МАТЕМАТИЧКА ГИМНАЗИЈА

МАТУРСКИ РАД

из програмирања и програмских језика

ПРОГРАМ ЗА НАЛАЖЕЊЕ НАЈБОЉЕГ РЕШЕЊА У ИГРИ „МОЈ БРОЈ“

Ученик:
Дејан Томић, 4д

Ментори:
Станка Матковић
Мијодраг Ђуришић

Београд, мај 2011.

Садржај

1. УВОД

Игра „Мој број“ је једна од игара из популарног домаћег квиза „ТВ слагалица“ који се емитује сваког радног дана на РТС-у. Циљ игре је да се од 6 понуђених, случајно изабраних бројева (не обавезно различитих), коришћењем четири основне математичке операције (сабирања, одузимања, множења и дељења) и заграда састави израз чија је вредност једнака, или што приближнија, задатом броју (другачије званом и „тачан“ или „тражени“ број). Сваки од 6 понуђених бројева се може употребити највише једном у изразу. Прва четири од шест понуђених бројева су једноцифрени (али различити од нуле). На петом месту је један од бројева 10, 15, 20 или 25, а на последњем 25, 50, 75 или 100.

Програм описан у овом раду је првенствено намењен решавању проблема из игре „Мој број“. Осим што се бројеви окрећу случајним редом и могу да се зауставе, баш као и у игри „Мој број“, могу да се задају и бројеви по жељи. Једноставно се уместо дугмета „СТОП“ може кликнути на „Постави број“ и изабрати број који ће бити уписан после заустављања. Након избора бројева корисник може да саставља математички израз, а програм да рачуна његову вредност, или може програм сам да тражи решење – израз чија је вредност тражени број, или ако такав не постоји, онда онај чија је вредност најближа траженом броју.

Програм има разне оптимизације у алгоритму тражења најбољег решења, што га чини веома брзим. Један од главних циљева израде програма био је управо брзина рада. Испробано је неколико различитих начина за тражење решења, а први од њих је био обичан *brute-force*; затим су испробане рекурзивне и нерекурзивне методе, без и са памћењем израчунатих резултата; рачунање „унапред“ и „уназад“ (где се полази од тачног броја); и коначно, у неким методама, разне математичке оптимизације којима се избегава рачунање неких непотребних израза (пример је комутативност сабирања и множења) и упоређивање брзине рада. О свим овим методама и њиховој ефикасности биће речи касније.

Програм има могућност тражења свих решења, или може после једног нађеног решења (у случају да је нашао тачан број) да стане. Решења су суштински различита – с обзиром на то да су у програм убачене већ поменуте математичке оптимизације (а поред комутативности и многе друге), онда су „слична“ решења самим тим већ избачена.

2. ПРОГРАМСКИ ИНТЕРФЕЈС

Програмски интерфејс је једноставан и сличан ономе у игри „Мој број“ у квизу „ТВ слагалица“. У горњем делу програма се налазе 3 поља са по једном цифром траженог броја, а испод њих још 6 дугмића, на којима су написани понуђени бројеви, на сваком по један, и распоређени исто као и у квизу (прво иду 4 једноцифрена). Испод тих бројева је дугме за почетак нове игре, односно дугме „СТОП“ када се бројеви окрећу (увек се види једно од та два дугмета). Одмах испод дугмета „СТОП“ налази се дугме са менијем за постављање броја – кликом на то дугме може се изабрати било који број који је дозвољен на тренутној позицији (по правилима квиза), и он ће бити уписан уместо тренутног, случајног броја. Након постављања једног броја, или кликом на „СТОП“, тренутни број остаје уписан на својој позицији и почиње да се врти следећи број, све до последњег након којег игра почиње.

