Математички факултет
Универзитет у Београду

Алгоритми за генерисање пермутација и комбинација

Мастер рад

Студент:
Јана Марковић 1014/2009

Комисија:
проф. др Миодраг Живковић, ментор
проф. др Предраг Јаничић
доц. др Младен Николић

Садржај

[1] овде се Х односи на енглеско слово X

Поглавље 1

Увод

Тема рада је представљање неелементарних алгоритама за генерисање свих пермутација и комбинација, описаних у поглављима 7.2.1.2 и 7.2.1.3 књиге *Уметност рачунарског програмирања*.

Доналд Ервин Кнут је амерички научник на пољу рачунарства, математичар и почасни професор универзитета Стенфорд у пензији. Фундаментално је допринео развоју више грана теоријског рачунарства. Допринео је формалном проучавању комплексности алгоритама и систематизовао математичке технике за то, током чега је популарисао и асимптотску нотацију сложености алгоритама. Творац је језика за обраду и прелом текста на рачунару TeX, као и језика за опис фонтова METAFONT. Направио је програмске системе WEB и CWEB, намењене за подршку „књижевном програмирању" - идеји да се програмира као што се пише књижевно дело - угнежђујући код унутар текста који га описује (обрнуто од традиционалног система кодирања са коментарима), са акцентом на томе да је погодно за читање човеку, а не диктирано од стране преводиоца (компајлера)[3].
Ипак, оно по чему је Кнут најпознатији је *Уметност рачунарског програмирања*, свеобухватна монографија о алгоритмима и њиховој анализи. 1970-их година рачунарство је било поље без идентитета, објављивано је доста чланака али њихов стандард није био висок, а доста радова је садржало грешке. Кнут је започео дело као основу за своју књигу о програмирању компајлера, и убрзо видео да је пре тога потребно написати књигу о основама теорије рачунарског програмирања. Замишљено је као дело од седам томова, од којих су тренутно објављена непуна четири. Прва три тома објављена су 1968., 1969. и 1973. године. Покривају редом: основне алгоритме, семинумеричке алгоритме, и алгоритме за претрагу и сортирање. Четврти том је због своје обимности издељен у више делова - „фасцикли". Фасцикле 2 и 3 о комбинаторним алгоритмима су објављене 2005. године, а последња издата је фасцикла о задовољивости, у децембру 2015. године [3].

Пермутација скупа S је избор елемената скупа S при чему је редослед елемената битан.
Комбинација скупа S је избор елемената скупа S при чему редослед елемената није битан.

Пермутације и комбинације су у корену многих битних грана математике - теорије бројева, алгебре, геометрије, вероватноће, статистике, дискретне математике, теорије графова, и других. Пермутације и комбинације имају широку примену. Користе се у рачунарским мрежама, криптографији и сигурности на мрежи. Рутирање различитих комбинација путева на мрежи ради процене перформанси је чест проблем у овим областима. Рачунарске мреже захтевају сигуран пренос информација, што води до криптографије и сигурности на мрежи, области

које су засноване на комбинаторици, а посебан значај у криптографији имају пермутације. У архитектури рачунара пројектовање чипова укључује разматрање могућих пресликавања улазних на излазне пинове.
Молекуларна биологија укључује многе проблеме везане за ређање и анализу шаблона атома, молекула, ДНК, протеина и гена, који се могу посматрати као комбинаторни и проблеми пермутација. Језици - природни и вештачки, су блиско повезани са комбинаториком. На пример, алгоритми за претрагу текста често су засновани на комбинаторици речи и карактера. Директне примене ових алгоритама су у базама података и процесирању текста. Анализа ефикасности ових алгоритама је још једна велика област примене комбинаторике.
У научним истраживањима, генерисање свих комбинација кандидата за решење је важан проблем. Упити у базама података који се састоје од више *join* операција се посматрају као пермутације у којима су елементи чиниоци *join*-а. Одређивање оптималне пермутације која даје упит који ће се најбрже извршити је чест и важан проблем у базама података. Многи проблеми оптимизације у области операционих истраживања (енг. *Operations research*), као на пример проблем распоређивања послова, засновани су на комбинаторним проблемима. Симулације, национална сигурност су још неке од области где се комбинације и пермутације значајно примењују [4].

Поглавља 7.2.1.2 и 7.2.1.3 садрже, по Кнутовом избору, једанаест најважнијих алгоритама за генерисање пермутација, и пет алгоритама за генерисање комбинација. Сви алгоритми су описани неформално и са „go to" скоковима. Поред представљања алгоритама, циљ овог рада је и прецизно формулисање свих алгоритама на псеудојезику, без „go to" скокова, а затим и развијање пратеће имплементације.
У оригиналној верзији алгоритми су описани низом нумерисаних *корака*, где корак $i+1$ следи након корака $i, i \geq 1$. У зависности од испуњености услова везаних за алгоритам, ток алгоритма се често преусмерава на неки од наредних или неки од претходних корака, за шта се користи „go to" команда. Овај начин приказа алгоритама је краћи и алгоритми су читљивији, али је имплементација тежа. Након трансформације извршене у овом раду, имплементација алгоритама је значајно поједностављена.
Основна идеја иза трансформације је да се за елиминацију „go to" користе команде контроле тока. Типично за све алгоритме је постојање „спољашњег" „go to" - преусмеравања корака са последњег корака на корак исписа пермутације који се налази на почетку алгоритма. Овај „go to" је елиминисан тако што су кораци између почетног и последњег смештени у *while(true)* петљу, из које се излази када је испуњен услов за завршетак алгоритма. Осим ове најједноставније трансформације, велики број алгоритама у оригиналној верзији садржи гранања у зависности од услова везаних за алгоритам, а потом „go to" на неки од претходних или наредних корака. Овакве конструкције решене су употребом *while*(*true*) петље, а потом командама *break* и *continue* које служе да усмере извршавање на потребан корак. У неким алгоритмима који имају више гранања (на пример, алгоритам X из 2.1.9) овај принцип је примењен на више нивоа, тј. постоји више угнеждених петљи. Овакав приступ елиминисању „go to" подразумева и да се неки делови кода, тј. кораци, понављају, чега у изворној верзији нема. Унутар корака нису прављене измене.
На пример, најједноставнији алгоритам Л за лексикографско генерисање пермутација (детаљније у 2.1.2) садржи четири корака у изворној варијанти и гласи:

Алгоритам 1.0.1. *Л (генерисање пермутација лексикографским редоследом оригинална верзија)*
L1. [Испис пермутације] Испис пермутације $a_1 ... a_n$
L2. [Одређивање j] Постави $j \leftarrow n-1$. *Ако је* $a_j \geq a_{j+1}$ *смањуј j за 1 све док не буде* $a_j < a_{j+1}$.
Уколико је $j = 0$ *заврши алгоритам.*
L3. [Повећавање a_j*]Постави* $l \leftarrow n$. *Ако је* $a_j \geq a_l$ *смањуј l за 1 све док не буде* $a_j < a_l$.
Онда замени $a_j \leftrightarrow a_l$.
L4. [Обртање подниза $a_{j+1} ... a_n$*] Постави* $k \leftarrow j+1, l \leftarrow n$.
Онда, ако је $k < l$, *замени места* $a_k \leftrightarrow a_l$, *постави* $k \leftarrow k+1, l \leftarrow l-1$,
и понављај док не буде $k \geq l$. *Врати се на L1.*

док прерађен изгледа овако:

Алгоритам 1.0.2. *Л (генерисање пермутација лексикографским редоследом верзија без GO TO скокова)*
begin

(1) **while true do**
(2) *Испис [испис пермутације* $a_1 ... a_n$*]*
(3) $j \leftarrow n-1$
(4) **while** $j \geq 0$ **and** $a_j \geq a_{j+1}$ **do**
(5) $j \leftarrow j-1$
(6) **if** $j = 0$ **then**
(7) **exit** *[нема наредне пермутације]*
(8) $l \leftarrow n$
(9) **while** $a_j \geq a_l$ **do**
(10) $l \leftarrow l-1$
(11) *Заменити* a_j *и* a_l
(12) $k \leftarrow j+1$, $l \leftarrow n$
(13) **while** $k < l$ **do**
(14) *Заменити* a_k *и* a_l
(15) $k \leftarrow k+1$, $l \leftarrow l-1$
end

Прерађени алгоритми су представљени псеудокодовима, а потом је направљена имплементација псеудокодова у оквиру конзолне апликације. Апликација је тестирана користећи примере наведене у поглављима 7.2.1.2 и 7.2.1.3. На самом крају, алгоритми су експериментално упоређени - до које границе се могу извршавати, и које време им је за то потребно.

Поглавље 2.1 се бави алгоритмима за генерисање свих пермутација. Представљено је једанаест алгоритама за генерисање свих пермутација, само одређеног подскупа пермутација који задовољава одређене услове, а неки алгоритми показују и како се алгоритми за генерисање пермутација могу применити. Поглавље 2.2 обрађује пет алгоритама који се баве генерисањем комбинација и проблеме на које се могу применити. У поглављу 3 наведени су детаљи имплементације прерађених псеудокодова и алгоритми су експериментално упоређени.

Посебну захвалност дугујем ментору проф. др. Миодрагу Живковићу на великој помоћи и стрпљењу током израде овог рада, као и члановима комисије проф. др. Предрагу Јаничићу и доц. др. Младену Николићу за вредне сугестије којима су помогли да се рад употпуни и заокружи као целина.

Поглавље 2

Алгоритми за генерисање пермутација и комбинација

2.1 Генерисање свих пермутација

2.1.1 Увод

Једна од најважнијих тема у комбинаторним алгоритмима за генерисање је проблем обилажења, тј. генерисања свих пермутација. Ово поглавље се бави најбитнијим алгоритмима за генерисање свих пермутација, и показује како се неки од њих могу применити.
Поглавље почиње основним алгоритмом за генерисање пермутација у лексикографском поретку, алгоритмом Л. Наредна тачка 2.1.3 представља *метод простих замена*, алгоритам у коме се свака пермутација добија од претходне заменом места два суседна елемента. Индекси елемената који мењају места за дато n чине константан низ, па се тај низ може израчунати унапред, а алгоритам за то је тема тачке 2.1.4. Следи поглавље 2.1.5 које приказује примену пермутација за решавање *алфаметика* проблема. Након тога следи неколико тачака које обрађују алгоритме који су засновани на мултипликативним својствима пермутација. Основна идеја иза ових алгоритама је да се пермутација посматра као пресликавање, а пермутације се генеришу множењем са специјалним пресликавањима. Поглавље 2.1.9 представља алгоритам Х, намењен у применама у којима је потребан само подскуп скупа свих пермутација, који чине пермутације које задовољавају задати скуп услова. Затим је размотрен концептуално вероватно најједноставнији алгоритам за генерисање комбинација - алгоритам Ц, у тачки 2.1.11, који пермутације генерише цикличним померајима. Након њега следи алгоритам Е који пермутације генерише користећи посебну врсту замене - звезда замене, где се свака наредна пермутација добија заменом места елемента a_0 и неког другог елемента. На крају је размотрен проблем тополошког сортирања - када нису потребне све пермутације неког скупа, већ само одређен подскуп пермутација које задовољавају одређене услове, и алгоритам који спроводи тополошко сортирање, назван алгоритам В.
Сви приказани алгоритми, осим алгоритма Л који се разматра први, и алгоритма В за проналажење свих тополошких уређења, служе за генерисање пермутација без понављања.

2.1.2 Алгоритам Л - генерисање пермутација у лексикографском поретку

Ово је класичан и једноставан алгоритам који генерише пермутације са и без понављања у лексикографском поретку. Потиче из Индије, из 14. века, а изумео га је Нарајана Пандита. За задати низ n елемената $a_1a_2...a_n$, сортиран тако да је $a_1 \leq a_2 \leq ... \leq a_n$, алгоритам генерише све пермутације скупа $\{a_1, a_2, ..., a_n\}$, посећујући их у лексикографском поретку.
Алгоритам је заснован на генералном правилу за генерисање лексикографски следећег елемента сваке комбинаторне схеме:

1. Наћи највећи индекс j такав да a_j може да се увећа
2. Увећати j за најмању могућу вредност
3. Наћи начин да се $a_1...a_j$ прошири до комплетне схеме, прве наредне у лексикографском поретку

У наставку је псеудокод алгоритма.

Алгоритам 2.1.1. *Л (генерисање пермутација лексикографским редоследом)*
Улаз: *сортиран низ a дужине n, тако да је $a_1 \leq a_2 \leq \cdots \leq a_n$.*
Излаз: *све пермутације скупа $\{a_1, a_2, \ldots, a_n\}$, у лексикографском поретку.*
Пример: *за скуп $\{1, 2, 2, 3\}$ добија се:*

$$1223, 1232, 1322, 2123, 2132, 2213, 2231, 2312, 2321, 3122, 3212, 3221.$$

Напомена: *због једноставности се претпоставља да је низу a додат помоћни елемент a_0, такав да је $a_0 < a_n$*
begin

(1) **while true do**
[Инваријанта петље: $\{a_1, a_2, \ldots, a_n\}$ је лексикографски највећа од до сада исписаних пермутација и лексикографски мања од свих пермутација које још нису исписане]
(2) *Испис [испис пермутације $a_1 a_2 \ldots a_n$ и одређивање наредне у наставку]*
(3) $j \leftarrow n - 1$ *[одређивање индекса j]*
(4) **while** $j \geq 0$ **and** $a_j \geq a_{j+1}$ **do**
(5) $j \leftarrow j - 1$
(6) **if** $j = 0$ **then**
(7) **exit** *[нема наредне пермутације]*
[на овом месту је j најмањи индекс такав да су исписане све пермутације које почињу са $a_1 \ldots a_j$]
[повећавање a_j; одређивање највећег индекса l таквог да је $a_l > a_j$]
(8) $l \leftarrow n$
(9) **while** $a_j \geq a_l$ **do**
(10) $l \leftarrow l - 1$
(11) *Заменити a_j и a_l*
[пошто је $a_{j+1} \geq \cdots \geq a_n$, a_l је најмањи елемент већи од a_j који може да следи иза $a_1 \ldots a_{j-1}$ у пермутацији; пре замене било је $a_{j+1} \geq \cdots \geq a_{l-1} \geq a_l > a_j \geq a_{l+1} \geq \cdots \geq a_n$; после замене је $a_{j+1} \geq \cdots \geq a_{l-1} \geq a_j > a_l \geq a_{l+1} \geq \cdots \geq a_n$]
[инверзија редоследа $a_{j+1} \ldots a_n$]
(12) $k \leftarrow j + 1$, $l \leftarrow n$
(13) **while** $k < l$ **do**
(14) *Заменити a_k и a_l*
(15) $k \leftarrow k + 1$, $l \leftarrow l - 1$
end

Када су елементи улазног низа различити алгоритам се може унапредити препознавањем специјалног случаја у кораку тражења индекса j (линије $3 - 5$ у псеудокоду), када је $j = n - 2$, који важи за половину од $n!$ пермутација, то су пермутације код којих важи $a_{n-1} < a_n$. Алгоритам је ефикаснији када су елементи улазног низа a различити, јер је вероватноћа да је $j \leq n - t$ само $1/t!$, па петље у линијама 4 и 13 имају мали број итерација. Алгоритам је ефикасан и када се одређени елементи понављају, осим у случају када се неке вредности улазног низа појављују много чешће него друге.

2.1.3 Алгоритам П - генерисање пермутација методом простих замена

Алгоритам простих замена (*енг. plain changes*) генерише свих $n!$ пермутација низа различитих елемената дужине n правећи $n! - 1$ замену суседних елемената. Добио је назив просте замене

зато што су то замене суседних елемената. Потиче из Енглеске, из 17. века. Идеја алгоритма је узети низ пермутација дужине $n-1$ у коме је свака добијена од претходне заменом места два суседна елемента, и убацити n на сваку могућу позицију.
На пример, за низ пермутација $\{123, 132, 312, 321, 231, 213\}$, додавањем $n = 4$ на све четири позиције, добија се:

$$\begin{array}{cccccc} \mathbf{1234} & 1\mathbf{3}24 & 31\mathbf{2}4 & 321\mathbf{4} & 23\mathbf{1}4 & 213\mathbf{4} \\ 12\mathbf{43} & 13\mathbf{4}2 & 31\mathbf{4}2 & 32\mathbf{4}1 & 23\mathbf{4}1 & 21\mathbf{4}3 \\ 1\mathbf{423} & 1\mathbf{432} & 3\mathbf{412} & 3\mathbf{421} & 2\mathbf{431} & 2\mathbf{413} \\ \mathbf{4}123 & \mathbf{4}132 & \mathbf{4}312 & \mathbf{4}321 & \mathbf{4}231 & \mathbf{4}213 \end{array} \tag{2.1}$$

Добијени низ се чита на следећи начин: прва колона одозго-надоле, друга колона одоздо-нагоре, трећа колона одозго-надоле, итд.
Свака пермутација се од претходне добија једном заменом два суседна елемента, па се овакав редослед назива још и „грејовски", јер подсећа на Грејов бинарни код, у коме се сваке две узастопне вредности разликују на само једном биту.
Алгоритам се заснива на коришћењу узајамне повезаности пермутације и њене табеле инверзије. Инверзије и табеле инверзија се дефинишу на следећи начин:

Нека је $a_1a_2...a_n$ пермутација скупа $\{1, 2, ..., n\}$. Уколико постоје i и j, такви да је $i < j$ и $a_i > a_j$, пар (a_i, a_j) се назива **инверзија** пермутације. На пример, пермутација 3 1 4 2 има 3 инверзије: $(3, 1)$, $(3, 2)$ и $(4, 2)$. Инверзија је заправо пар елемената који је „несортиран", односно сортиран опадајуће, па је самим тим једина пермутација без инверзија пермутација код које су елементи поређани у растућем поретку. Из овог разлога су инверзије важне у проблемима сортирања [5].

