десно од пивота, са изменом да ће вредност k у следећем извршавању алгоритма партиције умањити за број елемата мањих или једнаких пивоту.

Бирање пивота и партиционисање идентично је као и код алгоритма брзог сортирања, са изменом рекурзивног позивања алгоритма као што је претходно наглашено.

Алгоритам се извршава у месту, па не захтева додатну меморију. Уколико нам је битно да не изменимо поредак елемената у низу, потребно је да направимо нови низ и да на њему извршавамо алгоритам, што захтева додатну линеарну потрошњу меморије.

3.2.2 Анализа временске сложености

Најбољи случај се дешава када је пивот елемент у првој партицији баш k-ти најмањи елемент. Тада се алгоритам партиције извршава једанпут

па је сложеност $O(N)$.

Као и код алгоритма брзог сортирања, најгори случај дешава се када је елемент који је изабран као пивот најмањи или највећи елемент у низу. Може се показати да рекурентна једначина сложености има следећу форму:

$$T(N) = T(N-1) + O(N)$$

из чега добијамо:

$$T(N) = O(N^2)$$

За просечну сложеност, као код алгоритма брзог сортирања, важи рекурентна релација:

$$T(N) = T\left(\frac{N-1}{2}\right) + O(N)$$

$$T(N) = O(N)$$

4 Фингерпринтинг и Фреивалдсова техника

У рачунарству, фингерпринтинг (енг. *fingerprinting*) или техника дигиталних отисака, је техника којом се елементи неког великог скупа мапирају у скуп доста мање кардиналности. Овакве технике користе се када је пренос података скуп или када је пожељно избећи поређење великих података. Нека од стандардних примена је у случају када интернет претраживач или прокси сервер проверава да ли је неки фајл измењен тако што ће само поредити фингерпринт фајла са претходно сачуваном копијом фингерпринта. Фингерпринтинг функције се могу посматрати као хеш функције високих перформанси које се користе за идентификацију блокова података у ситуацији када криптографске хеш функције нису неопходне.
У овом поглављу радићемо на неназначеном пољу $\mathbb{F}$. Део разлога зашто поље није назначено је јер технике које следе углавном захтевају униформно бирање из неког коначног подскупа поља $\mathbb{F}$. Из тога разлога, није битно да ли је то поље коначно или не. Ради лакшег разумевања пожељно је посматрати поље $\mathbb{F}$ као $\mathbb{Q}$, поље рационалних бројева или у случају када је наглашено да је $\mathbb{F}$ коначно можемо га посматрати као $\mathbb{Z}_p$, поље целих бројева по модулу неког простог броја p [1].

4.1 Фреивалдсов алгоритам

Фреивалдсов алгоритам је пробабилистички, рандомизирани алгоритам који се користи за верификацију множења матрица.
Дате су три матрице димензија $N \times N$. Потребно је проверити да ли важи $AB = C$. Наивним детерминистичким алгоритмом долазимо до решења у временској сложености $O(N^3)$, једноставном провером у сложености $O(N)$ за свако од N^2 поља матрице C .
Најбржи детерминистички алгоритам за решавање овог проблема откривен

је 2022. године и ради у временској сложености $O(N^{2.37188})$ [4]. Овај приступ је изузетно компликован, па такав алгоритам није могуће имплементирати у ограниченом временском интервалу као што је такмичење из програмирања.
Фреивалдсов алгоритам користи чињеницу да није потребно израчунати вредност AB, већ само проверити $AB = C$ и решава овај проблем у сложености $O(N^2)$ уз вероватноћу тачности $P \geq 1/2$.

4.1.1 Опис рада алгоритма

Алгоритам бира вектор $r \in \{0,1\}^N$. Сваки од елемената вектора r се бира независно и униформно из скупа $0, 1$, где су 0 и 1 адитивни односно мултипликативни неутрал поља $\mathbb{F}$, респективно. Рачунају се $x = Br$, $y = Ax = ABr$ и $z = Cr$ који се могу добити у сложености $O(N^2)$. Уколико је $y = z$, алгоритам ће вратити да је $AB = C$. У супротном алгоритам ће вратити да $AB \neq C$. Алгоритам се понавља произвољан број пута независним генерисањем вектора r при свакој итерацији како би се побољшала тачност.

```
1  bool freivalds(mat A, mat B, mat C, int N) {
2      mat r = generisi(N);
3      mat x = pomnozi(B,r);
4      mat y = pomnozi(A,x);
5      mat z = pomnozi(C,r);
6      return isti(y,z);
7  }
8  bool proveriProizvod(mat A, mat B, mat C, int N, int k) {
9      bool tacno = true;
10     while(k--)
11         tacno |= freivalds(A, B, C, N);
12     return tacno;
13 }
```

Слика 1: $C++$ код Фреивалдсовог алгоритма

4.1.2 Анализа тачности

Једноставно је закључити да $AB = C \Rightarrow y = z$. Потребно је доказати да у случају $AB \neq C$, $P(y \neq z) \geq \frac{1}{2}$.