Испод дугмета „СТОП“ (и оног за постављање броја) се налази ред дугмића са математичким операцијама и заградама, а десно од њих дугме за преправљање израза, рачунање резултата, и дугме за аутоматско решавање (добијање тачног, односно њему најближег могућег броја). Осим на дугмиће са математичким операцијама и заградама, може се кликтати и на дугмиће са понуђеним бројевима. Свако дугме са бројем се искључује након једног клика, те се као по правилима квиза не може један број искористити више пута. Помоћу дугмета за исправку може се обрисати последњи уписан број или операција, у случају броја на тастер са одговарајућим бројем се може поново кликнути. Програм води рачуна о томе да унети израз не буде неисправан: на почетку израза, као и после математичке операције, може се унети само број, или отворена заграда; после броја се може унети или операција, или затворена заграда под условом да њих има мање у изразу него отворених заграда. Самим тим се не може унети више од 5 математичких операција у изразу. Такође, при клику на дугме за рачунање проверава се да ли је крај израза исправан – на крају може бити или број, или затворена заграда – као и да ли су све заграде затворене (не може их на крају бити мање него отворених!) На самом дну програма се налазе два реда поља, у првом пише тренутни израз који корисник саставља, и његова вредност након клика на дугме за рачун, а испод њих такође израз и његова вредност – али у овом случају за оптимално решење добијено након клика на дугме за решавање.

Слика 1: Прозор програма "Мој број"

2.1. Подешавања игре; главни мени

Програм поседује и главни мени, помоћу кога се могу подесити неке ствари. Први део менија омогућава започињање нове игре, баш као и већ наведено дугме „Нова игра“, као и излаз из програма.

Следећа ставка у менију је намењена подешавању интерфејса игре, и има четири опције. Прва омогућава или онемогућава постављање броја (на почетку је постављање броја омогућено); друга омогућава клик на дугме за решавање било кад (и пре израчунавања првог укуцаног израза, односно одмах након што се одреде сви бројеви) – и она је на почетку искључена. Трећа опција онемогућава исправке израза након што се једном кликне на дугме за његово израчунавање. Тиме се једном састављен и потрвђен / израчунат израз узима као коначан, као што наравно важи и у квизу. Четврта опција ради само када је трећа укључена (а на почетку није) и омогућава аутоматско решавање – чим корисник једном кликне на дугме за рачунање, будући да не може више да преправља свој израз, компјутер одмах почиње с тражењем оптималног решења.

Трећа ставка у менију служи подешавању компјутерског решавања. Прве три опције у овој ставци се тичу дозвољених резултата: прва опција не дозвољава уопште коришћење не-целих бројева (што значи да ни један део израза не сме да буде разломак, односно нема дељења осим с оним с чим је дељиво без остатка); док друга опција дозвољава таква дељења само под условом да коначан резултат буде цео број – односно, изрази чији резултат није цео број неће улазити у избор оптималних. Трећа опција омогућава да у избор оптималних решења уђу и изрази са резултатом већим од (или једнаким) 1000. Некада у квизу „ТВ слагалица“ нису били дозвољени бројеви од 1000 и већи као коначни резултат; те би онда у неким случајевима (рецимо кад је тражени број 999) најближи могући број био мањи од траженог, јер иако је број 1000 можда још ближи, он није дозвољен као коначан резултат.

Четврта опција постоји ради статистике, и она када се укључи, након решавања проблема се, испод израза и добијеног најближег броја, исписује време рада алгоритма за решавање, као и укупан број израза који је израчунат током тог процеса.

Укључивањем последње опције у трећој ставци менија, омогућава се испис свих оптималних решења, уместо само једног. Наравно, тада ће процес тражења решења (у случају када може да се добије тачан број) трајати дуже, јер алгоритам неће стати након првог добијеног израза који даје тачан број. Исписивање свих оптималних решења важи и када оптимално решење није тачан број – у том случају се исписују и изрази који дају мањи, и они који дају већи број – јер, на пример, за тражени број 543 оптимална решења могу бити бројеви 542 и 544 јер се оба разликују за 1 од траженог.

Треба напоменути и то да су решења **суштински различита** – односно, решења добијена операцијама комутативности, асоцијативности, и уопште на потпуно исти начин као неко друго решење, не узимају се у обзир. Ово може бити јако корисна опција за све радознале људе, које занимају други начини на које неки број може да се добије.

Последња, четврта ставка у менију говори нешто мало о аутору.