Табела инверзија пермутације $a_1a_2..a_n$, је низ $b_1b_2..b_n$, где је b_j број елемената лево од j који су већи од j. b_j се може посматрати као број инверзија у пермутацији чија друга компонента је j.
На пример, табела инверзија пермутације

$$5\ 9\ 1\ 8\ 2\ 6\ 4\ 7\ 3 \tag{2.2}$$

је

$$2\ 3\ 6\ 4\ 0\ 2\ 2\ 1\ 0, \tag{2.3}$$

јер су 5 и 9 лево од 1 и већи су од 1; 5, 9 и 8 су лево од 2 и већи од 2, итд., укупно 20 инверзија (овај број се добија када се сумира табела инверзија).
По дефиницији увек важи: $0 \leq b_1 \leq n-1, 0 \leq b_2 \leq n-2, ..., 0 \leq b_{n-1} \leq 1, b_n = 0$.

Инверзија једнозначно одређује одговарајућу пермутацију тако да је за задату инверзију увек могуће добити њену пермутацију. Често је у проблемима са пермутацијама лакше радити са инверзијама него са њима одговарајућим пермутацијама [5].
На пример, од табеле инверзија (2.3), пермутација се добија на следећи начин:
Анализирају се елементи табеле инверзија са десне стране, од b_n ка b_1. 9 као највећи елемент

се напише први. Потом, пошто је $b_8 = 1$, значи да је 8 после 9 у пермутацији, и напише се десно од 9. Како је $b_7 = 2$, значи да су 8 и 9 лево од 7, па се 7 се напише после 8 и 9. Елементи су тренутно поређани 9 8 7. Даље, како је $b_6 = 2$, то значи да су 2 елемента лево од 6 већа од 6, па 6 мора стајати након 8 и 9, а пре 7. Поредак је дакле 9 8 6 7. Даље, како је $b_5 = 0$, 5 се поставља лево од 9, b_4 је 4 па се ставља након 4 већа броја од њега, дакле након 6, b_3 је 6 па се ставља након шест већих бројева од њега. Како је $b_2 = 3$, место му је након три већа броја од њега, дакле након 8, и обзиром да је $b_1 = 2$, ставља се након 5 и 9. Тако се добија пермутација (2.2).

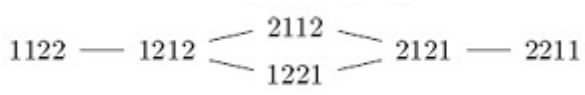
Алгоритам П користи нешто другачије дефинисане инверзије:
$c_1c_2...c_n$ је **табела инверзија** пермутације $a_1a_2...a_n$, где је c_j број елемената са десне стране j који су мањи од j.
Када се пермутације дефинишу на тај начин, пермутације из (2.1) имају следеће табеле инверзија:

0000	0010	0020	0120	0110	0100
0001	0011	0021	0121	0111	0101
0002	0012	0022	0122	0112	0102
0003	0013	0023	0123	0113	0103

Заменом места два суседна елемента у пермутацији број инверзија за ту пермутацију се мења тачно за ± 1, и ова чињеница се користи у алгоритму П.
Алгоритам је описан псеудокодом у наставку.

Алгоритам 2.1.2. *П (генерисање пермутација методом замене места суседних елемената)*
Улаз: *низ a дужине n*
Излаз: *све пермутације скупа* $\{a_1, a_2, \ldots, a_n\}$, *излистане у поретку „грејовске" секвенце.*
Пример: *за скуп* $\{1, 2, 3, 4\}$ *добија се:*
$1234, 1243, 1423, 4123,$
$4132, 1432, 1342, 1324,$
$3124, 3142, 3412, 4312,$
$4321, 3421, 3241, 3214,$
$2314, 2341, 2431, 4231,$
$4213, 2413, 2143, 2134.$
Напомена: *Алгоритам користи два помоћна низа: вектор инверзија пермутација* $c_1c_2\ldots c_n$, *где је* c_j *дефинисан као број елемената у пермутацији* $a_1a_2\ldots a_n$ *десно од* j *који су мањи од* j, *и за који важи* $0 \leq c_j < j, 1 \leq j \leq n$ *и вектор* $o_1o_2\ldots o_n$ *који дефинише правац мењања елемената* c_j.
begin

(1) **for** $1 \leq j \leq n$ **do**
(2) $c_j \leftarrow 0, o_j \leftarrow 1$ *[Иницијализација низова c и o]*
(3) **while true do**
(4) *Испис [испис пермутације* $a_1a_2\ldots a_n$*]*
(5) $j \leftarrow n, s \leftarrow 0$ *[Припрема за замену места.*
Следећи кораци одређују координату c_j *за који се se j променити,*
s је број индекса k већих од j, таквих да је $c_k = k - 1$*]*
(6) $q \leftarrow c_j + o_j$ *[Замена?]*
(7) **while** $q < 0$ **or** $q = j$ **do**
(8) **if** $q < 0$ **then** *[у овом тренутку дошло се до десног краја табеле c]*
(9) $o_j \leftarrow -o_j, j \leftarrow j - 1, q \leftarrow c_j + o_j$
(10) **if** $q = j$ **then** *[у овом тренутку дошло се до левог краја табеле c]*
(11) **if** $j = 1$ **then**
(12) **exit** *[нема наредне пермутације]*
(13) **else**
(14) $s \leftarrow s + 1, o_j \leftarrow -o_j, j \leftarrow j - 1, q \leftarrow c_j + o_j$
(15) *Заменити места елементима* a_{j-c_j+s} *i* a_{j-q+s}
(16) $c_j \leftarrow q$
end



Слика 2.1: Граф пермутација скупа $\{1, 1, 2, 2\}$ добијених методом простих замена

Уколико се неки од елемената улазног низа понавља, алгоритам у општем случају не гарантује налажење „грејовске" секвенце пермутација. Разлог лежи у томе што граф пермутација добијених заменом суседних елемената када се елементи понављају нема увек Хамилтонов пут. Слика 2.1 приказује пример пермутација са понављањем скупа $\{1, 1, 2, 2\}$. Ако се граном повежу парови пермутација такви да се једна од друге може добити заменом места суседних елемената, добија се граф на слици 2.1. Са слике се види да овај граф нема Хамилтонов пут.

Посебна погодност овог алгоритма је што генерише наизменично парне и непарне пермутације (**парне** пермутације су оне које имају паран број инверзија, а **непарне** - непаран), што омогућава да се модификује и да се не обилазе парне када су потребне само непарне и обрнуто.
Алгоритам је могуће унапредити - петља из линије 3 се извршава n пута. Могуће је направити модификацију тако да се у $(n-1)$ од n пролазака кораци увећавања променљиве s и мењања правца замена o (линије 13 - 14) прескачу.

2.1.4 Алгоритам T - транзиције алгоритма простих замена

За примене када је потребно рачунати пермутације истог скупа више пута, могуће је унапред припремити листу индекса елемената који учествују у заменама које алгоритам П изводи. Ова листа се потом користи сваки пут када су потребне пермутације тог скупа. Генерисање овакве листе замена је задатак алгоритма транзиције простих замена. За улазни параметар n алгоритам Т враћа низ индекса $t[1], t[2], ..., t[n!-1]$ такав да су акције алгоритма П еквивалентне узастопним заменама $a_{t[k]} \leftrightarrow a_{t[k]+1}$, $1 \leq k < n!$. Агоритам је применљив на све $n \geq 2$. Као и алгоритам П, и овај алгоритам је применљив на скупове различитих елемената.

Алгоритам 2.1.3. *Т (генерисање низа транзиција алгоритма простих замена)*
Улаз: n, $n \geq 2$
Излаз: *низ замена* $t[1], t[2], ..., t[n!-1]$ *такав да су акције алгоритма П еквивалентне узастопним заменама* $a_{t[k]} \leftrightarrow a_{t[k]+1}$, $1 \leq k < n!$
Пример: *за* $n = 4$ *niz t[1], t[2],..,t[23] je jednak:*

$$3, 2, 1, 3, 1, 2, 3, 1, 3, 2, 1, 3, 1, 2, 3, 1, 3, 2, 1, 3, 1, 2, 3$$

begin

$N \leftarrow n!, d \leftarrow N/2, t[d] \leftarrow 1, m \leftarrow 2$ *[Иницијализација]*
while true do *[Петља по m]*
if $m = n$
exit *[нема наредне пермутације]*
$m \leftarrow m + 1, d \leftarrow d/m, k \leftarrow 0$
$k \leftarrow k + d, j \leftarrow m - 1$
while $j > 0$ *[Генерисање вредности* $t[k]$ *опадајуће]*
$t[k] \leftarrow j, k \leftarrow k + d, j \leftarrow j - 1$
$t[k] \leftarrow t[k] + 1$ *[Увећавање вредности* $t[k]$ *и* d*]*
$k \leftarrow k + d, j \leftarrow 1$
while $j < m$ *[Генерисање вредности* $t[k]$ *растуће]*
$t[k] \leftarrow j, k \leftarrow k + d, j \leftarrow j + 1$
while $k < N$
$k \leftarrow k + d, j \leftarrow m - 1$
while $j > 0$
$t[k] \leftarrow j, k \leftarrow k + d, j \leftarrow j - 1$
$t[k] \leftarrow t[k] + 1$
$k \leftarrow k + d, j \leftarrow 1$
while $j < m$
$t[k] \leftarrow j, k \leftarrow k + d, j \leftarrow j + 1$
end

2.1.5 Алгоритам A - "alphametics"

Једна од многобројних примена пермутација је у проблемима званим *алфаметике (енг. alphametics)*. Овај термин је сковао J. A. X. Хантер (J. A. H. Hunter) 1955. године [6], а ради се о проблемима следећег облика:
Које вредности цифара 0 - 9 је потребно да узимају слова да би важила једнакост:

SEND + MORE = MONEY, или
VIOLIN + VIOLIN + VIOLA = TRIO + SONATA, и еквивалентно:
2(VIOLIN) + VIOLA - TRIO - SONATA = 0.

Описани проблеми се могу једноставно решавати ручно, а овде ће бити показано како алгоритми за генерисање пермутација могу бити корисни у њиховом решавању на примеру:

$$2(VIOLIN) + VIOLA - TRIO - SONATA = 0. \tag{2.4}$$

За свако слово у оваквом проблему се рачуна „потпис", тако што се реши једначина у систему са основом 10, у којој се то слово замени бројем 1, а сва остала слова добију вредност 0. Нумеричка вредност која се добије као решење једначине је „потпис" за то слово. На пример, потпис за слово **I** је једнак:

2(010010) + 01000 - 0010 - 000000,

односно 21010. Уколико низ $s_1 s_2 ... s_{10}$ садржи потписе за слова из алфаметике (2.4): (V, I, O, L, N, A, T, R, S, X) редом, онда ти потписи имају следеће вредности (у декадном систему):

$$s_1 = 210000, s_2 = 21010, s_3 = -7901, s_4 = 210, s_5 = -998,$$
$$s_6 = -100, s_7 = -1010, s_8 = -100, s_9 = -100000, s_{10} = 0.$$

На овај начин, проблем се своди на налажење свих пермутација $a_1 ... a_{10}$ цифара $\{0, 1, ..., 9\}$ таквих да је:

$$a \cdot s = \sum_{j=1}^{10} a_j s_j = 0$$

Уколико у проблему има мање од 10 различитих слова, додају се додатна слова да би укупан број цифара био 10. Додатна слова добијају потпис 0. У примеру је тако вештачки додато слово X. Потребан је и додатни услов - наиме, бројеви у алфаметици не би требали да имају 0 као почетну цифру, тј. решења облика:

7316+0823 = 08139, или 5731+0647=06378

се не сматрају валидним. Стога се уводи скуп првих слова **F**, такав да је

$$\text{за свако } j \in F, a_j \neq 0 \tag{2.5}$$

У овом примеру, елементи скупа F су слова V, T и S, тј. $F = \{1, 7, 9\}$.
Овако задат проблем могуће је једноставно решити, пролазећи кроз свих 10! пермутација цифара $0, 1, 2, ..., 9$ и рачунајући вредност једначине која се добија када се слова замене вредностима својих потписа. За ово се може користити унапред припремљена табела транзиција која се добија алгоритмом T из 2.1.4. Алгоритам који следи решава дати проблем полазећи од низа потписа $\{s_1, s_2, ..., s_{10}\}$ и скупа првих слова F. Алгоритам исписује пермутације цифара које су решење задате алфаметике.

Алгоритам 2.1.4. *A (примена алгоритма T у алфаметикама)*
Улаз: *низ потписа алфаметике* $s[1]s[2]...s[10]$*, низ који представља скуп првих слова F, и низ транзиција* t *величине* $10! - 1$ *који је повратна вредност алгоритма T и чува индексе замена узастопних елемената за пермутације 10 елемената.*
Излаз: *све пермутације скупа* $\{1, 2, \ldots, 10\}$*, које представљају решење улазне алфаметике.*
Напомена: *Низ* δ *служи да чува разлике између узастопних елемената у низу потписа, променљива* v *је сума потписа алфаметике*
Пример: *2(VIOLIN) + VIOLA - TRIO - SONATA = 0. За задати низ потписа алфаметике s={210000, 21010, -7901, 210, -998, -100, -1010, -100, -100000,0}, и скуп првих слова F={1, 7, 9} алгоритам исписује следеће пермутације:*
3 5 4 6 2 8 1 9 7 0, 3 5 4 6 2 9 1 8 7 0, 1 7 6 4 8 5 2 0 3 9, 1 7 6 4 8 0 2 5 3 9, односно, слова скупа (V, I, O, L, N, A, T, R, S, X) могу узимати следеће вредности да би задата једначина била тачна:
V = 3, I = 5, O = 4, L = 6, N = 2, A = 8, T = 1, R = 9, S = 7, X = 0, или
V = 3, I = 5, O = 4, L = 6, N = 2, A = 9, T = 1, R = 8, S = 7, X = 0, или
V = 1, I = 7, O = 6, L = 4, N = 8, A = 5, T = 2, R = 0, S = 3, X = 9,или
V = 1, I = 7, O = 6, L = 4, N = 8, A = 0, T = 2, R = 5, S = 3, X = 9.
begin

for $1 \leq j \leq 10$
$a_j \leftarrow j - 1$
for $1 \leq j < 10$
$\delta_j \leftarrow s_{j+1} - s_j$
$v \leftarrow \sum_{j=1}^{10}(j-1)s_j$,
$k \leftarrow 1$
while true do
if $v = 0$ **and** *Vazi*(2.5) **then** *[Испитивање да ли је алфаметика тачна за тренутну пермутацију*
када елементи низа s узимају вредности елемената пермутације $a_1...a_{10}$*;*
Потребно је и да буде испуњен услов (2.5)*]*
Ispis [испис пермутације $a_1a_2\ldots a_{10}$ *која задовољава задату алфаметику]*
if $k = 10!$ **then**
exit *[Све пермутације су проверене]*
else
[Добијање наредне пермутације заменом елемената a_{t_k} *са* $a_{t_{k+1}}$*]*
$j \leftarrow t_k$
$v \leftarrow v - (a_{j+1} - a_j)\delta_j$ *[*v *се умањује за разлику елемената* a_{j+1} *и* a_j*]*
Zameniti mesta elementima a_{j+1} *i* a_j
$k \leftarrow k + 1$
end

Уколико алфаметика има јединствено решење, каже се да је *чиста*. Као што се види из примера, алфаметика (2.4) није чиста, јер има четири решења:

$$\begin{aligned} 2(354652) &+ 35468 - 1954 - 742818 = 0,\\ 2(354652) &+ 35469 - 1854 - 742919 = 0,\\ 2(176478) &+ 17645 - 2076 - 368525 = 0,\\ 2(176478) &+ 17640 - 2576 - 368020 = 0. \end{aligned}$$

Може се десити и да алфаметика јесте чиста, а да описани алгоритам врати две различите пермутације које је решавају. Ово је на пример, алфаметика SEND + MORE = MONEY. Разлог

за то је што алфаметика има 8 различитих слова, па се морају додати два „лажна" потписа $s_9 = s_{10} = 0$ за додатна слова, која не учествују у алфаметици. У општем случају, алфаметика са m различитих слова има $10-m$ „лажних" потписа, па ће свако од њених решења бити нађено $(10-m)!$ пута. Ово се може избећи постављањем услова $a_{m+1} < \cdots < a_{10}$ у алгоритам при испису пермутација.