Лема. *Одабир вектора* $r = (r_0, r_1, ..., r_{N-1})$ *униформно случајно је еквивалентно одабиру сваког* r_i *независно и униформно из* $\{0, 1\}$.

Доказ. Ако је сваки од r_i изабран независно и униформно, тада је сваки од 2^N могућих вектора изабран са вероватноћом 2^{-N}. □

Теорема 2. *Нека су A, B и C матрице димензија $N \times N$ над пољем $\mathbb{F}$. Тада за униформно изабран вектор r из $\{0,1\}^N$ важи $P(ABr = Cr) \leq \frac{1}{2}$.*

Доказ. Нека је $D = AB - C \neq 0$. Тада $ABr = Cr$ имплицира $Dr = 0$. Како D ненула матрица, мора имати ненула елемент. Без умањења општости, нека је тај елемент D_{00}.[8] Да би Dr било једнако 0 мора да важи:

$$\sum_{j=0}^{N-1} D_{0j} r_j = 0$$

или еквивалентно:

$$r_0 = -\frac{\sum_{j=1}^{N-1} D_{0j} r_j = 0}{D_{00}}$$

Овде можемо применити принцип одложених одлука [1]. Уместо да разматрамо вектор r, замислимо да бирамо његове координате, r_i, независно и униформно почевши од координате r_{N-1} до координате r_0, редом. По леми, то је еквивалентно као и одабир вектора униформно случајно. Потребно је посматрати тренутак директно пре одабира r_0. У овом тренутку, десна страна горње једначине је одређена и постоји највише једна вредност за r_0 тако да једнакост буде тачна. Како постоје 2 избора за вредност r_0, једнакост ће бити тачна са вероватноћом која је мања или једнака од $\frac{1}{2}$. □

Понављањем Фреивалдсовог алгоритма k пута независним генерисањем вектора r при свакој итерацији вероватноћа грешке је мања или једнака од 2^{-k}.

За извршавање алгоритма потребно је издвојити додатних $O(N)$ меморије за сваки од x, y и z.

4.1.3 Анализа временске сложености

Рачунање матрица x, y и z се извршава у $O(N^2)$. Поређење y и z је такође $O(N^2)$. Из тога следи укупна сложеност целог алгоритма $O(N^2)$. Уколико се алгоритам понавља k пута временска сложеност је $O(kN^2)$.

Фреивалдсову технику можемо применити за верификацију било ког матричног идентитета $X = Y$. Наравно, ако су X и Y експлицитно дати, једноставним поређењим њихових елемената постижемо сложеност $O(N^2)$. У неким случајевима као што је множење матрица, рачунање матрице X је непрактично или чак немогуће, али је рачунање Xr једноставно.

4.2 Верификација полиномских идентитета

Фреивалдсова техника може се користити за разне врсте провера идентитета, као што је на пример, провера идентитета полинома. Провера идентитета полинома је генерализација мноштва случајева, као што су, на пример, провера једнакости целих бројева, или уопштено, провера једнакости стрингова над неким фиксним алфабетом. То постижемо тако што k-ти карактер у стрингу посматрамо као коефицијент k-тог степена симболичне променљиве x. Кажемо да су полиноми $P(x)$ и $Q(x)$ једнаки ако имају исте коефицијенте испред одговарајућих степена x-а.

Размотримо проблем верификације производа полинома. Дати су полиноми $P_1(x)$, $P_2(x)$, $P_3(x) \in \mathbb{F}[x]$. Потребно је проверити да ли важи $P_1(x)P_2(x) = P_3(x)$. Претпоставимо да су полиноми $P_1(x)$ и $P_2(x)$ степена највише n. Према томе, $P_3(x)$ је степена највише $2n$.
Производ $P_1(x) \cdot P_2(x)$ најбржом детерминистичком методом можемо израчунати помоћу брзих Фуријеових трансформација у сложености $O(nlogn)$, док рачунање вредности полинома у фиксној тачки захтева $O(n)$.

4.2.1 Опис рада алгоритма

Идеја за пробабилистички алгоритам слична је идеји за верификацију производа матрица. Нека је $\mathbb{S} \subseteq \mathbb{F}$ скуп кардиналности $\geq 2n + 1$. Бира се $r \in \mathbb{S}$ униформно случајно. Рачунају се $P_1(r)$, $P_2(r)$ и $P_3(r)$ у сложености $O(n)$. Алгоритам враћа да је полиномски идентитет $P_1(x)P_2(x) = P_3(x)$ тачан уколико важи $P_1(r)P_2(r) = P_3(r)$, у супротном враћа да није тачан.