3. МЕТОДЕ РАДА ПРОГРАМА

У овом поглављу су детаљно описани начини на које програм ради, као и други могући начини рада и њихове ефикасности и мане.

3.1. Рачунање израза

Први проблем у програму је тај, како израчунати израз који корисник упише? За то постоји неколико начина. Израз се може рачунати на „људски начин“, директно пролазећи кроз стринг и памтећи леви операнд и операцију, и затим прочитати (или израчунати, ако је израз) десни операнд, и онда применити запамћену математичку операцију и израчунати вредност.

Овај први, „људски начин“ се углавном не користи у рачунарским програмима. Најлакши начин за имплементацију би био рекурзиван – ако се у изразу наиђе на један број, он се памти као први операнд, чита се операција и онда и други операнд и затим се израчунава вредност израза. Међутим, ако се наиђе на отворену заграду, онда се улази у рекурзију и рачуна се нови израз, до затворене заграде... Када се његова вредност израчуна она се памти као леви или десни операнд, у првом случају се наставља читање израза (операција и десни операнд) а у другом се примењује математичка операција и рачуна вредност израза. Приоритет операција множења и дељења додатно компликује овај метод рачунања, а најлакши начин да се то реши је додавањем заграда око тих операција, под условом да ниједан од операнада над којим се извршава није већ под заградом.

Уместо у *инфиксној нотацији*, стандардној за људе, математички израз се може записати и у *префиксној* и *постфиксној* нотацији. Израз у префиксној нотацији има математичке операције испред оба операнда, а у постфиксној – иза оба операнда. Посебна предност ових нотација је што нема потребе за заградама – сам распоред операција и бројева одређује и којим ће се редом, односно са којим приоритетом, извршавати математичке операције. Тиме се, дакле, при рачунању не мора водити рачуна ни о приоритету операција, већ се оне само извршавају редом (од прве до последње у постфиксној, односно од последње до прве у префиксној нотацији).

Мало већи проблем од рачунања израза у постфиксној нотацији је превођење израза из инфиксне нотације у постфиксну. То се може урадити на следећи начин:

- Прво се иницијализују два празна *стека*: један у коме ће бити тражени израз у постфиксној нотацији (RPN), а у другом, помоћном, ће бити операције и заграде.
- Сада се из израза у инфиксном облику узима један по један елемент и обрађује:
    - Ако је елемент отворена заграда, она се ставља на помоћни стек.
    - Ако је затворена заграда, тада се са помоћног стека све операције до отворене заграде пребацују на главни стек. Отворена заграда се уклања са стека, али се не додаје на главни стек јер се у постфиксној нотацији заграде не користе.

    - Бројеви се стављају на главни стек.
    - Математичке операције се стављају на помоћни стек, али уз претходну проверу приоритета рачунања: пре него што се операција сабирања или одузимања дода на помоћни стек, са њега се уклањају и пребацују на главни стек све операције до заграде; уколико је у питању операција множења или дељења, само се те две операције пребацују са помоћног на главни стек (све док последњи елемент на помоћном стеку не буде заграда, или операција нижег приоритета).
- На крају, када се цео инфиксни израз препише, преостале операције са помоћног стека (на коме више не би смело да буде заграда) пребацују се на главни стек.

Рачунање израза у постфиксној нотацији, другачије званој и *обрнута пољска нотација*, тривијално је: на стек се додаје један по један број из израза, а када се наиђе на операцију, она се извршава над два последња елемента на стеку, након чега се она бришу и уместо њих уписује резултат операције. Када се дође до краја израза и изврши последња операција, на стеку треба да остане само један број који је резултат траженог израза.

Поменимо још један од могућих начина за рачунање математичког израза. Наиме, исти се може чувати и у *бинарном стаблу*: бројеви се уписују у *листове* стабла, а у остале чворове стабла се уписују операције. У корену стабла је такође операција и то она са нижим приоритетом (која се последња рачуна). Имплементација овог решења је као и у „људској методи“ рекурзивна – при наиласку на заграду улази се у рекурзију и даљи израз се уписује у стабло чији је корен лева или десна грана главног стабла... Дакле, решење је слично као и без коришћења бинарног стабла. За рачунање израза записаног у стаблу довољан је *in-order* обилазак стабла. Међутим, то је такође рекурзивно и поново се пролази кроз цео израз, те је директно рачунање дупло брже. Због тога се ова метода не препоручује.