2.1.6 Алгоритми засновани на композицији пермутација

Велика група алгоритама за генерисање пермутација се заснива на мултипликативним својствима пермутација, због чега је пре свега потребно увести најосновније термине везане за композицију (множење) пермутација. Ову групу алгоритама одликује и то што се пермутовање одвија са леве на десну страну, тј. пермутације првих $0, 1, ..., k-1$ елемената се посећују у току првих $k!$ корака. Једна од оваквих схема је „обрнути колекс поредак" (*енг. reverse colex order*), који представља обрнут лексикографски поредак. На пример, за $n = 4$, обрнути колекс поредак гласи:

$$\begin{aligned} &0123, 1023, 0213, 2013, 1203, 2103,\\ &0132, 1032, 0312, 3012, 1302, 3102,\\ &0231, 2031, 0321, 3021, 2301, 3201,\\ &1230, 2130, 1320, 3120, 2310, 3210. \end{aligned} \tag{2.6}$$

Уколико се стрингови читају са десне на леву страну, добија се 3210, 3201,...,0123, што је обрнут лексикографски поредак - од последњег елемента ка првом. Лексикографски поредак за скуп {0, 1, 2, 3} гласи:

0123, 0132, 0213, 0231, 0312, 0321,
1023, 1032, 1203, 1230, 1302, 1320,
2013, 2031, 2103, 2130, 2301, 2310,
3012, 3021, 3102, 3120, 3201, 3210.

Мултипликативна својства пермутација се заснивају на томе да се пермутација посматра као пресликавање елемената. За представљање пермутација могуће су две нотације:

- Дволинијска нотација
$$\alpha = \begin{pmatrix} 0\,1\,2\,3\,4\,5 \\ 2\,5\,0\,1\,4\,3 \end{pmatrix}$$
- Циклична нотација
$$\alpha = (0\ 2)\,(1\ 5\ 3)$$

У представљеном пресликавању, 0 се слика у 2, 1 се слика у 5, 2 се слика у 0, 3 у 1, 4 у 4, и 5 се слика у 3.
Елемент 4 се пресликава у самог себе па се каже да је он *фиксан*, односно да га пермутација фиксира. Фиксни елементи чине цикл дужине 1, и он се не записује. Уколико пермутација слика сваки елемент у самог себе, назива се *идентичка* пермутација, и обично се означава са (1), или (). Једна иста пермутација се може представити цикличном нотацијом на више начина.
Производ пермутација је примена једне пермутације након друге. На пример,

$$\alpha\beta = \begin{pmatrix} 0\,1\,2\,3\,4\,5 \\ 2\,5\,0\,1\,4\,3 \end{pmatrix} \begin{pmatrix} 0\,1\,2\,3\,4\,5 \\ 5\,4\,3\,2\,1\,0 \end{pmatrix} = \begin{pmatrix} 0\,1\,2\,3\,4\,5 \\ 3\,0\,5\,4\,1\,2 \end{pmatrix}$$

Множење пермутација изводи се на следећи начин: пермутација α пресликава 0 у 2, а пермутација β пресликава 2 у 3, па се 0 пресликавањем $\alpha\beta$ пресликава у 3. 1 се слика у 5, а 5 у 0 па се 1 слика у 5. 2 се слика у 0, а 0 у 5, па се 2 слика у 5. 3 се слика у 1, а 1 у 4, па се 3 слика у 4.4 се слика у 4, а 4 у 1, па се 4 слика у 1, и слика 5 је 3, а слика 3 је 2, па се 5 слика у 2.
Производ пермутација није комутативан. Применити β на α се означава са $\beta\alpha$, а не $\alpha\beta$.
Непразан скуп пермутација који је затворен за операцију множења је *група*.

Фамилија подскупова $S_1, S_2, ...$ групе G која има својство да садржи тачно једну пермутацију σ_{kj} која слика k у j и фиксира вредности свих елемената већих од k, кад год G садржи такву пермутацију, се назива *Симсова табела*, уведена од стране математичара Чарлса Симса (Charles Sims)[7]. Пресликавање σ_{kk} може бити идентичка пермутација, а за $0 \leq j < k$ σ_{kj} може бити било која пермутација која одговара условима.
Симсове табеле су погодне за репрезентацију група пермутација у рачунару, јер је доказано следеће:
Лема С. Уколико је $S_1, S_2, ..., S_{n-1}$ Симсова табела групе G, онда сваки елемент α из G има јединствену репрезентацију:

$$\alpha = \sigma_1\sigma_2\cdots\sigma_{n-1}, \text{ где је } \sigma_k \in S_k \text{ за } 1 \leq k < n$$

Ово води до једноставног начина за генерисање свих пермутација групе - проласком кроз све пермутације облика:

$$\sigma(1, c_1)\sigma(2, c_2)\cdots\sigma(n-1, c_{n-1}),$$

где је $\sigma(k, c_k)$ $(c_k + 1)$-и елемент S_k, за $0 \leq c_k < s_k = |S_k|$ и $1 \leq k < n$,
користећи неки од алгоритама за обилазак свих $(n-1)$-торки $(c_1, ..., c_{n-1})$ за одговарајуће *основе* $(s_1, ..., s_{n-1})$ у **систему са мешовитим основама**.
Бројевни систем са мешовитим основама је нестандардни позициони бројевни систем у коме се основе разликују од позиције до позиције. Уобичајен је, на пример, за изражавање времена. На пример, период од 5 дана, 7 сати, 45 минута, 15 секунди и 500 милисекунди би се као број у систему са мешовитим основама написао као [8]:

$$\begin{bmatrix} 5, & 7, & 45, & 15, & 500 \\ 7, & 24, & 60, & 60, & 1000 \end{bmatrix}$$

За проблем генерисања свих пермутација скупа $\{0, 1, ..., n-1\}$ сваки скуп S_k Симсове табеле треба да садржи $k+1$ елемент $\sigma(k, 0), \sigma(k, 1), ..., \sigma(k, k)$ где је $\sigma(k, 0)$ идентичка пермутација, а остале сликају k у вредности $\{0, ..., k-1\}$ у неком редоследу.
Свака оваква Симсова табела дефинише један генератор пермутација, по општем алгоритму који је описан у следећој тачки.

2.1.7 Алгоритам Г - општи агоритам за генерисање пермутација

Алгоритам Г је општи алгоритам за генерисање пермутација, а редослед којим се пермутације генеришу зависи од Симсове табеле која се користи.

Алгоритам 2.1.5. *Г (Општи алгоритам за генерисање пермутација)*
Улаз: *Симсова табела $S_1, S_2, ..., S_{n-1}$ у којој сваки S_k има $k+1$ елемената $\sigma(k,j)$ као што је описано у претходној тачки*
Излаз: *Све пермутације $a_0a_1...a_{n-1}$ скупа $\{0,1,...,n-1\}$*
Напомена: *Редослед којим се пермутације обилазе (генеришу) зависи од Симсове табеле која се користи. Алгоритам користи додатну контролну табелу $c_nc_{n-1}...c_1$.*
begin

(1) **for** $0 \leq j < n$ **do**
(2) $a_j \leftarrow j, c_{j+1} \leftarrow 0$ *[Иницијализација]*
(3) **while true do**
(4) *Ispis [Испис пермутације $a_0a_1...a_{n-1}$]*
[На овом месту, број са мешовитом основом

$$\begin{bmatrix} c_{n-1}, & \ldots, & c_2 & c_1 \\ n, & \ldots, & 3 & 2 \end{bmatrix}$$

представља број генерисаних пермутација.]
(5) $k \leftarrow 1$ *[Увећавање броја са мешовитом основом за 1]*
(6) **if** $c_k = k$
(7) **do**
(8) $c_k \leftarrow 0, k \leftarrow k+1$
(9) **while not** $c_k < k$
(10) $c_k \leftarrow c_k + 1$
(11) **if** $k = n$
(12) **exit** *[Вредност броја са мешовитом основом је n!, тј. све пермутације су посећене]*
(13) *PrimeniPermutaciju* $\tau(k,c_k)\omega(k-1)^-$ *на* $a_0a_1...a_{n-1}$ *[Пермутовање - генерисање нове пермутације применом композиције пермутација]*
end

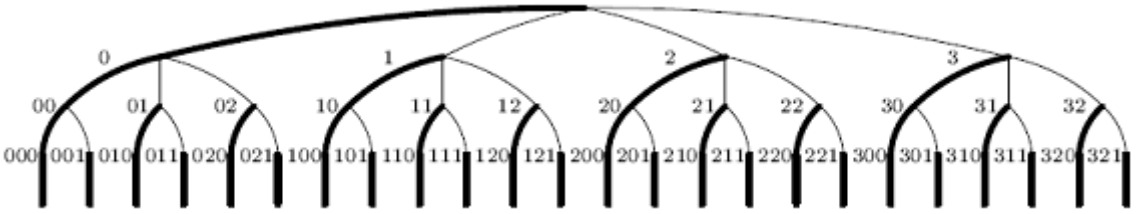
Пресликавања τ и ω се дефинишу на следећи начин:

$$\tau(k,j) = \sigma(k,j)\sigma(k,j-1)^- , \quad 1 \leq j < k$$

$$\omega(k) = \sigma(1,1)...\sigma(k,k),$$

σ^- представља инверзну пермутацију.
Кораци увећавања броја са мешовитом основом c (линије 5 - 10 у псеудокоду) и пермутовања (линија 13) обезбеђују да је $a_0a_1...a_{n-1}$ пермутација добијена као производ $\sigma(1,c_1)\sigma(2,c_2)\cdots\sigma(n-1,c_{n-1})$. Лема С је доказ да се свака пермутација посећује само једном.

Слика 2.2 приказује како алгоритам ради за $n = 4$. Слика приказује стабло вредности контролне табеле $c_3c_2c_1$. Свака пермутација $a_0a_1a_2a_3$ одговара једном контролном стрингу, где је $0 \leq c_3 \leq 3$, $0 \leq c_2 \leq 2$, $0 \leq c_1 \leq 1$. Чворови стабла који су означени само једном цифром c_3 одговарају пермутацијама $\sigma(3,c_3)$ коришћене Симсове табеле. Чворови стабла који су означени са две цифре c_3c_2 одговарају пермутацијама $\sigma(2,c_2)\sigma(3,c_3)$. Чворови c_3 и c_30 су еквивалентни, јер је $\sigma(2,0)$ идентичко пресликавање. На слици су из тог разлога повезани подебљаном линијом. На исти начин, чворови c_3c_2 и c_3c_20 су еквивалентни, јер је $\sigma(1,0)$ такође идентичко пресликавање. Додавањем 1 броју са мешовитом основом прелази се са тренутног чвора на следећи чвор у *корен-леви-десни* обиласку стабла, а корак пермутовања прелази на следећу пермутацију у складу са тим. На пример, након што се бројач $c_3c_2c_1$ промени са 121 на 200, у кораку пермутовања текућа пермутација се множи са:



Слика 2.2: Обилазак стабла за $n = 4$

$$\tau(3,2)\omega(2)^- = \tau(3,2)\sigma(2,2)^-\sigma(1,1)^-$$

Множењем са $\sigma(1,1)^-$ прелази се из чвора 121 у чвор 12, множењем са $\sigma(2,2)^-$ прелази се из чвора 12 у чвор 1, а множењем са $\tau(3,2) = \sigma(3,2)\sigma(3,1)^-$ прелази се из чвора 1 у чвор $2 \equiv 200$, следбеник чвора 121 у *корен-леви-десни* обиласку стабла.
Као што је речено, наведени алгоритам је општи, а редослед генерисања пермутација зависи од Симсове табеле која се користи. Постоји велики број генератора пермутација у зависности од коришћене Симсове табеле, а овде ће бити поменуто неколико најбитнијих. На пример, уколико се пресликавања Симсове табеле, $\sigma(k,j)$ дефинишу на следећи начин:

$$\sigma(k,j) = (k-j \quad k-j+1 \quad \cdots \quad k), \tag{2.7}$$

пермутације се обилазе у обрнутом колекс поретку (2.6). Пресликавања τ и ω тада изгледају овако:

$$\tau(k,j) = (k-j \quad k) \tag{2.8}$$

$$\omega(k) = (0\ 1)(0\ 1\ 2)\ \cdots\ (0\ 1\ \ldots\ k)\ =\ (0\ k)\ (1\ k-1)\ (2\ k-2)\ \cdots\ = \phi(k). \tag{2.9}$$

Пресликавање $\phi(k)$ обрће $a_0 \ldots a_k$ у $a_k \ldots a_0$. $\omega(k)$ је једнак $\omega(k)^-$, јер је $\phi(k)^2 = ()$, тј. идентичко пресликавање.
Други специјалан случај алгоритма Г потиче од P. J. Орд-Смита (R. J. Ord-Smith), чији алгоритам се добија када се користи следеће $\sigma(k,j)$ пресликавање [17]:

$$\sigma(k,j) = (k\ \ \ldots\ \ 1\ \ 0)^j \tag{2.10}$$

Пресликавања τ и ω тада изгледају овако:

$$\tau(k,j) = (k\ \ldots\ 1\ 0)$$

$$\omega(k) = (0\ k)\ (1\ k-1)\ (2\ k-2)\ \cdots\ = \phi(k).$$

Пресликавање ω је исто као у претходном методу јер је $\sigma(k,k) = (0\ 1\ \ldots\ k)$. Посебна погодност овог метода је што тада корак пермутовања, тј. множења пермутација (линија 13) не зависи од c_k, јер је:

$$\tau(k,c_k)\omega(k-1)^- = (k\ \ldots\ 1\ 0)\phi(k-1)^- = \phi(k),$$

односно, корак пермутовања алгоритма Г у овом случају представља једноставан сет замена $a_0 \leftrightarrow a_k$, $a_1 \leftrightarrow a_{k-1}$,... што је брза операција уколико је k мало.
Још ефикаснији је приступ Б. Р. Хипа (B. R. Heap)[18] , према коме је корак пермутовања алгоритма Г (линија 13) једнак једној замени места елемената, слично алгоритму простих

замена (тачка 2.3), али не на суседним елементима. По Хиповом предлогу пресликавање $\tau(k, c_k)\omega(k-1)^-$ којим се множи пермутација $a_1...a_n$ у кораку пермутовања изгледа овако:

$$\tau(k, c_k)\omega(k-1)^- = \begin{cases} (k\ 0), \text{ ако је } k \text{ паран} \\ (k\ j-1), \text{ ако је } k \text{ непаран} \end{cases}$$

Пермутовање је у овом случају увек замена $a_k \leftrightarrow a_0$, осим када је $k = 3, 5, ...$, а вредност k је у 5 од 6 корака 1 или 2.

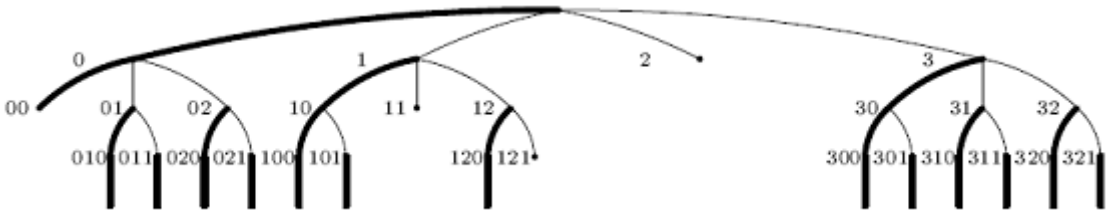
2.1.8 Алгоритам Г2 - генерисање пермутација са прескакањем нежељених суфикса

Једна од предности алгоритма Г је та што генерише све пермутације подниза $a_0a_1...a_{k-1}$ пре него што почне да мења a_k, а затим мења a_k на сваких $k!$ пролазака кроз петљу, итд. Ово омогућава да уколико је суфикс $a_k...a_{n-1}$ небитан за проблем на коме се ради, пермутације које се завршавају њиме буду прескочене. Да би се ово постигло, потребно је проширити корак обиласка пермутације (линија 4) алгоритма Г и додати додатну иницијализацију $k = n - 1$. Псеудокод овако измењеног алгоритма следи.