4.2.2 Анализа тачности алгоритма

Дефинишемо полином $Q(x) = P_1(x)tP_2(x) - P_3(x)$ степена $2n$. Кажемо да је полином P идентички једнак нули, $P \equiv 0$, уколико су сви његови коефицијенти једнаки нули. Полином $Q(x) \equiv 0$ акко је производ полинома

тачан. Алгоритам ће вратити нетачан резултат само у случају када је $Q(r) = 0$, а производ полинома није тачан за свако x. Показаћемо да је у случају када је $Q(x) \not\equiv 0$, вероватноћа $p(Q(r) = P_1(r)P_2(r) - P_3(r) \neq 0)$ ограничена одоздо.
Пошто полином степена n може имати највише n различитих нула, уколико $Q(x) \not\equiv 0$, постоји највише $2n$ различитих вредности r таквих да је $Q(r) = 0$. Због тога је вероватноћа грешке ограничена са $\frac{2n}{|\mathbb{S}|}$. Ова вероватноћа се може смањити избором довољно великог скупа $\mathbb{S}$. Такође, могуће је смањити вероватноћу независним понављањем овог алгоритма k пута, чиме се постиже вероватноћа грешке $\left(\frac{2n}{|\mathbb{S}|}\right)^k$.

Уколико су полиноми $P_1(x)$, $P_2(x)$ и $P_3(x)$ дати експлицитно, потребно је издвојити само $O(1)$ меморије за вредности $P_1(r)$, $P_2(r)$, $P_3(r)$ и $P_1(r) \cdot P_2(r) = P_3(r)$.

4.2.3 Анализа временске сложености

Временска сложеност за сваку од евалуација вредности полинома $P_1(r)$, $P_2(r)$ и $P_3(r)$ је $O(n)$, па је сложеност целог алгоритма $O(n)$.
Уколико се алгоритам понавља k пута временска сложеност је $O(kN)$.

Овакав процес верификације полиномских идентитета није ограничен само на проверу производа полинома, већ се може применити на проверу било ког идентитета $P_1(x) = P_2(x)$, трансформисацијом $Q(x) = P_1(x) - P_2(x) \equiv 0$.

5 Фишер-Јејц алгоритам за насумичну пермутацију

Размотримо проблем генерисања насумичне пермутације првих N природних бројева.
Наивна метода решавања овог проблема је да сваки елемент разменимо са елемнтом низа који је насумично изабран. Различите пермутације добијене овим алгоритмом имаће различите вероватноће да буду генерисане. За свако $N > 2$, број различитих пермутација $N!$ не дели укупан број насумичних низова генерисаних алгоритмом N^N. Конкретно, по Бертрановом постулату, постоји барем један прост број између $N/2$ и N који дели $N!$ а не дели N^N. Из тог разлога, оваква метода пристрасно генерише пермутације.
Фишер и Јејц су предложили алгоритам за генерисање случајних пермутација секвенци коначне дужине који решава овај проблем [2]. Алгоритам производи непристрасне пермутације тј. свака пермутација је једнако вероватна. Савремена верзија алгоритма је ефикасна, време потребно за извршавање програма пропорционално је дужини секвенце и размењује елементе у месту.
Алгоритам је такође познат као Кнут-шафл, по Доналду Кнуту. Варијанта Фишер-Јејц алгоритма, позната као Сатолов алгоритам, користи се за генерисање цикличних пермутација дужине N, уместо насумичних пермутација.

5.1 Оригинална метода

Ова метода предложена је 1938. године у књизи *Статистичке методе за биолошка, агрокултурна и медицинска истраживања.* Њихова метода се изводила помоћу папира и оловке.
Кораци оригиналне методе за генерисање насумичне пермутације првих

N природних бројева:

1. Написати бројеве од 1 до N у листу $s1$.
2. Изабрати случајан број k између 1 и броја елемената у листи $s1$.
3. Бројањем са леве стране, у листу $s2$ уписати k-ти елемент из листе $s1$, затим избрисати k-ти елемент из $s1$.
4. Понављати од 2. корака док листа $s1$ не постане празна.
5. Листа $s2$ је резултантна пермутација.

5.1.1 Анализа временске сложености

У свакој од итерација потребно је пронаћи k-ти елемент, и њега избрисати из листе $s1$. То захтева $O(|s1|)$ временске сложености.
Како се извршава N итерација, укупна временска сложеност је $O(N^2)$.
Ова метода такође захтева додатних $O(N)$ меморије због листе $s2$.

5.2 Модерни алгоритам

Како наивна имплементација оригиналне методе ради у временској сложености $O(N^2)$, било је потребно прилагодити је модерним рачунарима у то време.
Модерну верзију коју данас користимо преложио је Ричард Дурстенфелд 1964. године а Доналд Кнут учинио популарном [3].
Модерна верзија разликује се од оригиналне методе јер размењује елементе у месту, уместо да их брише и ставља у другу листу, конкретно:

1. Написати бројеве од 1 до N у низ a, који је индексиран од 0.
2. Поставити бројач i на $N - 1$.
3. Изабрати случајан број j између 0 и i.
4. Разменити вредности a_i и a_j.
5. Смањити вредност i за 1.
6. Понављати од 3. корака уколико је бројач $i > 0$.