У програму је имплементирана друга метода – дакле, израз се преводи у постфиксну нотацију и затим у том облику рачуна.

3.2. Тражење најбољег решења

Ово је главни задатак програма. Циљ је да тражење решења буде што брже. Због тога је испробано неколико начина за тражење најбољег решења, како би се дошло до оног најбржег. Следећи начини су испробани:

- Стандардни *brute-force search* – испробавање свих могућих израза (и потребних и непотребних). Овде су се изрази састављали и чували у *постфиксној нотацији*, и као такви брзо рачунали. Најлакша имплементација овог алгоритма је састављање свих могућих пермутација од 6 понуђених бројева (укупно 6!), свих могућих комбинација од 5 математичких операција (укупно 4^5), и свих могућих међусобних распореда операција и бројева (односно свих оних распореда који могу чинити исправан израз у постфиксној нотацији). То је укупно око 32 милиона математичких израза.

- Brute-force алгоритам се може мало оптимизовати, тако што се одбацују идентичне пермутације – на пример, ако имам више истих бројева (1 1 1 2 25 25), они не морају да се распореде на свих 6! начина, већ ће у овом случају број пермутација бити (6!÷3!÷2!). Још неке оптимизације (на пример правило комутативности) би се могле убацити у овај алгоритам, међутим он је ипак одбачен јер је и упркос тим оптимизацијама и даље био много спор.
- Уместо састављања свих могућих израза у постфиксној нотацији, решење се може тражити и овако: имамо 6 понуђених бројева; додајмо један број са једне стране израза, израз од преосталих 5 бројева са друге стране израза, а у средину убацимо операцију и израчунајмо тај израз. Овај алгоритам би био рекурзивни (јер би се и за те изразе са десне стране применио исти алгоритам за њихово састављање и рачунање). Након свих таквих комбинација, са леве стране би се додавали изрази са по 2 броја, а са десне са преостала 4; затим са 3 броја и тако даље. Овај цео процес чак не би морао да се понавља у случају када се употребљава само 5 (или мање) бројева, јер би сви изрази са мање од 6 употребљених бројева већ били израчунати у процесу рекурзије, те би њихова вредност већ тада могла да се упореди са траженим бројем и израз сачува у случају да је ближи од свих досадашњих.
- У претходном алгоритму се одмах може приметити то да се један те исти израз може рачунати пуно пута (то је познати проблем у рекурзивним алгоритмима). Због тога овај метод може бити много спорији чак и од *brute-force* алгоритма. Да би се то избегло, сви израчунати изрази се могу памтити у једном или више низова, те би се изрази, уместо да се (рекурзивно) састављају из почетка, они само узимали из низа. Очигледно је да се у том низу, осим самих израза, може памтити и њихова вредност. Заправо, за рачунање вредности једног израза, довољно је знати само вредности два израза (са леве и десне стране) која се у њему користе. Тиме се рачунање било ког израза своди на извршавање једне једине математичке операције, а не као у brute-force алгоритму на рачунање читавог израза. Овим се још више убрзава рад програма.

Како би онда радио овај последњи алгоритам?

Прво би се додало свих 6 понуђених бројева у један низ, и ти бројеви би се запамтили као њихове вредности. Затим би се рекурзивно састављали изрази од 6 бројева на већ описани начин: са низа би се узимао један по један број и стављао са леве стране новог израза, у комбинацији са сваким изразом од 5 преосталих бројева са десне стране, и сваком од четири математичке операције. Пошто до тог момента још нису израчунати сви изрази са по 5 бројева, они ће се састављати рекурзивно, на исти начин као и изрази са 6 бројева, након чега ће бити обележено да су сви изрази састављени од тих 5 бројева израчунати. Следећи пут када би се у рекурзији користили изрази састављени од тих 5 бројева, они би се само узимали из низа са већ израчунатом вредношћу уместо да се састављају поново.