Алгоритам 2.1.6. *Г2 (Алгоритам за генерисање пермутација са прескакањем нежељених суфикса)*
Улаз: *Симсова табела $S_1, S_2, ..., S_{n-1}$ у којој сваки S_k има $k+1$ елемената $\sigma(k,j)$ као што је описано у тачки 2.1.7*
Излаз: *Све пермутације $a_0a_1...a_{n-1}$ скупа $\{0,1,...,n-1\}$ које се не завршавају нежељеним суфиксом*
Напомена: *Редослед у коме се пермутације обилазе зависи од Симсове табеле која се користи. Алгоритам користи додатну контролну табелу $c_nc_{n-1}...c_1$.*
begin

(1) **for** $0 \leq j < n$ **do**
(2) $a_j \leftarrow j, c_{j+1} \leftarrow 0$ *[Иницијализација]*
(3) $k \leftarrow n-1$
(4) **while true do**
(5) **do**
(6) **if** $a_k...a_{n-1}$ *je neprihvatljiv sufiks [Провера да ли је суфикс прихватљив]*
(7) **if** $c_k = k$ *[Прескакање пермутација које се завршавају нежељеним суфиксом]*
(8) **do**
(9) *Primeniti* $\sigma(k,k)^-$ *na* $a_0...a_{n-1}$
(10) $c_k \leftarrow 0, k \leftarrow k+1$
(11) **while not** $c_k < k$
(12) **if** $k = n$
(13) **break**
(14) $c_k \leftarrow c_k + 1$
(15) *Primeniti* $\tau(k, c_k)$ *na* $a_0a_1...a_{n-1}$
(16) **else**
(17) $k \leftarrow k-1$
(18) **while** $k > 0$
(19) *Ispis [Испис пермутације* $a_0a_1...a_{n-1}$*]*
(20) $k \leftarrow 1$ *[Увећавање броја са мешовитом основом за 1]*
(21) **if** $c_k = k$
(22) **do**
(23) $c_k \leftarrow 0, k \leftarrow k+1$
(24) **while not** $c_k < k$
(25) $c_k \leftarrow c_k + 1$
(26) **if** $k = n$
(27) **exit** *[Вредност броја са мешовитом основом је n!, тј. све пермутације су посећене]*
(28) *PrimeniPermutaciju* $\tau(k, c_k)\omega(k-1)^-$ *na* $a_0a_1...a_{n-1}$ *[Пермутовање - генерисање нове пермутације применом композиције пермутација]*
end

Овако измењени алгоритам је сложенији, осим пресликавања $\tau(k,j)\omega(k-1)^-$ сада је потребно рачунати и $\tau(k,j)$ и $\sigma(k,k)$, али измењени алгоритам може радити знатно брже. Слика 2.3 приказује пример рада алгоритма за $n = 4$, за пермутације $a_0a_1a_2a_3$ скупа $\{0,1,2,3\}$, када пермутације са неодговарајућим суфиксима одговарају префиксима контролног стринга 00, 11, 121 и 2. Као и у примеру алгоритма Г, сваки чвор одговара једном контролном стрингу $c_3c_2c_1$. У општем случају, сваки суфикс $a_k...a_{n-1}$ пермутације $a_0...a_{n-1}$ одговара префиксу $c_n...c_k$ контролног стринга $c_n...c_1$, јер пермутације $\sigma(1,c_1), ..., \sigma(k-1,c_{k-1})$ не утичу на $a_k...a_{n-1}$. У линији 15, текућа пермутација се множи са $\tau(k,j)$ да би се прешло са чвора $c_{n-1}...c_{k+1}j$ на



Слика 2.3: Обилазак стабла применом алгоритма Г2 за $n = 4$ када неодговарајући суфикси одговарају префиксима контролног стринга 00, 11, 121 и 2. Чворови стабла који одговарају пермутацијама које одговарају овим суфиксима се не обилазе.

његовог десног брата $c_{n-1}...c_{k+1}j+1$, а у линији 9 тренутна пермутација се множи са $\sigma(k,k)^-$ да би се прешло из чвора $c_{n-1}...c_{k+1}$ у његовог оца. Тако на пример, да би се прешло из неодговарајућег чвора 121 у његовог *корен-леви-десни* следбеника, алгоритам множи текућу пермутацију са $\sigma(1,1)^-$, $\sigma(2,2)^-$, и $\tau(3,2)$. Множењем са $\sigma(1,1)^-$ прелази се из чвора 121 у чвор 12, множењем са $\sigma(2,2)^-$ из чвора 12 у чвор 1, а множењем са $\tau(3,2) = \sigma(3,2)\sigma(3,1)^-$ прелази се из чвора 1 у чвор $2 \equiv 200$, следбеник чвора 121 у *корен-леви-десни* обиласку стабла. Након што се 2 одбаци као неодговарајући чвор, множењем са $\tau(3,3)$ прелази се у чвор 3, итд.. Ова верзија алгоритма Г за прескакање пермутација које се завршавају одређеним суфиксом се може применити на алфаметика проблеме из тачке 2.1.5. Приступ из алгоритма А користи грубу силу и покушава да реши алфаметику тако што испробава свих 10! пермутација 10 цифара, у сваком покушају референцирајући меморију 6 пута (за замену места суседних елемената да би се генерисала наредна пермутација). Цео тај процес узима око 22MB, невезано за проблем који се решава јер се меморија троши на генерисање табеле замена која је константа за све алфаметика проблеме. Уколико се користи проширени алгоритам Г са прескакањем суфикса, и Хипов метод за Симсову табелу, за пример (2.4) из тачке 2.1.5 потребно је мање од 128 KB, што је 170 пута брже. Овај приступ међутим зависи од конкретног примера. У алфаметика проблемима, *корен-леви-десни* обилазак стабла се углавном састоји од множења са $\tau(k,c_k)$ који врше померање на десно, и знатно мање - множења са $\sigma(k,k)^-$, која померају навише. τ кораци доминирају јер се врло мали број комплетних пермутација посети, а сваком кораку множења са $\sigma(k,k)^-$ претходе множења са $\tau(k,1),\tau(k,2),...,\tau(k,k)$. Одавде следи да Хипов метод, који веома оптимизује пресликавање $\tau(k,j)\omega(k-1)^-$ тако да је свака транзиција у кораку пермутовања једноставна транспозиција, није погодан за алгоритам Г проширен за прескакање суфикса, осим уколико се не одбацује јако мало суфикса. Приступ (2.7) који генерише пермутације у обрнутом колекс поретку је погоднији када се прескачу суфикси, јер је ту $\tau(k,j)$ увек проста транспозиција. Користећи овај приступ за трио соната проблем (2.4), потребно је само 97KB. У просеку, Хипов метод је око 60 пута бржи од приступа грубом силом, док је обрнути колекс поредак приступ бржи око 80 пута.

2.1.9 Алгоритам X[1] - генерисање пермутација које задовољавају скуп услова лексикографским редоследом

Још један алгоритам који генерише пермутације које задовољавају одређени скуп услова је алгоритам X. Овај алгоритам је унапређена верзија алгоритма Л, генерише пермутације у лексикографском поретку, али за разлику од низа који користи алгоритам Л, алгоритам X

[1] овде се X односи на енглеско слово X

користи листу. Ово га чини погодним за примене када се очекује да ће доста пермутација бити прескочено јер не задовољавају услове примене. Алгоритам генерише све пермутације $a_1a_2...a_n$ скупа $\{1, 2, ..., n\}$ које пролазе низ тестова:

$$t_1(a_1), t_2(a_1, a_2), ..., t_n(a_1, a_2, ..., a_n),$$

посећујући их у лексикографском поретку. Користи помоћну табелу $l_0, l_1, ..., l_n$ за одржавање цикличне листе неискоришћених елемената, тако да ако су тренутно неискоришћени елементи:

$$\{1, ..., n\}\backslash\{a_1, ..., a_k\} = \{b_1, ..., b_{n-k}\}, \text{ где је } b_1 < ... < b_{n-k}, \text{ тада је}$$

$$l_0 = b_1, l_{b_j} = b_{j+1} \text{ за } 1 \leq j < n - k, \, l_{b_{n-k}} = 0$$

Алгоритам користи додатни помоћни низ $u_1...u_n$ за поништавање претходне операције (*енг. undo*) на низу l.

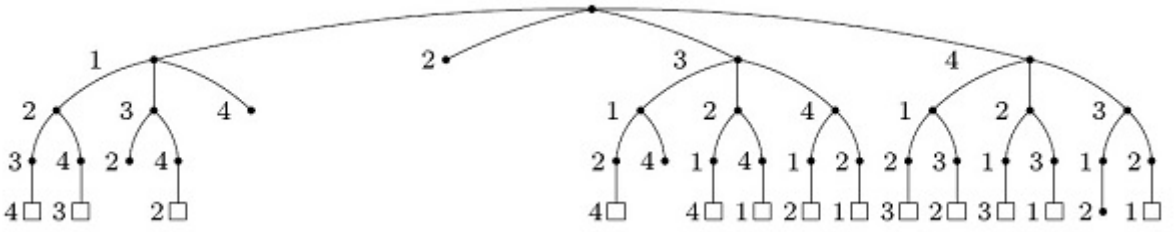
Алгоритам 2.1.7. *X (Алгоритам за генерисање пермутација које задовољавају скуп услова)*
Улаз: *n, функције* $t_i(a_1, \ldots, a_i)$ *које представљају тестове за префиксе* $(a_1, \ldots, a_i)$, $1 \leq i \leq n$
Излаз: *Све пермутације* $a_1a_2...a_n$ *скупа* $\{1, 2, ..., n\}$ *које пролазе тестове задате функцијама* $t_i(a_1, \ldots, a_i)$, $1 \leq i \leq n$ *на начин описан горе*
Напомена: *Алгоритам користи помоћну табелу l која представља цикличну листу неискоришћених елемената, и низ и који служи за поништавање претходне операције.*
begin

for $0 \leq k < n$ **do** *[Иницијализација]*
$l_k \leftarrow k + 1$
$l_n \leftarrow 0, k \leftarrow 1$
while true do
$p \leftarrow 0, q \leftarrow l_0$ *[Улазак на ниво k - тестирање услова Tk над поднизом* $a_1 \ldots a_k$
while true do
$a_k \leftarrow q$ *[Тестира се услов на поднизу* $a_1 \ldots a_k$*]*
if $t_k(a_1, ..., a_k)$ *је* **false** *[Уколико је нетачан]*
$p \leftarrow q, q \leftarrow l_p$ *[Повећавање* a_k *да би се покушало са другом вредношћу]*
if $q \neq 0$ **then**
continue *[Повратак на почетак петље, да би се услов* T_k *тестирао за друхе вредности* a_k*]*
while true
$k \leftarrow k - 1$ *[Смањивање k]*
if $k = 0$ **then**
exit *[Тестиране су све пермутације]*
else
$p \leftarrow u_k, q \leftarrow a_k, l_p \leftarrow q$ *[У супротном, корак уназад,*
и наставак тестирања услова са другим вредностима a_k*]*
$p \leftarrow q, q \leftarrow l_p$ *[Повећавање* a_k *да би се покушало са другом вредношћу]*
if $q \neq 0$ **then**
break
else if $k = n$
[Уколико је свих n услова тачно, исписује се пермутација која их задовољава,
и покушава се са пермутацијом која задовољава T_{k-1} *за* $a_1 \ldots a_{k-1}$ *са новом вредношћу* a_k*]*
Ispis permutacije $a_1a_2...a_n$ *[Испис пермутације која задовољава свих n услова]*
while true do
$k \leftarrow k - 1$
if $k = 0$ **then**
exit *[Тестиране су све пермутације]*
else
$p \leftarrow u_k, q \leftarrow a_k, l_p \leftarrow q$
$p \leftarrow q, q \leftarrow l_p$ *[Повећавање* a_k *да би се покушало са другом вредношћу]*
if $q \neq 0$ **then**
break
else
$u_k \leftarrow p, l_p \leftarrow l_q, k \leftarrow k + 1$ *[Повећавање k -* T_{k-1} *је испуњен за пермутацију* $a_1 \ldots a_{k-1}$
и прелази се на ниво k]
break
end

Овај алгоритам осмислио је М. Ц. Ер (М. С. Er)[19]. Може се применити на алфаметика

проблеме изложене у 2.1.5. Алгоритам решава проблем (2.4) користећи 49KB меморије. Ово га чини 165 пута бржим од приступа грубом силом из наведене тачке.

Слика 2.4 приказује стабло које алгоритам X имплицитно обилази када је $n = 4$ - штампају се све пермутације осим оних које почињу са 132, 14, 2, 314 и 4312 (ове пермутације не задовољавају услове задате тестовима).



Слика 2.4: Стабло које алгоритам X имплицитно обилази када је $n = 4$ - штампају се све пермутације осим оних које почињу са 132, 14, 2, 314 и 4312.

2.1.10 Алгоритам X - дуални алгоритам за генерисање пермутација

Као што се на основу Леме С из тачке 2.1.6 свака пермутација групе Г и њене Симсове табеле $S_1, S_2, ..., S_{n-1}$може представити као производ $\sigma(1)\cdots\sigma(n-1)$, за $\sigma(k) \in S_k$, може се показати да је тачно и дуално тврђење, тј. да се свака пермутација може представити и као:

$$\alpha = \sigma_{n-1}^{-}\cdots\sigma_2^{-}\sigma_1^{-}, \sigma_k \in S_k, 1 \leq k < n. \tag{2.11}$$

На овом тврђењу се заснива још једна велика група алгоритама за генерисање пермутација. Ако је G група пермутација, свака пермутација се може представити као:

$$\sigma(n-1, c_{n-1})^{-}\cdots\sigma(2, c_2)^{-}\sigma(1, c_1)^{-}, \text{ где је } 0 \leq c_k \leq k \text{ за } 1 \leq k < n\ .$$

Алгоритам X користи исти механизам као и алгоритам Г, пермутације $\sigma(k, j)$ су исте као у алгоритму Г, али су пресликавања τ и ω сада различита. Осим тога, због другачије дефинисаног производа пермутација, потребно је да се c_{n-1} сада најбрже мења, а c_1 најспорије.

Алгоритам 2.1.8. *X (Дуални алгоритам за генерисање пермутација)*
Улаз: *Симсова табела $S_1, S_2, ..., S_{n-1}$ у којој сваки S_k има $k+1$ елемената $\sigma(k,j)$ као што је описано у тачки 2.1.6*
Излаз: *Све пермутације $a_0a_1...a_{n-1}$ скупа $\{0,1,...,n-1\}$*
Напомена: *Редослед којим се пермутације обилазе зависи од Симсове табеле која се користи. Алгоритам користи додатну контролну табелу $c_0...c_{n-1}$ која служи за представљање броја са мешовитом основом c.*
begin

(1) **for** $0 \leq j < n$ **do**
(2) $a_j \leftarrow j, c_j \leftarrow 0$ *[Иницијализација]*
(3) **while true do**
(4) *Ispis [Испис пермутације $a_0a_1...a_{n-1}$]*
[На овом месту, број са мешовитом основом

$$\begin{bmatrix} c_1, & c_2, \ldots, & c_{n-1} \\ 2, & 3, \ldots, & n \end{bmatrix}$$

представља број исписаних пермутација.]
(5) $k \leftarrow n-1$ *[Увећавање броја са мешовитом основом за 1]*
(6) **if** $c_k = k$
(7) **do**
(8) $c_k \leftarrow 0, k \leftarrow k-1$
(9) **while not** $k = 0$ **or** $c_k < k$
(10) $c_k \leftarrow c_k + 1$
(11) **if** $k = 0$
(12) **exit** *[Нема наредне пермутације]*
(13) *PrimeniPermutaciju* $\tau(k,c_k)\omega(k+1)^-$ *на* $a_0a_1...a_{n-1}$ *[Пермутовање - генерисање нове пермутације применом множења пермутација]*
end

Пресликавања τ и ω се дефинишу као:

$$\tau(k,j) = \sigma(k,j)^-\sigma(k,j-1), \text{ за } 1 \leq j < k$$

$$\omega(k) = \sigma(n-1,n-1)^-\sigma(n-2,n-2)^- \cdots \sigma(k,k)^-,$$

где σ^- представља инверзно пресликавање. Као и код алгоритма Г, и овде постоји велики број варијација зависно од Симсове табеле која се користи. И овде је могуће користити Симсову табелу која код алгоритма Г производи пермутације у обрнутом колекс поретку:

$$\sigma(k,j) = (k-j \;\; k-j+1 \;\; ... \;\; k)$$

Алгоритам који се добија када се користи наведена Симсова табела је веома сличан алгоритму простих замена из 2.1.3.
Друга битна варијанта алгоритма је дуал Орд-Смит-овог метода (2.10), у којој се, као и у алгоритму Г, користи следећа Симсова табела:

$$\sigma(k,j) = (k \;\; ... \;\; 1 \;\; 0)^j \tag{2.12}$$

И овде су вредности пресликавања τ независне од j:

$$\tau(k,j) = (0\; 1\; ...\; k).$$

што омогућава да се избаци контролна табела $c_0c_1...c_{n-1}$. Разлог за то је што је $c_{n-1} = 0$ у кораку увећавања броја *c ако и само ако* је $a_{n-1} = n-1$, јер је тако дефинисано пресликавање α (2.11). Заиста, када је $c_j = 0$ за $k < j < n$ (линије 5 - 10), тада је $c_k = 0$ *ако и само ако* је $a_k = k$. На основу овог тврђења, ова варијанта алгоритма X се може преформулисати у веома једноставан алгоритам, којим се бави наредна тачка.

2.1.11 Алгоритам Ц - генерисање пермутација цикличним померајима

Алгоритам Ц је други дуални метод, и проистекао је из идеје алгоритма X, када се користи Орд-Смитова Симсова табела (2.12). Ипак, алгоритам генерише пермутације користећи циклична померања низа пермутација, и можда је и најједноставнији од свих агоритама за генерисање пермутација у смислу дужине програма. Творац је Г. Г. Лангдон Јр. (G.G. Langdon, Jr.) [9], а алгоритам је из 1967. године. Алгоритам је посебно користан за примене у којима је циклично померање (шифтовање) ефикасно, на пример када се пермутације чувају у машинском регистру а не низу.