5.2.1 Анализа временске сложености

За извршавање сваког од наведених корака потребно је $O(1)$ времена. Алгоритам ће итерирати N пута па је укупна временска сложеност $O(N)$. Пошто се све извршава у месту, алгоритам не захтева додатну меморију.

5.3 Анализа тачности

Размотримо вероватноћу да број k буде извучен у неком кораку. Вероватноћа да број k буде први извучен једнака је $\frac{1}{N}$. Да би број k био извучен други, неопходно је да неки други број буде извучен први и да у следећем кораку буде извучен број k, па је вероватноћа да k буде извучен други једнака $\frac{N-1}{N}\frac{1}{N-1} = \frac{1}{N}$. Овај приступ рачунања вероватноће се може уопштити за свако $i \in [1, N]$ следећом формулом:

$$P_i = (\prod_{j=1}^{i-1} \frac{n-j}{n-j+1})\frac{1}{n-i+1}$$

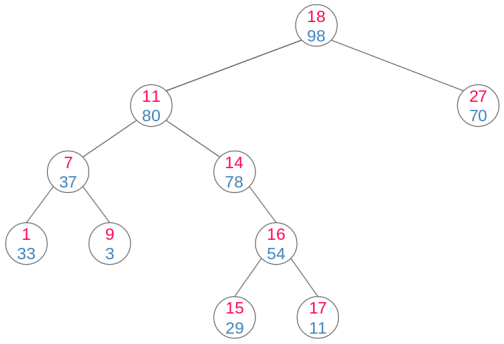
након унакрсног скраћивања чланова овог производа, добијамо:

$$P_i = \frac{1}{n}$$

Како су сви $P_i = \frac{1}{N}$, и не зависе од броја k, резултати алгоритма су непристрасни, тј. свака пермутација је једнако вероватна.

6 Трип

Трип (енг. *treap*) је рандомизирано самобалансирајуће бинарно стабло претраге. Име је добијено спајањем енглеских речи три (енг. *tree*) и хип (енг. *heap*). Структура у сваком чвору чува уређени пар (кључ, приоритет). *Inorder* обиласком стабла добија се низ сортиран по кључевима (бинарно стабло претраге). Такође, за сваки чвор важи да је приоритет већи од приоритета његове деце (хип). Због једноставније анализе претпоставимо да су кључеви и приоритети јединствени.



Слика 2: Визуелна репрезентација трипа где су кључеви представљени горњим бројем у чвору, а приоритети доњим.

Теорема 3. *Нека је* $S = \{(k_1, p_1), (k_2, p_2), ..., (k_n, p_n)\}$ *било који скуп такав да су кључеви* k *и приоритети* p *јединствени. Тада је трип* $T(S)$ *јединствено одређен.*

Доказ. Теорему доказујемо конструктивно. Теорема очигледно важи за $n < 2$. За $n \geq 2$ без умањења општости претпоставимо да p_1 има највећу вредност приоритета. Тада због особине хипа елемент (k_1, p_1) мора бити корен трипа $T(S)$. Сви елементи скупа S са кључем мањим од k_1

се налазе у левом подстаблу чвора (k_1, p_1). Аналогно, важи да се елементи са кључем већим од k_1 налазе у десном подстабу чвора (k_1, p_1). На тај начин се рекурзивно генеришу леви и десни подтрип који су такође јединствено одређени. □

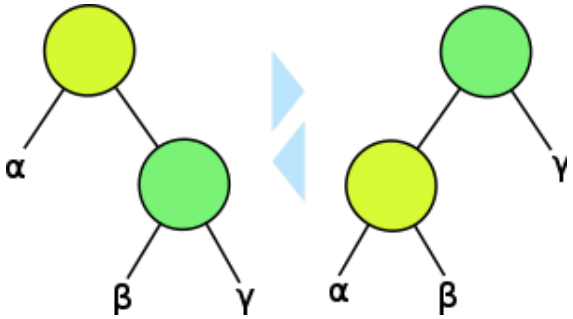
6.1 Основне операције

6.1.1 Претрага по кључу

Претрага по кључу се изводи као и претрага у обичном бинарном стаблу претраге. У овом случају се приоритети игноришу.

6.1.2 Додавање елемента

Операција додавања кључа x се састоји из три корака. Први корак је бирање вредности приоритета y, униформно насумично, који ће се упарити са k у неки чвор i. Други корак је додавање чвора i по вредности кључа као код бинарног стабла претраге. После првог корака елемент i ће се налазити на позицији листа и трип ће испуњавати услове бинарног стабла претраге. Трећи корак служи да се испуне услови хипа. Све док i није корен трипа и има већи приоритет од свог родитеља j, извршава се ротација између i и j.
Ротација је операција замене места неког чвора и његовог родитеља. Конкретно, у бинарном стаблу претраге, ако је неки чвор a лево дете свог родитеља b, после ротације ће b бити десно дете чвора a. У супротном смеру, ако је a десно дете свог родитеља b, после ротације ће b бити лево дете чвора a. Такође, њихова подстабла се мењају на начин као на слици 3.