Може се закључити да ће на овај начин прво бити израчунати изрази са по 2 броја (али не баш са било која 2), па онда са 3 и тако даље. Најбоље би било у овом алгоритму користити 2^6=64 стека, за сваку комбинацију искоришћених бројева. Такође, да би се на

крају налажења решења могао исписати израз који даје најбоље решење, у сваком изразу у низу морале би се запамтити и референце на два мања израза, односно појединачна броја, која су се користила у њему. То значи да би се изрази чували у облику *бинарног стабла* – где би корен стабла била операција, а свака грана би била или појединачни број (*лист* стабла), или други мањи израз (такође стабло).

Наведени алгоритам може бити веома брз, мада са једном једином маном: заузима пуно меморије. Међутим та количина меморије није превелика (неколико мегабајта), иако је значајно већа од оне коју заузима brute-force алгоритам (скоро нимало). А пошто је брзина свакако главни фактор, ипак је боље користити овај алгоритам него brute-force. Такође, овај алгоритам се лако може превести у нерекурзивни – који би чак био и бољи и ефикаснији. Заправо, овде се могу, уместо 64 низа, користити 6 низова. Како би овај алгоритам радио?

- Прво би се свих 6 понуђених бројева убацило у први низ, као и у рекурзивном алгоритму.
- Затим би се састављали сви изрази са по 2 броја, тако што се са првог низа узме један број, стави са леве стране израза, други (различити од првог) број са десне стране и једна математичка операција. Сваки такав израз би се додавао на други низ, уз памћење референци на два искоришћена броја, операцију и вредност израза.
- Након израза са 2 по броја, састављају се сви они са по 3 броја – са првог низа се узима један по један број, а са другог један по један израз са по 2 броја. Рачунају се и додају на трећи низ на исти начин као и изрази са по 2 броја. Након што се сви појединачни бројеви додају са леве стране, сада се са леве стране додају изрази са по 2 броја – а са десне стране, појединачни бројеви. Иста правила важе.
- На исти начин се рачунају и убацују у одговарајући низ и изрази са по 4, 5 или свих 6 искоришћених бројева.

Овај алгоритам је мало бржи од своје рекурзивне верзије, а може се још више убрзати тако што се у њега убаце неке оптимизације – као и у brute-force алгоритам. На пример, једна тривијална оптимизација би била провера комутативности: ако је са леве стране израз, који има више бројева у себи него онај са десне стране, тада над њима нећемо извршавати операције сабирања или множења (оне су већ извршене у супротном распореду израза). Ово исто важи и у случају да изрази имају исти број бројева (тј да су узети са истог низа) – али је један „раније“ у том низу него онај други.

Будући да се у сваком изразу памти операција и референце на леви и десни под-израз, лако се може увести и правило асоцијативности – опет се проверава који је израз „раније“ у низу и са десне стране се узима само онај последњи (наравно, само у случају да је тренутна операција за извршавање иста као и одговарајућа операција у изразу са леве стране).

Такође, једна од тривијалних оптимизација може бити избегавање множења или дељења са 1: само се провери резултат израза који је са десне стране (а у случају множења, и са леве). Слично важи и за сабирање, односно одузимање нуле. Заправо, најбоље је да се ниједан израз који има резултат 0 ни не додаје у низ. Јер, нула не може да да никакав нови

резултат. Такође, не треба додавати у низ ни изразе који имају негативну вредност – крајњи резултат мора бити позитиван, а сваки негативан израз има и свој еквивалентат позитиван израз (негативна вредност се може добити једино одузимањем A-B где је B>A – уместо тога, може се одузети A од B). Да би се касније добио позитиван резултат, од њега се мора или одузети други негативан израз (што се опет може „обрнути“ тако да нема негативних под-израза), или сабрати позитиван израз са већом апсолутном вредношћу (ово се опет може „обрнути“), или се може помножити / поделити са нечим негативним (такође се може „обрнути“ тако да оба израза буду позитивна, па да се добије иста вредност). Закључак је да са негативним вредностима, баш као и са нулом, не треба уопште радити. Зато се уводи и, такорећи, „комутативност одузимања“ – треба одузимати само израз са мањом вредношћу од оног са већом.