Алгоритам 2.1.9. *Ц(Алгоритам за генерисање пермутација цикличним померајима)*
Улаз: *Скуп различитих елемената* $\{x_1, ..., x_n\}$
Излаз: *Све пермутације* $a_1...a_n$ *скупа различитих елемената* $\{x_1, ..., x_n\}$
begin

```
for 0 ≤ j ≤ n do
  a_j ← x_j [Иницијализација]
a ←true
while true do
  PosetiPermutaciju [Обилазак пермутације a_1...a_n]
  if a = true
    k ← n
  Zameniti a_1a_2...a_k sa a_2...a_ka_1
  if a_k! = x_k
    a ← true
    continue
  else
    k ← k − 1
    if k > 1
      a ← false
      continue
    else
      exit [Нема наредне пермутације]
```

end

Главна мана дуалних метода је што нису погодни за примене у којима се велики блокови пермутација прескачу, јер скуп свих пермутација које одговарају префиксу $c_0c_1...c_{k-1}$ обично није од значаја. Изузетак од овога је варијанта у којој се користи Симсова табела која генерише реверсни колекс поредак (2.7), јер у том случају важи да су $n!/k!$ пермутација када је $c_0c_1...c_{k-1} =$ 00...0 управо пермутације у којима 0 претходи 1, 1 претходи 2,..., и $k-2$ претходи $k-1$.

2.1.12 Алгоритам E - генерисање пермутација Ерлиховим заменама

Алгоритам који пермутације генерише користећи замене (транспозиције), слично алгоритму простих замена (алгоритам П) из 2.1.3 осмислио је Гидеон Елрих (Gideon Ehrlich). Овај алгоритам, као и неколико претходних алгоритама, као бројачки механизам користи контролну табелу $c_1...c_n$, а генерише пермутације заменом елемента a_0 претходне пермутације са одређеним другим елементом. Сличан је алгоритму П јер као и алгоритам П генерише пермутације заменом елемената, али за разлику од алгоритма П у коме се нова пермутација добија заменом места суседним елементима a_{t-1} и a_t, алгоритам Е користи такозване „звезда замене" *(енг. star transpositions)*, где елемент a_0 мења место са елементом a_t. Звезда замене су за нијансу ефикасније од замена суседних елемената јер елемент a_0 није потребно учитавати из меморије - познат је из претходног корака. Иако у пракси није бржи од осталих алгоритама за генерисање пермутација, овај алгоритам има својство да сваку пермутацију мења минимално како би добио следећу, користећи $n-1$ тип замене. Као и код алгоритма П, и овде је могуће унапред изгенерисати листу замена која се накнадно може користити. Просечан број замена елемената b_j и b_k у линији 17 је мањи од 0.18.

Алгоритам 2.1.10. *E (Алгоритам за генерисање пермутација Ерлиховим заменама)*
Улаз: *Низ различитих елемената* $a_0...a_{n-1}$
Излаз: *Све пермутације низа* $a_0...a_{n-1}$
Напомена: *Алгоритам користи помоћне табеле* $b_0...b_{n-1}$ *и контролну* $c_1...c_n$
begin

(1) **for** $0 \leq j < n$ **do** *[Иницијализација]*
(2) $b_j \leftarrow j$
(3) $c_{j+1} \leftarrow 0$
(4) **while true do**
(5) *Ispis [Испис пермутације* $a_0a_1...a_{n-1}$*]*
[На овом месту, број са мешовитом основом с представља број посећених пермутација.]
(6) $k \leftarrow 1$ *[Увећавање броја са мешовитом основом за 1]*
(7) **if** $c_k = k$
(8) **do**
(9) $c_k \leftarrow 0, k \leftarrow k+1$
(10) **while not** $c_k < k$
(11) **if** $k = n$
(12) **exit** *[Вредност броја са мешовитом основом је n!, тј. све пермутације су посећене]*
(13) $c_k \leftarrow c_k + 1$
(14) *Zameniti mesta elementima* a_0 *i* a_{b_k} *[Заменом места елементима се добија нова пермутација]*
(15) $j \leftarrow 1, k \leftarrow k-1$ *[Обртање подниза b]*
(16) **while** $j < k$
(17) *Zameniti mesta elementima* b_j *i* b_k
(18) $j \leftarrow j+1, k \leftarrow k-1$
end

2.1.13 Генерисање свих пермутација са две операције

Поставља се питање да ли је могуће генерисати све пермутације са две различите операције, уместо $n-1$. На пример, Нејенхуис (Nijenhuis) и Вилф (Wilf) су показали да је могуће изгенерисати све пермутације за $n=4$ користећи само две врсте замене: мењајући $a_1a_2...a_n$ са или $a_2a_3...a_na_1$ или $a_2a_1a_3...a_n$ [10]. У општем случају, за групу пермутација G са елементима $\alpha_1, ..., \alpha_k$, *Кејлијев* (Cayley) граф са генераторима $(\alpha_1, ..., \alpha_k)$ је усмерен граф чији су чворови пермутације π групе G и чије гране иду од π ка чворовима $\alpha_1\pi, ..., \alpha_k\pi$ [11]. Питање да ли је могуће генерисати све пермутације са само два генератора је еквивалентно питању да ли је у Кејлијевом графу свих пермутација скупа $\{1, 2, ..., n\}$ са генераторима σ и τ, где је σ циклична пермутација (пресликавање) $(1\ 2\ \cdots\ n)$ и τ транспозиција $(1\ 2)$ постоји **Хамилтонов пут**. Основна теорема Р. А. Ранкина (R. A. Rankin) [12] води до закључка да у многом случајевима то није могуће.

Теорема Р. Нека је Г група од g пермутација. Уколико Кејлијев граф са генераторима (α, β) има **Хамилтонов циклус**, и ако пермутације $(\alpha, \beta, \alpha\beta^-)$ имају редом редове (a, b, c) онда или је c паран или су g/a и g/b непарни.

Теорема говори о Хамилтоновом циклусу а не путу. *Ред* пермутације α је најмањи позитиван цео број a такав да је α^a идентичка пермутација.

За n=4, Хамилтонов пут је лако наћи.

За n>4 претрага постаје значајно обимнија, па су тако за n=5 Раски (Ruskey), Цианг (Jiang) и Вестон (Weston) нашли 5 различитих Хамилтонових путева опширном претрагом [13]. Они су такође нашли и Хамилтонов пут за n=6, али је пут веома сложен и нема логичку структуру. Кнут је такође нашао нешто простији Хамилтонов пут, али и даље веома сложен. Закључено је да приступ генерисања свих пермутација са само 2 генератора за веће вредности није практичан. Комптон (Compton) и Вилијамсон (Williamson) [14] су показали да Хамилтонов циклус постоји за све n уколико се дода још један генератор σ^-, тј. уколико се користе три генератора: σ, σ^- и τ.

2.1.14 Тополошко сортирање

Често у проблемима нису потребне све пермутације скупа $\{1, 2, ..., n\}$, већ само одређен број пермутација, које притом задовољавају одређен тип услова. На пример, потребно је наћи све пермутације скупа {1, 2, 3, 4} у којима 1 претходи 3, 2 претходи 3, 3 претходи 4. Решење је следећих пет пермутација:

1234, 1243, 2134, 2143, 2413

Ово је пример проблема *тополошког сортирања*. У општем случају, тополошко сортирање подразумева налажење свих пермутација које задовољавају m услова облика $x_1 \prec y_1, ..., x_m \prec y_m$, где $x \prec y$ значи да x претходи y у пермутацији. Проблем је познат још и под називом *проблем линеарног угнежђавања* (*енг. linear embedding problem*), јер се објекти ређају у линију тако да задовољавају одређен поредак у склопу неке релације. Проблем се може дефинисати и тако да се тражи само једна од свих $n!$ пермутација које задовољавају дате услове, или све пермутације које задовољавају дате услове. Ово се може посматрати и као граф - услови одређују гране, а бројеви $\{1, 2, \ldots, n\}$ чворове, и потребно је тополоши сортирати граф. Ова тачка се бави проналажењем свих пермутација које задовољавају дате услове. Стога се у овој тачки претпоставља да су елементи x и y на којима су релације дефинисане цели бројеви између 1 и n, и да је $x < y$ кад год је $x \prec y$. Када то није случај, елементи се могу преименовати пре уласка у алгоритам тако да задовољавају ове услове.
Проблем тополошког сортирања се може посматрати и као проблем генерисања свих пермутација са понављањем. На пример, пермутације скупа $\{1, 2, ..., 8\}$ такве да је:

$$1 \prec 2, \;\; 2 \prec 3, \;\; 3 \prec 4, \;\; 6 \prec 7, \;\; 7 \prec 8$$

еквивалентне су пермутацијама са понављањем скупа {1, 1, 1, 1, 2, 3, 3, 3}, јер се {1,2,3,4} може пресликати у 1, 5 у 2, а {6, 7, 8} у 3. За пермутације са понављањем може се искористити алгоритам Л из тачке 2.1.1, а још један алгоритам који се бави проблемом генерисања пермутација са понављањем, односно проблемом тополошког сортирања, је тема следеће тачке.

2.1.15 Алгоритам B - проналажење свих тополошких уређења

Елемент x претходи y у пермутацији $a_1...a_n$ ако и само ако $a_x\prime < a_y\prime$ у инверзној пермутацији $a_1\prime...a_n\prime$. Наредни алгоритам налази такође и све пермутације $a_1\prime...a_n\prime$ такве да је $a_j\prime < a_k\prime$ када је $j \prec k$. Пример овакве релације су *Јангови таблои* (*енг. Young tableau*). Јангов табло је уређење скупа $\{1, 2, ..., n\}$ у правоугаону табелу, тако да у сваком реду вредности расту са лева на десно, и у свакој колони вредности расту одозго на доле. Проблем генерисања свих 3 $\times$ 3 Јангових таблоа еквивалентан је налажењу свих $a_1\prime...a_9\prime$ таквих да:

$$\begin{aligned} &a_1\prime < a_2\prime < a_3\prime, \quad a_4\prime < a_5\prime < a_6\prime, \quad a_7\prime < a_8\prime < a_9\prime, \\ &a_1\prime < a_4\prime < a_7\prime, \quad a_2\prime < a_5\prime < a_8\prime, \quad a_3\prime < a_6\prime < a_9\prime, \end{aligned} \tag{2.13}$$

и ово је посебна врста тополошког сортирања.
Други пример проблема тополошког сортирања је проналажење свих начина да се од елемената $\{1, 2, ..., 2n\}$ направи n парова. За ово постоји тачно $(2n-1)(2n-2)\cdots(1) = (2n)!/2^n n!$ начина, односно то су пермутације које задовољавају следећи скуп услова:

$$a_1\prime < a_2\prime, \;\; a_3\prime < a_4\prime, \;\; ..., \;\; a_{2n-1}\prime < a_{2n}\prime, \qquad a_1\prime < a_3\prime < \cdots < a_{2n-1}\prime.$$

Алгоритам који спроводи тополошко сортирање осмисмили су И. Л. Верол (Y. L. Varol) и Д. Ротем (D. Rotem) [15]. Аналоган је методу простих замена (тачка 2.1.3). Идеја је да се претпостави да постоји начин за тополошко уређење скупа $\{1, ..., n-1\}$, тако да пермутације $a_1...a_{n-1}$ задовољавају све услове који не укључују n. Уколико је то тачно, n се додаје на постојећу шему без мењања релативног редоследа: прво се n дода на крај тако да се добије $a_1...a_{n-1}n$, а онда се n помера лево за једно место све док су задовољени услови за то. Псеудокод алгоритма је у наставку.

Алгоритам 2.1.11. *B (Алгоритам за проналажење свих тополошких уређења)*
Улаз: *Релација $\prec$ на скупу $\{1, ..., n\}$, са својством да $x \prec y$ имплицира $x < y$ (неједнакост).*
Излаз: *Све пермутације $a_1...a_n$ и њихови инверзи $a_1'...a_n'$ са својством да је $a_j' < a_k'$ кад год је $j \prec k$*
Напомена: *Због једноставности претпоставља се да је $a_0 = 0$ и $0 \prec k$ за $1 \leq k \leq n$.*
begin

(1) **for** $0 \leq j \leq n$ **do** *[Иницијализација]*
(2) $a_j \leftarrow j$
(3) $a_j' \leftarrow j$
(4) *PosetiPermutaciju a [Обилазак пермутације $a_1...a_n$]*
(5) *PosetiPermutaciju a' [Обилазак пермутације $a_1'...a_n'$]*
(6) $k \leftarrow n$
(7) **while true do**
(8) **while true do**
(9) $j \leftarrow a_k'$
(10) $l \leftarrow a_{j-1}$
(11) **if** $l \prec k$ *[Да ли k може да се помери лево?]*
(12) **break**
(13) **else** *[k може да се помери лево, умањивање k]*
(14) $a_{j-1} \leftarrow k$
(15) $a_j \leftarrow l$
(16) $a_k' \leftarrow j - 1$
(17) $a_l' \leftarrow j$
(18) *PosetiPermutaciju a [Обилазак пермутације $a_1...a_n$]*
(19) *PosetiPermutaciju a' [Обилазак пермутације $a_1'...a_n'$]*
(20) $k \leftarrow n$
(21) **while** $j < k$ *[k не може да се помери лево, враћање десно]*
(22) $l \leftarrow a_{j+1}$
(23) $a_j \leftarrow l$
(24) $a_l' \leftarrow j$
(25) $j \leftarrow j + 1$
(26) $a_k' \leftarrow k$
(27) $a_k \leftarrow a_k'$
(28) $k \leftarrow k - 1$
(29) **if** $k > 0$
(30) **continue** *[Покушај поново са умањеном вредношћу k]*
(31) **exit**
end

Уколико се инверзна пермутација $a_1'...a_9'$ напише у следећем облику:

$$\begin{array}{|c|}\hline a'_1 a'_2 a'_3 \\ a'_4 a'_5 a'_6 \\ a'_7 a'_8 a'_9 \\ \hline \end{array}$$

применом алгоритма B на проблем Јангових таблоа величине 3×3 (релације (2.13)), добијају се 42 резултата (слика 2.5).

123 456 789	123 457 689	123 458 679	123 467 589	123 468 579	124 356 789	124 357 689	124 358 679	124 367 589	124 368 579	125 367 489	125 368 479	125 346 789	125 347 689
125 348 679	126 347 589	126 348 579	127 348 569	126 357 489	126 358 479	127 358 469	134 256 789	134 257 689	134 258 679	134 267 589	134 268 579	135 267 489	135 268 479
145 267 389	145 268 379	135 246 789	135 247 689	135 248 679	136 247 589	136 248 579	137 248 569	136 257 489	136 258 479	137 258 469	146 257 389	146 258 379	147 258 369

Слика 2.5: Јангови таблои величине 3×3

Нека је t_r број тополошких уређења за која важи да је $n-r$ елемената у својој почетној позицији $a_j = j$ за $r < j \leq n$. Еквивалентно томе, t_r је број тополошких уређења $a_1 ... a_r$ скупа $\{1, ..., r\}$ када се игноришу релације које укључују елементе веће од r. Тада се у алгоритму B корак обиласка пермутације a и њеног инверза извршава N пута, а линије 9-11 извршавају M пута, где су:

$$M = t_n + \cdots + t_1, \text{ и } N = t_n.$$

Линије 14-17 и петља из линије 21 се извршавају $N - 1$ пута, а линије 26-30 $M - N + 1$ пута. Одавде следи да је укупно време извршавања алгоритма линеарна комбинација M, N и n. Међутим, може се десити да је почетно преименовање извршено тако да је M много веће од N. На пример, следећи проблем тополошког сортирања се може решити на два начина. Уколико су услови алгоритма следећи:

$$2 \prec 3, \;\; 3 \prec 4, \;\; ..., \;\; n-1 \prec n,$$

тада је $t_j = j$ за $1 \leq j \leq n$, и $M = (n^2 + n)/2, N = n$, што води до квадратне сложености алгоритма. Међутим, уколико се изврши преименовање елемената, исти услови се могу написати овако:

$$1 \prec 2, \;\; 2 \prec 3, \;\; 3 \prec 4, \;\; ..., \;\; n-2 \prec n-1,$$

M је тада $2N - 1 = 2n - 1$.
Ипак, увек је могуће урадити почетно преименовање елемената тако да се, уз благу модификацију алгоритма B, нађу сва тополошка уређења у времену $O(N + n)$.

2.2 Генерисање свих комбинација

2.2.1 Увод

Многи аутори се слажу да су у комбинаторној математици следећа најбитнија ствар после пермутација комбинације. Ово поглавље представља пет најважнијих алгоритама за генерисање комбинација.
Комбинације се могу представити на више начина, а осим тога су и еквивалентне многим другим конфигурацијама, стога се прва тачка овог поглавља бави разним начинима представљања комбинација. Након тога следи, као и у случају пермутација, основни и најједноставнији начин за генерисање комбинација - алгоритам који комбинације генерише у лексикографском поретку (алгоритам Л). Унапређена и самим тим нешто сложенија верзија овог алгоритма чини наредни алгоритам - Т, који је тема тачке 2.2.4. Да се процес генерисања свих комбинација може посматрати као процес обилажења специјалне врсте стабла - *биномног* стабла, показано је у тачки 2.2.5. Биномна стабла се могу употребити и за проблем пуњења ранца, што је, заједно са алгоритмом за пуњење ранца Ф, такође показано у овој тачки. Наредна тачка 2.2.6 показује инкременталну методу генерисања комбинација - где је основна идеја да се наредна пермутација генерише минимално мењајући претходну. Алгоритам за ово је алгоритам Р. Последња тачка 2.2.7 разматра ову идеју дубље, дефинише *хомогене* и *скоро савршене* секвенце комбинација, и даје алгоритам за њихово генерисање.