Слика 3: Визуелна репрезентација ротације два чвора у трипу.

Ротацијом се одржава поредак по кључевима, али мења поредак приоритета ротираних чворова. Приметити да се ротација извршава у константном времену јер се манипулише са неколико показивача.

6.1.3 Брисање елемента

Ако је чвор који треба обрисати лист трипа, уклонити га.
Ако чвор x који треба обрисати има тачно једно дете z , ставити чвор z на место где је био чвор x (поставити z на место детета x-овог родитеља или га поставити на место корена уколико x нема родитеља) и уклонити чвор x.
Ако чвор x који треба обрисати има два детета, заменити x са дететом које има већи приоритет.

Јасно је да је временска сложеност свих наведених операција пропорцијална дубини стабла.

6.2 Напредне операције

6.2.1 Раздвајање

Дата је вредност кључа x, потребно је раздвојити трип на два трипа, један са вредностима кључева строго већим, и један са вредностима кључева строго мањих од x.
У трип се убацује елемент (x, ∞). Јасно је да ће тај чвор сигурно бити корен трипа због вредности приоритета. Због особине бинарног стабла претраге, његово лево подстабло имаће кључеве мање од x, а његово десно подстабло кључеве веће од x. Та два подстабла су тражени трипови.

6.2.2 Спајање

Дата су два трипа T_1 и T_2 такви да је највећа вредност кључа из T_1 мања од најмање вредности кључа из T_2, потребно је спојити их у један трип. Прави се чвор i који има вредност $(x, -\infty)$, $x > max(p \in T_2) \wedge x < min(p \in T_2)$, и поставља се на место корена у новом трипу. За његово лево подстабло поставља се T_1, а T_2 за десно. За овакав трип важе особине бинарног стабла претраге али нису испуњене особине хипа. Због тога

је потребно понављати операције ротације између i и његовог детета са већим приоритетом, све док i не постане лист, када се уклања.

Као и основне операције, напредне операције такође имају сложеност пропорцијалну дубини стабла. Комбинацијом наведених операција могу се ефикасно имплементирати и операције уније, пресека и разлике.

6.3 Анализа сложености

Теорема 4. *Очекивана вредност дубине чвора у трипу пропорционална је логаритму броја елемената трипа.*

Доказ. Замислимо да су чворови сортирани по величини кључа. Нека је i чвор са i-тим најмањим приоритетом. Дубина неког чвора у стаблу једнака је броју чворова који су његови директни преци. Означимо са d_i дубину чвора i и са I_i^j индикатор догађаја да је чвор j директан предак чвора i.

$$d_i = \sum_{j=1, j\neq i}^{n} I_i^j$$

$$E[d_i] = E[\sum_{j=1, j\neq i}^{n} I_i^j]$$

због линеарности очекиване вредности имамо:

$$E[d_i] = \sum_{j=1, j\neq i}^{n} E[I_i^j]$$

Индикатор догађаја има очекивану вредност једнаку вероватноћи да се тај догађај оствари. Из тога следи:

$$E[d_i] = \sum_{j=1, j\neq i}^{n} p_{ij}$$

Анализирајмо p_{ij}. Ако неки елемент са кључем k има већи приоритет и од i и од j, и k је строго мањи или строго већи од кључева i и j, такав чвор не утиче на однос *директан предак-наследник* између i и j. Следећа три случаја утичу на тај однос [7]:

1. Елемент i има највећи приоритет
   У овом случају j не може бити предак i јер се би важила особина хипа.
2. Један од елемената чија је вредност кључа између вредности кључева i и j има највећи приоритет (ни i ни j немају највећи приоритет). Нека је тај елемент k.
   У овом случају j такође не може бити предак чвора i. Претпоставимо супротно, да је j директни предак чвора i. У том случају би j морао да буде директни предак чвора k, што је немогуће из истих разлога као у првом случају, јер би директан предак чвора k имао мањи приоритет од свог наследника, што не испуњава хип особине.
3. Елемент j има највећи приоритет
   У овом случају ће j бити директан предак чвора i, јер би у супротном постојао чвор са већим приоритетом, што није услов случаја.

Пошто су вредности приоритета одабране униформно случајно, вероватноћа да у поднизу $[i, j]$ (или $[j, i]$ ако је $j < i$) чвор j има највећи приоритет једнака је $\frac{1}{|j-i|+1}$. Када ову формулу за вероватноћу убацимо у формулу за очекивану вредност дубине добијамо:

$$E[d_i] = \sum_{j=1, j\neq i}^{n} \frac{1}{|j-i|+1}$$

$$= \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1}$$

$$= H_i - 1 + H_{n-i+1} - 1 < 2ln(n)$$

□

Како је очекивана вредност дубине стабла $E[d] < 2ln(n)$, закључујемо да је сложеност свих наведених операција $O(logn)$.