Наведене, као и још неке оптимизације, убачене су у програм. Још само једна ствар коју треба напоменути је та да се у низу чува и листа свих бројева који су коришћени у неком изразу – како би се брже могло проверити да ли нека два израза могу да се комбинују (јер не сме исти број да се користи више пута). Та листа бројева се чува као један цео број, а њено постављање (приликом убацивања израза у низ) и проверавање се врше помоћу операција над битовима – операција конјункције приликом провере (резултат мора бити 0), а операција дисјункције приликом убацивања у низ. N-ти понуђени број има вредност 2^N у тој листи.

Приликом додавања оптимизација у алгоритам, појавила су се следећа два проблема:

1. Да ли одређена оптимизација захтева пуно времена за проверу?
2. Да ли неке оптимизације, када се комбинују, избацују нека исправна решења?

У првом случају, оптимизацију не вреди убацивати у програм, јер би се она морала спроводити (уз претходну проверу) пре сваког рачунања, а ако провера траје дуго, укупно време потрошено на провере може прећи време које би та оптимизација потенцијално могла уштедети. Дакле, програм би само радио спорије што желимо да избегнемо.

Понекад, комбиновање неких оптимизација може чак да избаци и нека „исправна“ решења, односно и оне „потребне“ изразе поред „непотребних“. Резултат тога је да програм можда неће наћи најближе решење! Ово посебно желимо да избегнемо, и зато треба или избацити неке оптимизације (почевши од оних „мање ефикасних“), или покушати спојити их тако да и даље не отклањају ниједно „исправно“ решење. Ово друго се често може постићи, али понекад приликом тога нека од „исправљених“ оптимизација постане неефикасна, тј наилази се на први проблем. Због тога треба испробати разне комбинације и на крају оставити најефикаснију која и даље ради исправно (тј увек даје најбоље решење).

3.3. Испис нађеног решења на екран

Последња ствар коју треба решити у програму је та, како исписати нађено најбоље решење на екран. Начин исписа углавном зависи од облика у коме се чува сам израз током тражења решења.

Због тога, први начин који је коришћен у програму преводио је израз из стека у постфиксном облику (обрнутој пољској нотацији) у стринг у инфиксном облику. Израз је чуван у постфиксном облику у brute-force алгоритму. Израз се из постфиксног облика лако може превести у инфиксни, на сличан начин на који се и рачуна: уместо да се математичке операције извршавају, оне се исписују на екран; пре операције се исписује цео израз – леви операнд (уместо да се чува његова вредност), а после ње десни операнд. Наравно, на почетку и крају целог тог израза треба да буде заграда (због евентуалног приоритета извршавања операција – наравно ово не важи увек, јер су заграде потребне само у неким случајевима).

Најлакши начин, заправо, за превођење израза из постфиксног облика у стринг је тај, да се користи помоћни стек стрингова (уместо стека бројева који користи метода за рачунање израза). Тада би се сваки појединачни број из постфиксног израза додавао (након превођења броја у стринг) на стек, а у случају операције, уместо њеног извршавања над два последња елемента и писања резултата, сада би се на стек уписао израз (стринг) састављен од отворене заграде, левог операнда, операције, десног операнда и затворене заграде (и овде су такође леви и десни операнд два последња елемента – стринга на помоћном стеку). На крају би онда на стеку остао само један стринг, који би представљао цео израз преведен из постфиксне нотације.

Будући да се у програму користи нерекурзивни, оптимизовани алгоритам за налажење решења, у коме се израз чува као структура која садржи операцију и референце на два операнда, онда је и алгоритам за испис тог израза на екран морао да буде промењен.

Испис оваквог израза на екран може се постићи такозваним *in-order обиласком* бинарног стабла: рекурзијом се испише прво леви операнд – грана, затим операција (вредност чвора стабла) и најзад десни операнд. Наравно, и овде треба водити рачуна о испису заграда, због приоритета извршавања операција. Међутим, овде је много лакше избећи додавање заграда тамо где нису потребне – јер је лако проверити која је операција „главна“ у левом, односно десном операнду. У случају превођења из постфиксне нотације, та операција би се или морала памтити у посебном стеку, или би стек уместо стрингова садржао структуре које би, поред стринга – израза, имале и информацију о актуелној операцији у њему; у супротном, цео израз би морао да се прочита и у њему пронађе главна операција, што уопште није ефикасно.