2.2.2 Начини представљања комбинација

t-комбинација од n елемената је подскуп избора величине t из скупа величине n. Постоји $\binom{n}{t}$ начина да се изабере такав подскуп. Бирање t од n елемената еквивалентно је избору $s = n - t$ од n елемената. Зато се у наставку t-комбинације зову још и (s,t)-комбинације, где је

$$n = s + t$$

Постоји више начина за представљање (s,t)-комбинација. Два најбитнија су да се наброје само индекси t изабраних елемената: $c_t \dots c_1$, а други је бинарни стринг $a_{n-1} \dots a_1 a_0$ са t јединица и s нула, у коме 1 означава да је елемент изабран а 0 да није, па важи:

$$a_{n-1} + \cdots + a_1 + a_0 = t$$

Прва репрезентација је погодна када су елементи чланови скупа $\{0, 1, 2, \dots, n-1\}$ и када се поређају у опадајућем поретку, тако да је:

$$n > c_t > \cdots > c_2 > c_1 \geq 0 \tag{2.14}$$

Ове две нотације повезане су на следећи начин:

$$2^{c_t} + \cdots + 2^{c_2} + 2^{c_1} = \sum_{k=0}^{n-1} a_k 2_k = (a_{n-1} \dots a_0)_2 \tag{2.15}$$

Осим ове две репрезентације комбинација, постоје и друге. Тако је могуће набројати позиције s нула $b_s \dots b_2 b_1$ из бинарног стринга $a_{n-1} \dots a_1 a_0$, где је:

$$n > b_s > \cdots > b_2 > b_1 \geq 0.$$

Комбинације су битне и због тога јер су еквивалентне многим другим конфигурацијама. На пример, свака (s,t)-комбинација одговара комбинацији са понављањем где се од $s+1$ бира t елемената. Ова комбинација означава се са $d_t \dots d_2 d_1$, и важи:

$$s \geq d_t \geq \cdots \geq d_2 \geq d_1 \geq 0. \tag{2.16}$$

Важи да $d_t \dots d_2 d_1$ задовољава (2.16) ако и само ако $c_t \dots c_2 c_1$ задовољава (2.14), где је:

$$c_t = d_t + t - 1, \quad \dots, \quad c_2 = d_2 + 1, \quad c_1 = d_1.$$

Постоји још једна веза између комбинација са понављањем и обичних комбинација. Наиме, могу да се дефинишу бројеви:

$$e_j = \begin{array}{l} c_j, \text{ако је } c_j \leq s \\ e_{c_j - s}, \text{ако је } c_j > s \end{array}$$

За ову форму важи да је надскуп $\{e_1, e_2, \dots, e_t\}$ једнак $\{c_1, c_2, \dots, c_t\}$ ако и само ако је $\{e_1, e_2, \dots, e_t\}$ скуп.
Осим овога, свака (s,t)-комбинација одговара разлагању $n+1$ објеката у t делова, односно може се представити као уређена сума:

$$n + 1 = p_t + \cdots + p_1 + p_0, \quad p_t, \dots, p_1, p_0 \geq 1. \tag{2.17}$$

Ово се повезује са (2.14) преко:

$$p_t = n - c_t, \; p_{t-1} = c_t - c_{t-1}, \; \dots, \; p_1 = c_2 - c_1, \; p_0 = c_1 + 1.$$

Еквивалентно, ако се уведе смена: $q_j = p_j - 1$, важи:

$$s = q_t + \ldots + q_1 + q_0, \qquad q_t, \ldots, q_1, q_0 \geq 0, \tag{2.18}$$

што представља разлагање s елемената у $t+1$ непразних скупова, која се повезује са (2.16) преко:

$$q_t = s - d_t,\ q_{t-1} = d_t - d_{t-1},\ \ldots,\ q_1 = d_2 - d_1,\ q_0 = d_1.$$

Свакој (s,t)-комбинацији одговара и пут дужине $s+t$ кроз мрежу димензија $s \times t$, који садржи s вертикалних и t хоризонталних корака.
Свих осам начина на који се комбинације могу посматрати приказано је на слици 2.6, за $s = t = 3$.

$a_5a_4a_3a_2a_1a_0$	$b_3b_2b_1$	$c_3c_2c_1$	$d_3d_2d_1$	$e_3e_2e_1$	$p_3p_2p_1p_0$	$q_3q_2q_1q_0$	path
000111	543	210	000	210	4111	3000	
001011	542	310	100	310	3211	2100	
001101	541	320	110	320	3121	2010	
001110	540	321	111	321	3112	2001	
010011	532	410	200	010	2311	1200	
010101	531	420	210	020	2221	1110	
010110	530	421	211	121	2212	1101	
011001	521	430	220	030	2131	1020	
011010	520	431	221	131	2122	1011	
011100	510	432	222	232	2113	1002	
100011	432	510	300	110	1411	0300	
100101	431	520	310	220	1321	0210	
100110	430	521	311	221	1312	0201	
101001	421	530	320	330	1231	0120	
101010	420	531	321	331	1222	0111	
101100	410	532	322	332	1213	0102	
110001	321	540	330	000	1141	0030	
110010	320	541	331	111	1132	0021	
110100	310	542	332	222	1123	0012	
111000	210	543	333	333	1114	0003	

Слика 2.6: (3,3)-комбинације и њихови еквиваленти

2.2.3 Лексикографско генерисање комбинација и алгоритам Л

Као и у случају пермутација, и овде је најједноставнији и најинтуитивнији алгоритам за генерисање алгоритам који комбинације генерише у лексикографском поретку. Како је свака (s,t)-комбинација једнака пермутацији са понављањем скупа $\{s\cdot 0, t\cdot 1\}$, за генерисање комбинација у основној форми $a_{n-1}\dots a_1a_0$ може се искористити лексикографски алгоритам за генерисање пермутација (и пермутација са понављањем) из 2.1.1. Међутим, обично је погоднија форма за представљање комбинација друга основна форма - $c_t\dots c_2c_1$, пре свега због компактности када је t мало у односу на n. Најједноставнији алгоритам за генерисање комбинација у овој форми може се применити уколико је t веома мали број, а то је искористити t угнеждених петљи. На пример, за $t=3$, основни алгоритам изгледа овако:

begin
  За $c_3 = 2,3,\dots,n-1$ уради:
    За $c_2 = 1,2,\dots,c_3-1$ уради:
      За $c_1 = 0,1,\dots,c_2-1$ уради:
        Посети комбинацију $c_3c_2c_1$.
end

Међутим, када је t променљив, или није веома мали, основни алгоритам за генерисање пермутација у лексикографском поретку се добија према рецепту за генерисање било које комбинаторсне схеме а који је споменут и у 2.1.1: наћи најдеснији елемент c_j који може да се увећа, а потом поставити елементе $c_{j-1}\dots c_1$ на најмање могуће вредности. Алгоритам који следи добијен је на тај начин.

Алгоритам 2.2.1. *Л (Лексикографске комбинације) - Алгоритам генерише све t-комбинације $c_t\dots c_2c_1$ n бројева $\{0,1,\dots,n-1\}$, за $n\geq t\geq 0$ у лексикографском поретку.*
Улаз: *бројеви n и t, $n\geq t\geq 0$*
Излаз: *све t-комбинације $c_t\dots c_2c_1$ n бројева $\{0,1,\dots,n-1\}$.*
Напомена: *Додатне променљиве c_{t+1} и c_{t+2} се користе као границе приликом генерисања.*
begin

(1) **for** $1\leq j\leq t$
(2) $c_j \leftarrow j-1$
(3) $c_{t+1}\leftarrow n, c_{t+2}\leftarrow 0$
(4) **while true do**
(5) *Ispisi kombinaciju $c_t\dots c_2c_1$ [Испис комбинације]*
(6) $j\leftarrow 1$ *[Налажење j]*
(7) **while** $c_j+1=c_{j+1}$ **do**
(8) $c_j=j-1, j=j+1$
(9) **if** $j>t$ **then**
(10) **exit** *[Нема наредне комбинације]*
(11) $c_j\leftarrow c_j+1$ *[Увећавање j]*

end

Најсложенији корак овог алгоритма - налажење вредности j (линије 6 - 8) поставља c_j на $j-1$ одмах након посете комбинацији за коју важи $c_{j+1}=c_1+j$, и број оваквих комбинација једнак је броју решења неједнакости

$$n > c_t > \dots > c_{j+1} \geq j$$

Вредност овог израза једнака је броју $(t-j)$-комбинација од $n-j$ објеката $\{n-1,\ldots,j\}$, па се додела $c_j \leftarrow j-1$ дешава тачно $\binom{n-j}{t-j}$ пута. Сумирајући за $1 \le j \le t$ добија се да се петља из линија 7-8 извршава тачно

$$\binom{n-1}{t-1}+\binom{n-2}{t-2}+\cdots+\binom{n-t}{0}=\binom{n-1}{s}+\binom{n-2}{s}+\cdots+\binom{s}{s}=\binom{n}{s+1}$$

пута укупно, односно у просеку:

$$\binom{n}{s+1}/\binom{n}{t}=t/(s+1)$$

пута по посети комбинације. Када је $t \le s$ овај број је мањи од 1, па је алгоритам ефикасан. Међутим, када је t близу n, ова вредност је веома велика па алгоритам постаје неефикасан. Ипак, алгоритам се може унапредити. Дешава се да алгоритам понекад ради доделу $c_j \leftarrow j-1$ беспотребно, када је c_j већ једнако $j-1$. Није увек потребно тражити следећу вредност j кроз петљу јер се она може закључити из претходних акција. На пример, након што је c_4 увећан и $c_3c_2c_1$ се поставе на почетне вредности 2 1 0, акција која следи је повећавање c_3. Ова анализа води до унапређене верзије алгоритма Л која следи.

2.2.4 Алгоритам Т

Алгоритам 2.2.2. *Т (Унапређене лексикографске комбинације) - Овај алгоритам генерише све t-комбинације $c_t \ldots c_2c_1$ n бројева $\{0,1,\ldots,n-1\}$, за $n \ge t \ge 0$.*
Улаз: *бројеви n и t, $n \ge t \ge 0$*
Излаз: *све t-комбинације $c_t \ldots c_2c_1$ n бројева $\{0,1,\ldots,n-1\}$.*
Напомена: *Због једноставности се претпоставља да је $t < n$*
begin

```
  for 1 ≤ j ≤ t [Иницијализација]
    c_j ← j − 1
  c_{t+1} ← n, c_{t+2} ← 0, j ← t
  while true do
    Ispisi kombinaciju c_t ... c_2c_1 [Испис комбинације]
    while j > 0 do
      x ← j, c_j ← x, j ← j − 1
      Ispisi kombinaciju c_t ... c_2c_1
    while c_1 + 1 < c_2 do
      c_1 = c_1 + 1
      Ispisi kombinaciju c_t ... c_2c_1
      while j > 0 do
        x ← j, c_j ← x, j ← j − 1
        Ispisi kombinaciju c_t ... c_2c_1
    j ← 2
    c_{j−1} ← j − 2, x ← c_j + 1 [Налажење j]
    while x = c_{j+1}
      j ← j + 1, c_{j−1} ← j − 2, x ← c_j + 1
    if j > t then
      exit [Нема наредне комбинације]
    c_j ← x, j ← j − 1 [Повећавање j]
end
```

Алгоритми Л и Т користе параметар n само у почетном кораку иницијализације, стога се они могу посматрати као алгоритми за генерисање бесконачне листе комбинација која зависи само од t. Разлог за ово је што се користи лексикографски поредак на опадајућој секвенци $c_t \dots c_2 c_1$, што омогућава да листа првих $n+1$ комбинација почиње листом првих n комбинација. Алгоритми Л и Т имају још једно корисно својство, изражено у следећој теореми.

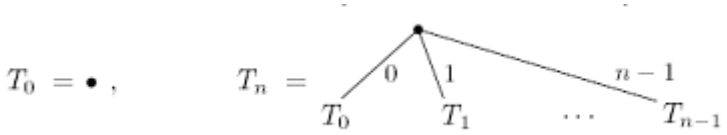
Теорема 2.2.3. *Комбинација $c_t \dots c_2 c_1$ је посећена након што је посећено тачно:*

$$\binom{c_t}{t} + \dots + \binom{c_2}{2} + \binom{c_1}{1}$$

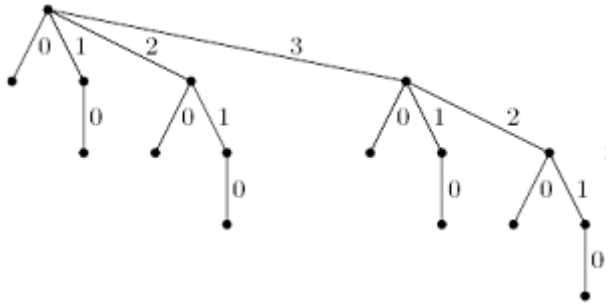
других комбинација.

2.2.5 Биномна стабла и алгоритам Ф

Фамилија стабала, дефинисана са:



за $n > 0$, пружа додатан поглед на проблем генерисања пермутација. Стабло T_n је сачињено од стабла T_{n-1} и још једне копије стабла T_{n-1}. Стога стабло T_n има 2^n чворова. Број чворова стабла на нивоу t је биномни коефицијент $\binom{n}{t}$, отуда име *биномна стабла*. Низ ознака од корена до произвољног чвора на нивоу t представља комбинацију $c_t \ldots c_1$, а све комбинације се појављују у лексикографском поретку, са лева на десно. Стога се алгоритми Л и Т из претходне тачке могу посматрати и као начини обиласка чворова на нивоу t биномног стабла T_n.



Биномно стабло T_4

Једна од примена биномних стабала је **проблем паковања ранца**. Од датих n елемената, са тежинама $w_{n-1}, \ldots, w_1, w_0$, за које важи:

$$w_{n-1} \geq \cdots \geq w_1 \geq w_0$$

потребно је генерисати све бинарне векторе $a_{n-1} \ldots a_1 a_0$ такве да:

$$a \cdot w = a_{n-1} w_{n-1} + \cdots + a_1 w_1 + a_0 w_0 \leq N$$

где је N капацитет ранца. Овај захтев еквивалентан је захтеву за генерисање свих подскупова C скупа $\{0, 1, \ldots, n-1\}$ таквих да је $w(C) = \sum_{c \in C} w_c \leq N$, такозваних *остваривих* подскупова. Остварив подскуп се означава са $c_1 \ldots c_t$, где је $c_1 > \cdots > c_t \geq 0$, обрнуто конвенцији из претходне тачке. Сваки остварив подскуп одговара *достижном* чвору у стаблу T_n, и циљ је посетити сваки достижан чвор. Алгоритам који следи имплицитно обилази чворове стабла T_n редоследом *леви-корен-десни*. Отац и леви брат сваког достижног чвора су достижни чворови, а алгоритам прескаче подстабла која нису достижна. Елемент $c > 0$ се ставља у ранац ако може да стане, након што су посећени сви елементи који уместо њега укључују елемент $c-1$. Време рада алгоритма је пропорционално броју комбинација које се обилазе. Алгоритам није погодан за решавање класичног проблема пуњења ранца, јер разматра све могуће начине за пуњење ранца за задати капацитет, док класични проблем тражи само једну комбинацију елемената која максимизује попуњеност ранца.

Алгоритам 2.2.4. *Ф (Пуњење ранца). Алгоритам генерише све начине $c_1 \dots c_t$ за пуњење ранца, за дате тежине $w_{n-1} \dots w_1 w_0$ и капацитет N.*
Улаз: *Низ тежина $w_{n-1} \dots w_0$, капацитет ранца N*
Излаз: *Сви могући начини за пуњење ранца, представљени низом индекса елементас$_1 \dots c_t$*
Напомена: *Алгоритам користи додатни низ разлика у тежинама δ_j, $\delta_j = w_j - w_{j-1}$, за $1 \le j < n$.*
begin

```
  for 1 ≤ j < n [Иницијализација]
    δ_j ← w_j − w_{j−1}
  t ← 0, c_0 ← n, r ← N [Иницијализација]
  Ispis c_1c_2 ... c_t [Комбинација користи N − r јединица капацитета]
  while c_t > 0 and r ≥ w_0 do [Покушај додавања нултог елемента]
    t ← t + 1, c_t ← 0, r ← r − w_0
    Ispis c_1c_2 ... c_t
  while true do
    if t = 0
      exit [Нема наредне комбинације]
    while c_{t−1} > c_t + 1 and r ≥ δ_{c_t+1} then [Покушај увећања c_t]
      c_t ← c_t + 1, r ← r − δ_{c_t} [Додавање следећег елемента на позицију t
                и умањивање капацитета за вредност његове тежине ]
      Ispis c_1c_2 ... c_t
      while c_t > 0 and r ≥ w_0 do [Покушај додавања нултог елемента]
        t ← t + 1, c_t ← 0, r ← r − w_0
        Ispis c_1c_2 ... c_t
      if t = 0
        exit [Нема наредне комбинације]
    r = r + w_{c_t}, t = t − 1 [Уклањање елемента c_t]
end
```

2.2.6 Грејов код у комбинацијама и алгоритам P

Често се од алгоритма за генерисање комбинација захтева да генерише комбинације тако да се две узастопне комбинације разликују минимално. Пример овога је *алгоритам обртних врата*: између две собе се налазе обртна врата. У првој соби се налази s људи а у другој t. Кад год неко изађе из једне собе и пређе у другу, особа из друге собе изађе (и пређе у прву). Да ли је могућ низ прелазака такав да се свака (s,t)- комбинација јави само једном? Испоставља се да постоји више таквих низова, а један од њих, који је овде изложен, користи Грејов бинарни код. *Грејов бинарни код* је уређење бинарних стрингова које листа свих 2^n бинарних стрингова дужине n тако да се свака два суседна стринга разликују у само једном биту. Најједноставнији Грејов код, тзв. бинарни рефлектовани Грејов код, може се индуктивно дефинисати на следећи начин [1]:

$$\begin{aligned} \Gamma_0 &= \epsilon; \\ \Gamma_{n+1} &= 0\Gamma_n, 1\Gamma_n^R. \end{aligned} \tag{2.19}$$

ϵ означава празан стринг, $0\Gamma_n$ означава бинарни стринг Γ_n са префиксом 0, а $1\Gamma_n^R$ означава секвенцу Γ_n у реверсном поретку са префиксом 1 испред сваког стринга.