7 Милер-Рабинов тест за просте бројеве

Милер-Рабинов тест за просте бројеве је један од најраспрострањених алгоритама за испитивање сложености бројева, како због брзине, тако и због једноставности имплементације. Открио га је Гери Л. Милер 1976. године. Његова верзија алгоритма је детерминистичка, али је тачност алгоритма упитна јер се ослања на проширену Риманову хипотезу, која није доказана. Мајкл О. Рабин је алгоритам прилагодио у пробабилистички алгоритам 1980. године.

7.1 Математички концепти

Милер-Рабинов тест заснива се на провери да ли неки број испуњава одређена својства која важе за просте бројеве.

7.1.1 Јаки вероватно прости бројеви

Нека је дат неки непаран број $n > 2$. Тада број $n - 1$ можемо записати као $2^S d$, где је $S > 0$ и d непаран број. Нека је број a узајамно прост са n. Тај број ћемо звати *база*. Број n називамо *јаким вероватним простим бројем за базу* a ако је тачна нека од следећих конгруентних релација:

- $a^d \equiv 1 \ (mod \ n)$
- $a^{2^r d} \equiv -1 \ (mod \ n)$ за неко $0 \leq r < S$

Из мале Фермаове теореме имамо $a^{n-1} - 1 \equiv 0 \ (mod \ n)$, односно:

$$(a^d - 1)(a^d + 1)(a^{2d} + 1)(a^{4d} + 1) \cdots (a^{2^{S-1}d} + 1) \equiv 0 \ (mod \ n)$$

Идеја је да ако је n непаран прост број, проћи ће овај тест јер су 1 и -1 једини квадратни корени јединице по модулу n.

По принципу контрапозиције, ако n није јак вероватно прост број за базу a, онда са сигурношћу знамо да је n сложен број. Тада се a зове *сведок*. Ако је n сложен број, могуће је да прође овај тест, тј. да буде јак вероватно прост број за базу a. Такав број зовемо *јак псеудопрост број*, а број a зовемо *јак лажни сведок*.

7.1.2 Избор базе

Ни један сложен број не може бити јак псеудопрост број за све базе. Међутим, не постоји једноставан начин да се сведок пронађе. Наивно решење било би да се n провери за сваку базу, али овај приступ резултује спорим детерминистичким алгоритмом. Решење понуђено у Милер-Рабиновом алгоритму јесте да се база бира униформно случајно, што значајно убрзава поступак провере на уштрп тачности. Тачност можемо побољшати независним понављањем алгоритма.

7.2 Алгоритам

```
1  bool MillerRabin(ll n) {
2      ll a = r()%(n-2)+2;
3      ll s = 0, d = n-1;
4      while(~(d>>s)&1)
5          s++;
6      d >>= s;
7      ll x = stepen(a,d,n), y;
8      while(s--){
9          y = (x*x)%n;
10         if ( y == 1 && x != 1 && x != n-1 )
11             return false;
12         x = y;
13     }
14     if (y!=1)
15         return false;
16     return true;
17 }
18 bool prost(ll n, int k) {
19     while(k--)
20         if ( !MillerRabin(n) )
21             return false;
22     return true;
23 }
```

Слика 4: $C++$ код за Милер-Рабинов алгоритам који се независно понавља k пута

7.3 Анализа тачности

Грешка при извршавању Милер-Рабиновог теста се дешава када јак псеудопрост број прође тест. Може се показати да ако је n сложен број, постоји највише $\frac{n}{4}$ јаких лажних сведока. Због тога је вероватноћа грешке једнака:

$$p = \frac{\frac{n}{4}}{n} = 4^{-1}$$

Ако тест независно понављамо k пута, вероватноћа греше једнака је 4^{-k}.

Означимо са P догађај да је број који тестирамо прост, и са MR_k догађај да број који тестирамо пролази Милер-Рабинов тест независно поновљен k пута. Тада можемо израчуати $p(\neg P|MR_k)$ коришћењем Бајесове формуле.

$$\begin{aligned} p(\neg P|MR_k) &= \frac{p(\neg P \wedge MR_k)}{p(\neg P \wedge MR_k) + p(P \wedge MR_k)} \\ &= \frac{1}{1 + \frac{p(MR_k|P)}{p(MR_k|\neg P)} \frac{p(P)}{p(\neg P)}} \\ &= \frac{1}{1 + \frac{1}{p(MR_k|\neg P)} \frac{p(P)}{p(\neg P)}} \end{aligned}$$

У последњој једначини се користи чињеница да ће прост број сигурно проћи тест, па је $p(MR_k|P) = 1$.

$$p(\neg P|MR_k) < p(MR_k|\neg P) \left(\frac{1}{1 + p(P)} - 1 \right) < 4^{-k} \left(\frac{1}{1 + p(P)} - 1 \right)$$

Ова формула је корисна када су нам познате информације о расподели простих бројева на улазу. [6]

8 Чести типови задатака

У овом поглављу описани су чести типови задатака који се поред претходно наведених алгоритама често појављују на такмичењима. За све задатке подразумевају су стандардна временска и меморијска ограничења.