Заграде се додају по следећем правилу:

- Ако је тренутна операција сабирање, не додају се заграде ни око једног операнда.
- Ако је операција одузимање, око левог операнда нису потребне задраде; међутим, око десног јесу, у случају да је у њему главна операција истог приоритета (сабирање или одузимање). Али будући да се одузимање збира или разлике избегава (то се решава двоструким одузимањем, односно сабирањем па одузимањем) у алгоритму тражења решења, онда се тај случај неће ни десити при испису решења.

- Ако је операција множење, заграде се додају око операнда у случају да је његова операција нижег приоритета (сабирање или одузимање). Ово важи за оба операнда.
- Ако је операција дељење, исто правило важи као и код множења за леви операнд; међутим, око десног операнда морају увек да се додају заграде (наравно, и дељење производом или количником се такође избегава у алгоритму тражења решења).
- Уколико је било који операнд један број (његова операција има вредност „ништа“), око тог броја се ни у ком случају не додају заграде.

На овај начин добија се исправан израз у инфиксној нотацији, у коме постоје заграде само тамо, где су неопходне. Тај израз се исписује у поље за најбоље решење у прозору игре.

3.4. Тражење свих решења. Чување решења

У случају да корисник одабере да се нађу сва најбоља решења за неки проблем, уместо само једног решења, њихово чување и испис је мало другачији.

Израз који садржи решење памти се као референца на одговарајући израз на стеку, израчунат у процесу тражења решења. Такође би могао да се памти и као стринг, у који би се превео одмах пре чувања. На крају извршавања алгоритма, тај израз би се исписао на екран.

У случају да се тражи само једно решење, оно се памти као једна референца на израз, односно као један стринг. Ако је вредност неког новог израчунатог израза ближа тачном броју него вредност решења, тај нови израз се памти као ново најбоље решење и замењује дотадашње решење.

У случају да се тражи више решења, метод је мало другачији: сада се уместо једне референце, односто једног стринга, чува низ референци, односно стрингова који представљају најбоља решења. Ако се вредност новог израза једнако разликује од тачног броја као и вредност тренутног решења, нови израз се додаје у низ решења као још једно могуће решење. У случају памћења решења као стринг, морала би да се запамти и вредност сваког од тих израза (јер као што је већ речено, за тражени број 222 изрази чија је вредност 221 су исто добра решења као и они чија је вредност 223). Међутим у случају памћења референци на одговарајуће изразе на стеку, то није потребно јер се у изразу на стеку већ памти и његова вредност. Ако је вредност новог израза ближа тачном броју него вредност тренутног најбољег решења, сва решења се бришу са низа а у њему се сада памти само нови израз, као ново боље решење.

Изрази који се додају као најбоља решења морају да испуњавају одређене услове. На пример, резултат израза не сме да буде број који није цео (мада ова опција може да се промени у игри). Због тога се овај услов проверава пре памћења новог решења. Исто важи и за услов да резултат израза не буде једнак или већи од 1000 (уколико је то правило укључено). На стеку који се користи за памћење свих израза и међурезултата, који су

потребни алгоритму за тражење решења, чувају се и они изрази који не испуњавају наведене услове. Они се само не записују као најбоља решења.

Изрази сачувани као решења се на крају преводе у стринг и исписују. У случају да се тражи једно решење, његова вредност и израз преведен у стринг се уписују у одговарајућа два поља на дну прозора. У случају да се траже сва решења, сваки израз у низу се преводи у стринг, а у поље са решењима се исписује стринг састављен од вредности израза, знака „=“ и израза преведеног у стринг. (Ово писање вредности и знака „=“ код сваког израза је, опет, потребно због две различите могуће вредности најбољег решења.)

Након исписа решења, сви стекови, коришћени као међурезултати у алгоритму тражења решења, празне се како би следеће тражење решења почело из почетка, без запамћених резултата другог проблема. Исто важи и за стек са најбољим решењима.