Уколико се посматрају сви n-битни стрингови $a_{n-1}\dots a_1a_0$ у поретку Грејовог бинарног кода, и одаберу само они који имају s нула и t јединица, резултујући стрингови формирају *код обртних врата*, или *Грејов код обртних врата*.
На пример, за $s=t=3$ комбинације у *Грејовом поретку обртних врата* Γ_{33} изгледају овако:

000111	011010	110001	101010
001101	011100	110010	101100
001110	010101	110100	100101
001011	010110	111000	100110
011001	010011	101001	100011

(2.20)

Γ_{23} се налази у прве две колоне овог низа. Први елемент се добија од последњег једним окретом врата. Када се (2.20) напише у индексној нотацији $c_3c_2c_1$, особина овог низа комбинација види се још боље:

210	431	540	531
320	432	541	532
321	420	542	520
310	421	543	521
430	410	530	510

(2.21)

Прва компонента c_3 се јавља у растућем поретку. За сваку фискирану вредност c_3, вредности c_2 се јављају у опадајућем поретку. За фиксне вредности c_3c_2 вредности c_1 се јављају у растућем поретку. У општем случају, важи да се у Грејовом поретку обртних врата све комбинације $c_t\dots c_2c_1$ појављују у лексикографском поретку алтернирајуће секвенце:

$$(c_t, -c_{t-1}, c_{t-2}, \dots, (-1)^{t-1}c_1) \tag{2.22}$$

Следећи алгоритам генерише комбинације у наведеном лексикографском поретку (2.22).

Алгоритам 2.2.5. *R (Алгоритам обртних врата) - овај алгоритам генерише све t-комбинације* $c_t \dots c_2 c_1$ *скупа* $\{0, 1, \dots, n-1\}$ *у лексикографском поретку алтернирајуће секвенце, за* $n > t > 1$. *Алгоритам се одмах после иницијализације грана, у зависности од тога да ли је* t *паран или непаран.*

Улаз: *бројеви* n *и* t, $n > t > 1$

Излаз: *све t-комбинације* $c_t \dots c_2 c_1$ *скупа* $\{0, 1, \dots, n-1\}$.

begin

for $1 \leq j \leq t$ *[Иницијализација]*
$c_j \leftarrow j - 1$
$c_{t+1} \leftarrow n$
if t *je neparan* **then** *[Посебан ток уколико је t непаран]*
while true do
Ispisi kombinaciju $c_t \dots c_2 c_1$ *[Обилазак комбинације]*
[Када c_1 *и* c_2 *нису узастопни елементи, увећавањем* c_1 *се добија нова комбинација]*
while $c_1 + 1 < c_2$ **do**
$c_1 \leftarrow c_1 + 1$
Ispisi kombinaciju $c_t \dots c_2 c_1$
$j \leftarrow 2$
while true do
[Покушај смањивања c_j *- у овом тренутку је* $c_j = c_{j-1} + 1$*]*
if $c_j \geq j$ **then**
$c_j \leftarrow c_{j-1}, c_{j-1} \leftarrow j - 2$
break
else
$j \leftarrow j + 1$
[Покушај увећавања c_j*]*
if $c_j + 1 < c_{j+1}$ **then**
$c_{j-1} \leftarrow c_j, c_j \leftarrow c_j + 1$
break
else
$j \leftarrow j + 1$
if $j \leq t$ **then**
continue *[још нису посећене све комбинације, понавља се процес са увећаним j]*
else
exit *[Нема наредне комбинације, све комбинације су посећене]*
else
[наставак на следећој страни због прегледности]

while true do *[Грана за генерисање комбинација уколико је t паран]*
Ispisi kombinaciju $c_t \dots c_2 c_1$
while $c_1 > 0$ **do**
$c_1 \leftarrow c_1 - 1$
Ispisi kombinaciju $c_t \dots c_2 c_1$
$j \leftarrow 2$
while true do
if $c_j + 1 < c_{j+1}$ **then**
$c_{j-1} \leftarrow c_j, c_j \leftarrow c_j + 1$
break
else
$j \leftarrow j + 1$
if $j \leq t$ **then**
if $c_j \geq j$
$c_j \leftarrow c_{j-1}, c_{j-1} \leftarrow j - 2$
break
else
$j \leftarrow j + 1$
else
exit *[Нема наредне комбинације]*
end

Алгоритам P има занимљиво својство, аналогно теореми 2.2.3 која важи за алгоритме Л и T из 2.2.3 и 2.2.4:
Комбинација $c_t c_{t-1} ... c_2 c_1$ посећује се након тачно

$$N = \binom{c_t + 1}{t} - \binom{c_{t-1} + 1}{t - 1} + \cdots + (-1)^t \binom{c_1 + 1}{1} - t \ mod\ 2$$

других посећених пермутација. Оваква представа броја N се може посматрати и као репрезентација N у „алтернирајућем комбинаторном бројевном систему". Једна од последица овога је и да сваки позитиван цео број N има јединствену репрезентацију облика $N = \binom{a}{3} - \binom{b}{2} + \binom{c}{1}$, за $a > b > c > 0$. Алгоритам P показује како се у том бројевном систему додаје 1.

Алгоритам P комбинације обилази у такозваном *genlex* поретку. Низ стрингова је у *genlex* поретку уколико се сви стрингови са заједничким префиксом појављују узастопно. На пример, у (2.21), све комбинације које почињу са 53 се појављују једна до друге.
Генлекс поредак омогућава да се стрингови представе помоћу стабла, са произвољним редоследом деце сваког чвора. Када се стабло обилази тако да се сваки чвор обилази одмах пре, или одмах после својих потомака, сви чворови са истим префиксом се појављују узастопно. Ово је погодно за рекурзивне алгоритме генерисања.

Две узастопне комбинације добијене алгоритмом P разликују у само једном елементу, али се често дешава да се промене два индекса c_j, да би се сачувао услов $c_t > \cdots > c_1$. На пример, алгоритам P генерише комбинацију 210, а потом комбинацију 320, а приликом генерисања

(2.21) ово мењање два индекса се дешава 9 пута. Алгоритам из следеће тачке показује како да се ово унапреди.

2.2.7 Хомогене, скоро савршене секвенце комбинација и алгоритам Ц

Грејова секвенца комбинација се назива *хомогена* уколико мења само један индекс c_j у једном кораку. Карактеристична је по томе што се при генерисању нове комбинације користе само транзиције облика $10^a \leftrightarrow 0^a1$, за $a \geq 1$. Као илустрација хомогене шеме може се искористити свирање свих комбинација (акорда) од t-дирки на клавијатури од n дирки, мењајући само један прст за свирање наредне комбинације (акорда).

Уз благу модификацију дефиниције Грејовог бинарног кода за комбинације (2.19), могуће је добити хомогену секвенцу (s,t)-комбинација. Она се може описати следећом индуктивном конструкцијом:

$$K_{s0} = 0^s, K_{0t} = 1^t, K_{s(-1)} = \emptyset$$
$$K_{st} = 0K_{(s-1)t}, 10K^R_{(s-1)(t-1)}, 11K_{s(t-2)}, \quad st > 0$$

На месту где се налази први зарез у дефиницији дешава се прелаз из $01^t0^{(s-1)}$ у $101^{(t-1)}0^{(s-1)}$, а код другог $10^s1^{(t-1)}$ у $110^s1^{(t-2)}$; обе транзиције су хомогене, иако у другој јединица „прескаче" s нула. На пример, за $s = t = 3$, K_{33} изгледа овако у облику низа битова:

000111	010101	101100	100011
001011	010011	101001	110001
001101	011001	101010	110010
001110	011010	100110	110100
010110	011100	100101	111000

а овако када се напише у индексној нотацији (као „прсторед" на клавијатури):

210	420	532	510
310	410	530	540
320	430	531	541
321	431	521	542
421	432	520	543

Уколико се хомогена шема за комбинације $c_t \ldots c_1$ преведе у одговарајућу шему за комбинације са понављањем $d_t \ldots d_1$, задржава својство да се само један индекс d_j мења у једном кораку. Уколико се преведе у шеме (2.17) и (2.18), само два суседна дела се мењају када се мења c_j. Међутим, испоставља се да је могуће генерисати све (s,t)-комбинације не само користећи хомогене транзиције већ и „јако" хомогене транзиције - транзиције које су или $01 \leftrightarrow 10$ или $001 \leftrightarrow 100$. Ово значи да се индекс c_j мења највише за 2. Овакве шеме генерисања се називају још и „скоро савршене" [16]. Следећа теорема ближе одређује „скоро савршене" алгоритме:

Теорема 2.2.6. *Уколико је $st > 0$, постоји тачно $2s$ начина за генерисање свих (s,t)-комбинација у genlex поретку. За $1 \leq a \leq s$, постоји тачно један начин, N_{sta}, који почиње са 1^t0^s и завршава се са $0^a1^t0^{s-a}$, и s реверсних листи N^R_{sta}.*

N_{sta} се дефинише следећим рекурзивним формулама, које важе за $st{>}0$:

$$N_{s0a} = 0^s,$$
$$N_{sta} = \begin{array}{ll} 1N_{s(t-1)1}, 0N_{(s-1)t(a-1)}, & 1 < a < s; \\ 1N_{s(t-1)2}, 0N^R_{(s-1)t1}, & 1 = a < s; \\ 1N_{1(t-1)1}, 01^t, & 1 = a = s. \end{array}$$

Уколико се уведу $A_{st} = N_{st1}$ и $B_{st} = N_{st2}$, добијају се скоро савршене шеме које је открио Филип Чејз (Phillip J. Chase) 1976. године, занимљиве јер имају ефекат шифтовања левог блока јединица на десно за једну (A_{st}) и две позиције (B_{st}). А и В задовољавају и узајамне рекурзије:

$$A_{st} = 1B_{s(t-1)}, 0A^R_{(s-1)t}\,; \qquad B_{st} = 1A_{s(t-1)}, 0A(s-1)t.$$

Сликовито: „За један корак напред, иди два корака унапред а онда један уназад; за два корака унапред, иди један корак унапред, а онда још један". Ове једначине важе за све целе бројеве s и t, и важи да су A_{st} и B_{st} једнаки $\emptyset$ када су s или t негативни, као и $A_{00} = B_{00} = \epsilon$ (празан стринг). Тако A_{st} заправо води унапред $\min(s, 1)$ корака, а В $\min(s, 2)$ корака. Табела 1 приказује пример Чејзових секвенци комбинација за $s = t = 3$.

$A_{33} = C^R_{33}$			
543	531	321	420
541	530	320	421
540	510	310	431
542	520	210	430
532	521	410	432

$B_{33} = C_{33}$			
543	520	432	410
542	510	430	210
540	530	431	310
541	531	421	320
521	532	420	321

Табела 1: Чејзове секвенце за (3,3)-комбинације, у форми листе индекса $c_3c_2c_1$

Када се уведу следеће смене, имплементација у рачунару постаје једноставнија:

$$C_{st} = \begin{array}{ll} A_{st}, \text{ ако} & \text{је } s+t \text{ непаран;} \\ B_{st}, \text{ ако} & \text{је } s+t \text{ паран;} \end{array} \quad C_{st} = \begin{array}{ll} A^R_{st}, \text{ ако} & \text{је } s+t \text{ паран;} \\ B^R_{st}, \text{ ако} & \text{је } s+t \text{ непаран;} \end{array}$$

Тада се добија:

$$C_{st} = \begin{array}{ll} 1C_{s(t-1)}, \ 0C_{(s-1)t}, \text{ ако} & \text{је } s+t \text{ непаран;} \\ 1C_{s(t-1)}, \ 0C_{(s-1)t}, \text{ ако} & \text{је } s+t \text{ паран;} \end{array} \tag{2.23}$$

$$C_{st} = \begin{array}{ll} 0C_{(s-1)t}, \ 1C_{s(t-1)}, \text{ ако} & \text{је } s+t \text{ паран;} \\ 0C_{(s-1)t}, \ 1C_{s(t-1)}, \text{ ако} & \text{је } s+t \text{ непаран;} \end{array} \tag{2.24}$$

Алгоритам за генерисање секвенце C_{st} је заснован на општим идејама за генерисање било које *genlex* шеме. Бит a_j је активан уколико је потребно да се промени пре било ког другог бита са његове леве стране. Уколико се одржава помоћна табела $w_n \ldots w_1 w_0$, где је $w_j = 1$ ако и само ако важи да или је a_j активан бит, или је $j < r$, где је r најмањи индекс такав да је $a_r \neq a_0$. Такође важи да је $w_n = 1$. Тада следећи алгоритам налази стринг након $a_n \ldots a_1 a_0$ у поретку Чејзове секвенце:

begin
Поставити $j \leftarrow r$
Уколико је $w_j = 0$:
Постави $w_j \leftarrow 1, j \leftarrow j+1$, понављај док не буде $w_j = 1$
Уколико је $j = n$ заврши алгоритам, иначе $w_j \leftarrow 0$
Постави $a_j \leftarrow 1 - a_j$ и уради неопходне промене на $a_{j-1} \ldots a_0$ и r тако да .. буде валидан у оквиру генлекс шеме која се користи.
end

Доказано је да је ова петља ефикасна: операција $j \leftarrow j+1$ се извршава мање од једном у просеку при генерисању једне комбинације у секвенци. Уколико се користе транзиције (2.23) и (2.24), добија се комплетан алгоритам:

Алгоритам 2.2.7. *Ц (Чејзове секвенце) - Алгоритам генерише све (s,t)-комбинације $a_{n-1} \ldots a_1 a_0$, за $n = s+t$, у „скоро савршеном" поретку Чејзове секвенце комбинација C_{st}.*
Улаз: *бројеви n и t, $n \geq t > 0$*
Излаз: *(s,t)-комбинације $a_{n-1} \ldots a_1 a_0$, n=s+t*
begin

for $0 \leq j < s$
  $a_j \leftarrow 0$
for $s \leq j < n$
  $a_j \leftarrow 1$
for $0 \leq j \leq n$
  $w_j \leftarrow 1$
if $s > 0$ **then**
  $r \leftarrow s$
else
  $r \leftarrow t$
while true do
  Ispisi kombinaciju $a_{n-1} \ldots a_1 a_0$
  $j \leftarrow r$
  if $w_j = 0$ **then**
    while $w_j \neq 1$
      $w_j \leftarrow 1, j \leftarrow j+1$
  if $j = n$
    exit *[Нема наредне комбинације]*
  $w_j \leftarrow 0$
  [Наставак на следећој страни, због прегледности

```
    if neparan(j) and a_j ≠ 0 then
      a_{j-1} ← 1, a_j ← 0
      if r = j and j > 1
        r ← j − 1
      else if r = j − 1
        r ← j
    else if paran(j) and a_j ≠ 0
      if a_{j-2} ≠ 0
        a_{j-1} ← 1
        a_j ← 0
        if r = j and j > 1
          r ← j − 1
        else if r = j − 1
          r ← j
      else
        a_{j-2} ← 1
        a_j ← 0
        if r = j
          r ← maksimum(j − 2, 1)
        else if r = j − 2
          r ← j − 1
    else if paran(j) and a_j = 0
      a_j ← 1
      a_{j-1} ← 0
      if r = j and j > 1
        r ← j − 1
      else if r = j − 1
        r ← j
    else if neparan(j) and a_j = 0
      if a_{j-1} ≠ 0
        a_j ← 1
        a_{j-1} ← 0
        if r = j and j > 1
          r ← j − 1
        else if r ← j − 1
          r ← j
      else
        a_j ← 1
        a_{j-2} ← 0
        if r = j − 2
          r ← j
        else if r = j − 1
          r ← j − 2
end
```

За генерисање комбинација у форми листе индекса $c_t \dots c_2 c_1$, потребно је благо променити нотацију, и дефинисати $C_t(n)$ за C_{st} и $C_t(n)$ за C_{st} када је $s + t = n$. Индукција се тада, за $t \geq 0$, дефинише на следећи начин:

$$C_0(n) = \epsilon$$

$$C_{t+1}(n+1) = \begin{array}{l} nC_t(n), C_{t+1}(n) \text{ ако је } n \text{ паран;} \\ nC_t(n), C_{t+1}(n) \text{ ако је } n \text{ непаран} \end{array} \tag{2.25}$$

$$C_{t+1}(n+1) = \begin{array}{l} C_{t+1}(n), nC_t(n) \text{ ако је } n \text{ непаран} \\ C_{t+1}(n), nC_t(n) \text{ ако је } n \text{ паран} \end{array} \tag{2.26}$$

На пример, за $n = 9$, када се претходне једнакости развију, добија се:

$$\begin{array}{l} C_{t+1}(9) = 8C_t(8), 6C_t(6), 4C_t(4), \ldots, 3C_t(3), 5C_t(5), 7C_t(7); \\ C_{t+1}(8) = 7C_t(7), 6C_t(6), 4C_t(4), \ldots, 3C_t(3), 5C_t(5); \\ C_{t+1}(9) = \quad 6C_t(6), 4C_t(4), \ldots, 3C_t(3), 5C_t(5), 7C_t(7), 8C_t(8); \\ C_{t+1}(8) = \quad 6C_t(6), 4C_t(4), \ldots, 3C_t(3), 5C_t(5), 7C_t(7); \end{array} \tag{2.27}$$

Све четири секвенце имају исти шаблон, а значење „...” зависи од вредности t, изостављају се $nC_t(n)$ и $nC_t(n)$ где је $n < t$. Сви развоји су, осим неколико изузетака на почетку и крају, засновани на бесконачној прогресији:

$$\ldots, 10, 8, 6, 4, 2, 0, 1, 3, 5, 7, 9, \ldots \tag{2.28}$$

која представља природан начин да се ненегативни цели бројеви уреде у двосмерно бесконачну листу. Уколико се из (2.28) избаце сви елементи који су мањи од $t, t \geq 0$, остатак листе задржава својство да се суседни елементи разликују за 1 или 2.
На основу индуктивних формула (2.25) и (2.26) једноставно се добија рекурзивни алгоритам, али је такође могућ и итеративни алгоритам користећи развоје попут (2.27), тако се добија алгоритам сличан алгоритму Ц. Такав алгоритам захтева $O(t)$ простора, и посебно је ефикасан када је t мало у односу на n.