8.1 Задатак 1

(СИО 2021/22 Први задатак) - Интерактивни задатак -
У скупу од n људи, свако од њих или увек лаже, или увек говори истину. Потребно је открити једног од лажљиваца. Могуће је поставити q питања. Свако од питања поставља се некој особи и та особа одговара на питање да ли у скупу $S = \{s_1, s_2, ..., s_k\}$ постоји непарно лажљиваца. Скуп S може да буде празан, а може и да садржи особу којој се поставља питање.

Ограничења:
$n = 1000, q = 30$

8.1.1 Анализа решења

Приметимо да ако питамо неку особу, нека је то особа са индексом 1 ради једноставности, да ли у празном скупу има непаран број лажљиваца, можемо да закључимо да ли је та особа лажливац или не. Уколико је она лажљивац, враћамо индекс 1. У супротном знамо да се лажљивац налази на интервалу $[2, n]$. Користићемо одговоре особе 1 како бисмо пронашли лажљивца.
Претпоставимо да знамо који скуп има непарно лажљиваца. Техником половљења тог интервала, бинарном претрагом, можемо пронаћи скуп кардиналности 1 који има непаран број лажљиваца, односно наћи конкретног лажљивца. Та техника захтева највише 10 корака, јер је $\lceil log_2 n \rceil \leq 10$.
Остаје нам 19 корака за налажење тог подскупа који има непарно лажљи-

ваца. То можемо урадили пробабилистички. Сваки елемент убацујемо у наш подскуп са вероватноћом $\frac{1}{2}$. Самим тим је вероватноћа да тај подскуп има непарно много лажљиваца $\frac{1}{2}$. Уколико подскуп који конструишемо има паран број лажљиваца, генеришемо нови подскуп. Вероватноћа да ће за то требати преко 19 покушаја је мања од $\frac{1}{2^{19}}$, што је довољно прецизно.

Временска сложеност је $O(qn)$, а меморијска $O(n)$.

8.2 Задатак 2

- Интерактивни задатак -
Дат је природан број N. Потребно је погодити стринг s дужине N који се састоји само од карактера A, C, T и G, помоћу интеракције са грејдером. Грејдеру се поставља питање функцијом $prefix(t)$, која враћа $true$ уколико је t префикс стринга s, а $false$ у супротном. Стринг s је на почету фиксиран, односно грејдер није адаптиван.

Ограничења:
$N = 100000$
Дозвољени број питања $Q = 250000$

8.2.1 Анализа решења

Анализирајмо очекивани број питања потребних да бисмо погодили прво слово. Фиксирајмо редослед карактера A, C, T и G насумично, нпр. користећи Фишер-Јејц алгоритам. Нека је добијени редослед g_1, g_2, g_3 и g_4. Сада постављамо питање $prefix(g_1)$. Уколико је $s_1 = g_1$, знамо да је g_1 прво слово стринга s. У супротном понављамо поступак са g_2, па g_3. Ако је $prefix(g_1) = prefix(g_2) = prefix(g_3) = false$, нема потребе да постављамо питање за g_4, јер је то једина преостала могућност, па сигурно знамо да је $s_1 = g_4$. Очекивана вредност броја питања је:

$$E(broj\_pitanja) = p(s_1 = g_1) \cdot 1 + p(s_1 = g_2) \cdot 2 + p(s_1 = g_3) \cdot 3 + p(s_1 = g_4) \cdot 3$$

Како је редослед g_1, g_2, g_3 и g_4 униформно случајна пермутација, имамо:

$$E(broj\_pitanja) = \frac{1}{4}1 + \frac{1}{4}2 + \frac{1}{4}3 + \frac{1}{4}3 = 2.25$$

Када смо погодили префикс дужине k, p_k, претходно наведени поступак

понављамо за свако следеће слово, редом, тако што функцију $prefix$ позивамо са параметром $p_k + g_i$, где је + операција конкатенације стрингова.

Очекивани број постављених питања једнак је $N \cdot 2.25 = 225000 < 250000$.

Временска и меморијска сложеност је $O(N)$.

8.3 Задатак 3

Дат је број N и низ a од N целих бројева. Потребно је одговорити на питање колико има узастопних поднизова тако да се свака вредност у поднизу појављује паран број пута?

Ограничења:
$N \leq 10^5$
$1 \leq a_i \leq N$

8.3.1 Анализа решења

Назовимо ексклузивну дисјункцију свих елемената неког низа *xor* вредношћу. Тада сваки подниз који задовољава услов задатка има *xor* вредност једнаку нули, јер је $a \; xor \; a = 0$, а свака вредност се појављује паран број пута. То значи да ако посматрамо узастопни подниз $a_l, a_{l+1}, ..., a_r$ чија је *xor* вредност једнака нули, леви префиксни *xor* низ, низа a, означимо га са A, задовољава $A_r = A_{l-1}$.
Ако само избројимо колико интервала (l, r) задовољава $A_r = A_{l-1}$, може се десити лажно позитиван интервал, и самим тим нетачан резултат, На примеру низа $1, 2, 5, 6$ можемо да видимо да је *xor* вредност тог низа једнака нули, а ни један елемент се не појављује у пару.
Како бисмо то избегли, направимо низ b, $b_i = X1_{a_i}$, где је $X1$ нека случајно генерисана пермутација првих N бројева. Пошто се исте вредности низа a сликају у исту вредност у низу b, услов за једнак префиксни *xor* за низ b такође мора бити задовољен. Нека је префиксни *xor* низ, низа b, означен са B. Сада на сличан начин бројимо интервале (l, r), тако да важи $(A_r, B_r) = (A_{l-1}, B_{l-1})$. Тиме смо смањили вероватноћу да се лажно позитиван интервал (l, r) догоди и у A, и у B.
Како бисмо довољно смањили вероватноћу грешке, уместо генерисања једног низа, потребно је генерисати 4, где се за сваки низ користи друга, насумично генерисана пермутација Xi. Задатак на сличан начин ре-

шавамо бројећи интервале (l, r), тако да важи:
$(A_r, B_r, C_r, D_r, E_r) = (A_{l-1}, B_{l-1}, C_{l-1}, D_{l-1}, E_{l-1})$.
Ово најбрже решавамо користећи мапу, која за сваку уређену петорку $(A_i, B_i, C_i, D_i, E_i)$ чува колико пута се појавила. Детаљније, итерирамо по десном крају интервала r, додамо на резултат вредност из мапе за кључ $(A_r, B_r, C_r, D_r, E_r)$. Након сваке итерације у мапу додајемо 1 на вредност за кључ $(A_r, B_r, C_r, D_r, E_r)$. Пре свих итерација у мапу је потребно убацити вредност 1, за кључ $(0, 0, 0, 0, 0)$.

Временска сложеност је $O(NlogN)$, а меморијска $O(N)$.

8.4 Задатак 4

Дате су целобројне (x, y) координате N тачака у равни. Потребно је одговорити на питање колико се максимално тачака налази на некој правој. Гарантује се да је решење веће од $\frac{n}{4}$.

Ограничења:
$N \leq 10^5$

8.4.1 Анализа решења

Права је одређена са две тачке. Ако униформно случајно изаберемо две тачке, вероватноћа да су обе на резултантној правој једнака је $\frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16}$, односно, вероватноћа да оне не образују резултантну праву једнака је $\frac{15}{16}$. За конкретну праву p, потребно је проћи кроз све тачке и избројати колико њих се налази на њој. Независним понављањем овог поступка k пута, вероватноћа да ни један није дао тачнан резултат једнака је $(\frac{15}{16})^k$. Ограничења дозвољавају да k буде око 500, што задовољава тачност.

Временска сложеност је $O(kN)$, а меморијска $O(N)$.

9 Закључак

У овом раду представили смо неке од најзначајних алгоритама, структура података и типова проблема који у својој логици користе случајне величине и вероватноћу. Они су корисни како у такмичарском програмирању тако и у практичним индустријским применама. Неке од најчешћих су коришћене за предвиђање исхода неких процеса, статистичке анализе, разне физичке симулације, процене тржишта, машинско учење, а налазе примену и у медицинској хемији за генерисање једињења, у навигацији, теорији графова, криптографији и многим другим областима.

Литература

[1] Rajeev Motwani, Prabhakar Raghavan. *Randomized algorihms*. 1995.

[2] Ronald Aylmer Fisher, Frank Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. 1938.

[3] Donald E. Knuth. *The art of computer programming. Volume 2, Seminumerical algorithms*. 1969.

[4] Ran Duan, Hongxun Wu, Renfei Zhou. *Faster Matrix Multiplication via Asymmetric Hashing*. 2022. `https://arxiv.org/abs/2210.10173`

[5] CS Lecture Notes, Carnegie Mellon University. `https://www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture_notes/lect0123.pdf`

[6] CS Lecture Notes, Cornell University. `http://www.cs.cornell.edu/courses/cs482/2008sp/handouts/mrpt.pdf`

[7] Margaret Reid-Miller. *Parallel and Sequential Data Structures and Algorithms — Lecture 16 15-210 (Spring 2012)*. `http://www.cs.cmu.edu/afs/cs/academic/class/15210-s12/www/lectures/lecture16.pdf`

[8] Michael Mitzenmacher, Eli Upfal. *Probability and Computing, Randomized Algorithms and Probabilistic Analysis*. 2005. `http://lib.ysu.am/open_books/413311.pdf`