Поглавље 3

Програмска реализација и резултати

3.1 Програми који реализују алгоритме

Алгоритми приказани у претходном поглављу су имплементирани у склопу конзолне апликације GenerisanjePermutacijaIKombinacija. Апликација је написана у програмском језику C++, а за превођење се може користити било који преводилац компатибилан са C++98 стандардом.

Изворни код организован је у три целине. Алгоритми везани за комбинације налазе се у оквиру класе Kombinacije, алгоритми везани за пермутације у оквиру класе Permutacije. Класа PomocneFunkcije садржи функције за испис и форматирање резултата.

Након превођења и покретања, отвара се конзолна апликација у којој се прави избор алгоритма за генерисање пермутација или комбинација. Корисник изборе прави преко тастатуре, тако што бира слово које је назначено у загради за жељени избор. За сваки алгоритам се може изабрати један од два начина приказа: први је детаљан и укључује испис међукорака и вредности променљивих током рада алгоритма, док други исписује само генерисане пермутације, односно комбинације. Примери рада апликације за све алгоритме се налазе у прилогу рада и преузети су из оригиналног текста.

3.2 Поређење алгоритама за генерисање пермутација и комбинација

Имплементирани алгоритми су експериментално упоређени. Изабрани су алгоритми који генеришу све пермутације, и алгоритми који генеришу све комбинације, и тестирано је до које границе се извршавају и за које време завршавају генерисање. Добијени резултати су представљени табелом и дијаграмом.
За поређење пермутација изабрани су следећи алгоритми:

- алгоритам за генерисање пермутација у лексикографском поретку Л,
- алгоритам простих замена П,
- општи алгоритам за генерисање пермутација уз коришћење Симсове табеле за генерисање пермутација у реверсном колекс поретку Г

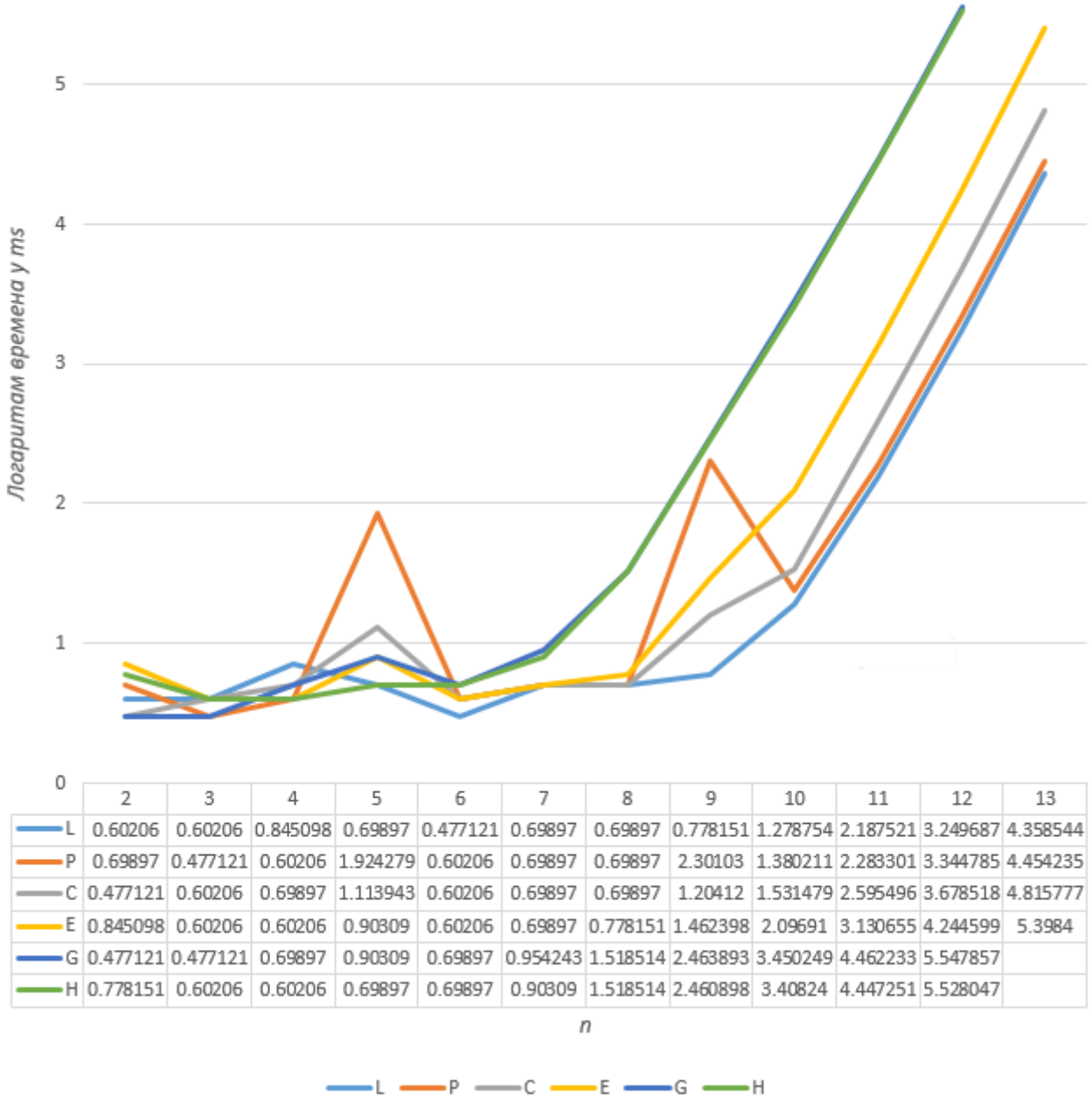
- дуални алгоритам за генерисање пермутација уз коришћење Симсове табеле за генерисање пермутација у реверсном колекс поретку Х
- алгоритам за генерисање пермутација цикличним померајима Ц
- алгоритам за генерисање пермутација Ерлиховим заменама Е

Табела и дијаграм на слици 3.1 приказују логаритам времена изражен у милисекундама у зависности од n. Алгоритми су, са изузетком алгоритама Г и Х, радили до $n = 13$. Са графика се може видети да је најефикаснији алгоритам за мале вредности n основни и најједноставнији алгоритам за генерисање Л. Уочљиво је и да време генерисања за све алгоритме нагло расте за $n > 10$.

За поређење комбинација изабрани су следећи алгоритми:

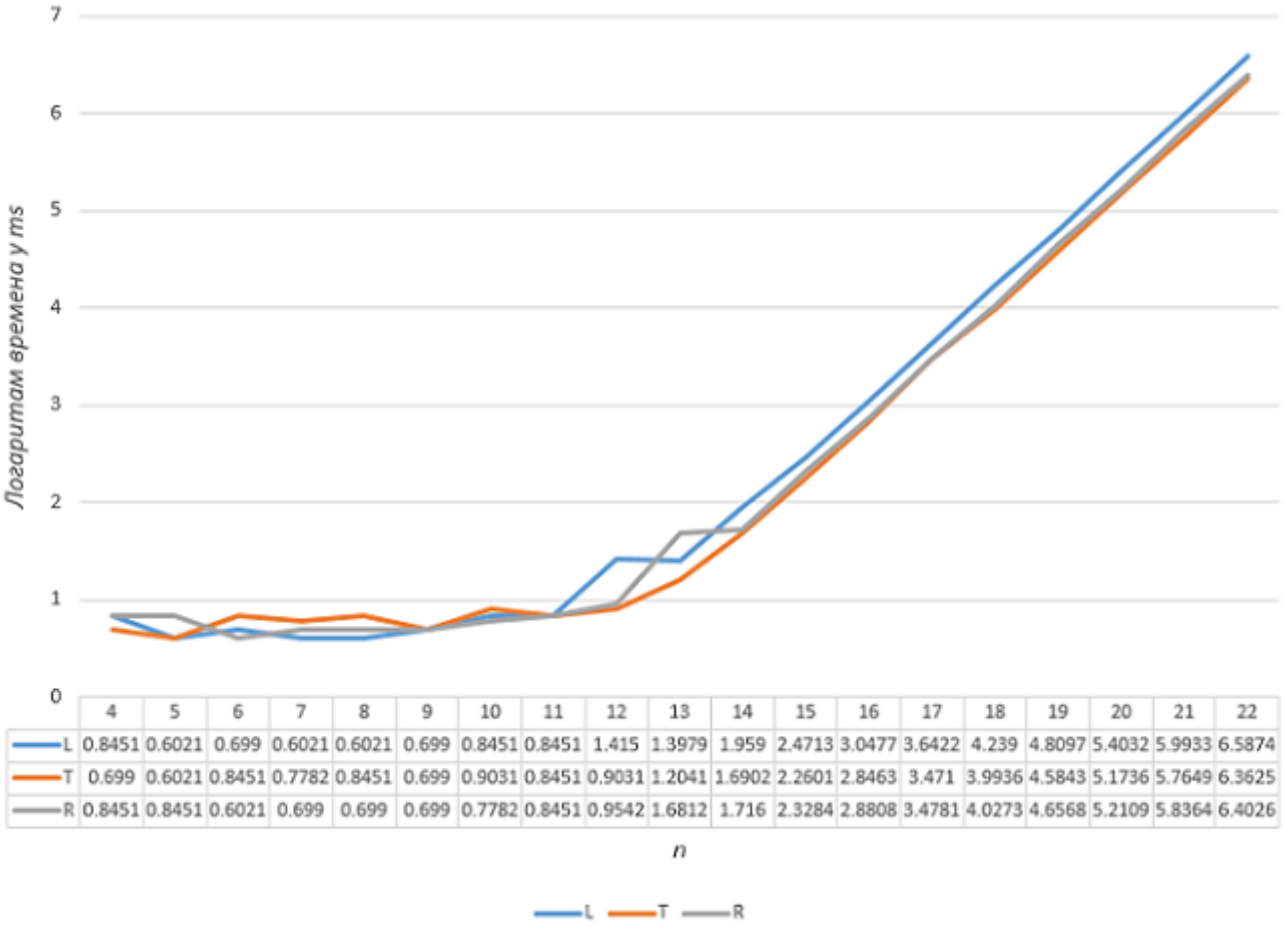
- алгоритам за генерисање комбинација у лексикографском поретку Л,
- унапређени алгоритам за генерисање комбинација у лексикографском поретку Т
- алгоритам за генерисање комбинација методом обртних врата Р

Алгоритми су генерисали све n-комбинације од $2n$ елемената. Табела и дијаграм на слици 3.2 приказују логаритам времена изражен у милисекундама у зависности од n. Са дијаграма се закључује да је најефикаснији алгоритам Т, потом алгоритам Р и на крају алгоритам Л. Време рада алгоритама нагло расте за $n > 13$.



	2	3	4	5	6	7	8	9	10	11	12	13
L	0.60206	0.60206	0.845098	0.69897	0.477121	0.69897	0.69897	0.778151	1.278754	2.187521	3.249687	4.358544
P	0.69897	0.477121	0.60206	1.924279	0.60206	0.69897	0.69897	2.30103	1.380211	2.283301	3.344785	4.454235
C	0.477121	0.60206	0.69897	1.113943	0.60206	0.69897	0.69897	1.20412	1.531479	2.595496	3.678518	4.815777
E	0.845098	0.60206	0.60206	0.90309	0.60206	0.69897	0.778151	1.462398	2.09691	3.130655	4.244599	5.3984
G	0.477121	0.477121	0.69897	0.90309	0.69897	0.954243	1.518514	2.463893	3.450249	4.462233	5.547857	
H	0.778151	0.60206	0.60206	0.69897	0.69897	0.90309	1.518514	2.460898	3.40824	4.447251	5.528047	

Слика 3.1: Поређење алгоритама за генерисање пермутација



	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
L	0.8451	0.6021	0.699	0.6021	0.6021	0.699	0.8451	0.8451	1.415	1.3979	1.959	2.4713	3.0477	3.6422	4.239	4.8097	5.4032	5.9933	6.5874
T	0.699	0.6021	0.8451	0.7782	0.8451	0.699	0.9031	0.8451	0.9031	1.2041	1.6902	2.2601	2.8463	3.471	3.9936	4.5843	5.1736	5.7649	6.3625
R	0.8451	0.8451	0.6021	0.699	0.699	0.699	0.7782	0.8451	0.9542	1.6812	1.716	2.3284	2.8808	3.4781	4.0273	4.6568	5.2109	5.8364	6.4026

Слика 3.2: Поређење алгоритама за генерисање комбинација

Поглавље 4

Закључак

У овом раду представљени су најважнији неелементарни алгоритми за генерисање пермутација и комбинација, описани у поглављима 7.2.1.2 и 7.2.1.3 књиге *Уметност рачунарског програмирања*. Алгоритми су представљени прецизно на псеудојезику, елиминисане су „go to ” команде и имплементирани су у склопу пратеће конзолне апликације. Имплементирани алгоритми су потом експериментално упоређени.

Како је циљ овог рада јасна демонстрација алгоритама и њиховог рада, основна употребна вредност овог рада је едукативна, односно за разумевање алгоритама. Стога је посебно погодан детаљни мод рада у направљеној апликацији, који исписује међукораке и вредности променљивих за све алгоритме.

Осим тога, направљена имплементација може послужити као добра основа за израду унапређеније и ефикасније верзије алгоритама за генерисање комбинација и пермутација.

Литература

[1] Donald E. Knuth, The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations, Addison–Wesley, 2005
[2] Donald E. Knuth, The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions, Addison–Wesley, 2005
[3] https://en.wikipedia.org/wiki/The_Art_of_Computer_Programming
[4] Toshinori Munakata, Application Areas of Combinatorics, Especially Permutations and Combinations, 2005
[5] Donald E. Knuth, The Art of Computer Programming, Volume 3, Section 5.1.: Combinatorial properties of permutations, Addison–Wesley, 1973
[6] J. A. H. Hunter, Globe and Mail, 27, 1955
[7] Charles C. Sims, Computational methods in Abstract Algebra, 169-173, Oxford:Pergamon, 1970
[8] https://en.wikipedia.org/wiki/Mixed_radix
[9] G. G. Langdon, Jr. CACM 10 (1967), 298-299; 11 (1968) 392
[10] Nijenhuis and Wilf, Combinatorial Algorithms, Exercise 6, 1976
[11] Arthur Cayley, American J. Math 1, 174-176, 1878
[12] R. A. Rankin, Proc. Cambridge Philos.Soc. 44 17 - 25, 1948
[13] Ruskey, Jiang and Weston, Discret Applied Math. 57 75 - 83, 1995
[14]R. C. Compton and S. G. Williamson, Linear and Multilinear Algebra 35, 237 - 293, 1993
[15] Y. L. Varol and D. Rotem, Comp. J. 24, 83 - 84, 1984
[16] T.A. Jenkuns, D. McCarthy, Ars Combinatoria 40, 153-159, 1995
[17] R. J. Ord-Smith, CACM 10, 452, 1967; CACM 12, 638, 1969
[18] B. R. Heap, Comp. J. 6, 293-294, 1963
[19] M.C.Er, Comp. J. 30, 282, 1987