4. ИСЕЧЦИ КОДА ПРОГРАМА

Сада следе делови изворног кода програма, у програмском језику Паскал, за сваку од функција наведених у претходном поглављу.

4.1. Дефиниције коришћених структура

TPNStack је структура у којој се чува израз у обрнутој пољској нотацији. То је низ елемената који могу бити бројеви или математичке операције (па и заграде). Сваки елемент, у случају да је број, има записану и вредност. У случају да није број, вредност се занемарује.

```
TZnak = (zBroj, zPlus, zMinus, zPuta, zPodeljeno,
         zOtvorenaZagrada, zZatvorenaZagrada, zNista);

TPNElement = record
  Broj: Extended;
  Znak: TZnak;
end;

TPNStack = array of TPNElement;
```

TProstIzraz је структура која садржи један израз током процеса тражења најбољег решења. Он садржи референце на леви и десни операнд, операцију, израчунату вредност, листу свих употребљених понуђених бројева, и индекс који означава који је по реду додат на стек. **ListaIzraza** је један стек који садржи све запамћене изразе (има их 6).

```
PProstIzraz = ^TProstIzraz;

TProstIzraz = record
  sign: byte; //opNone=0; opAdd=1; opSub=2; opMul=3; opDiv=4
  val: double;
  numList: byte;
  leftExp, rightExp: PProstIzraz;
  _index: Integer; //koristi se za asocijativnost...
end;

ListaIzraza = array of TProstIzraz;
```

4.2. Изворни код неких функција

Функција **StringToPN** преводи израз – стринг из стандардне нотације у постфиксну нотацију. Користи неколико функција, међу којима су **IzvrsiZagradu** (која пребацује елементе са једног стека на други све до отворене заграде), **GetElement** (која чита један елемент из стринга и додаје га на почетни стек), **StackPNPush** и **StackPNPop** (које служе за убацивање, односно избацивање и враћање последњег елемента на задатом стеку). Користе се три стека: **StackPocetni** у коме се памти цео израз, подељен на елементе стека (бројеви, операције и заграде), **StackNUkupni** у коме се памти израз у постфиксној

нотацији, и **StackNZnakova** који служи као помоћни стек за привремено памћење операција и заграда.

```
procedure IzvrsiZagradu(var IzNiza, UNiz: TPNStack);
var
  i, j, InsertIndex: integer;
begin
  for j := Length(IzNiza)-1 downto 0 do
    if IzNiza[j].Znak = zOtvorenaZagrada then Break;
  InsertIndex := Length(UNiz);
  SetLength(UNiz, InsertIndex+Length(IzNiza)-1-j);
  for i := Length(IzNiza)-1 downto j+1 do begin
    UNiz[InsertIndex] := IzNiza[i];  Inc(InsertIndex);
  end;
  if j<0 then SetLength(IzNiza, 0)
  else SetLength(IzNiza, j);
end;

function StringToPN(Izraz: string): TPNStack;
var
  DuzinaNiza, i: integer;
  StackPocetni, StackNUkupni, StackNZnakova: TPNStack;
  ZadnjiElement: TZnak;
begin
  DuzinaNiza := 0;
  while Izraz <> '' do begin
    Inc(DuzinaNiza); SetLength(StackPocetni, DuzinaNiza);
    StackPocetni[DuzinaNiza-1] := GetElement(Izraz);
  end;
  for i := 0 to DuzinaNiza-1 do
    case StackPocetni[i].Znak of
      zBroj: StackPNPush(StackNUkupni, zBroj, StackPocetni[i].Broj);
      zOtvorenaZagrada: StackPNPush(StackNZnakova, zOtvorenaZagrada,
0);
      zZatvorenaZagrada: IzvrsiZagradu(StackNZnakova, StackNUkupni);
      zPlus, zMinus: begin
        ZadnjiElement := ZadnjiElementNiza(StackNZnakova);
        while ZadnjiElement in [zPlus,zMinus,zPuta,zPodeljeno] do
begin
          StackPNPush(StackNUkupni, ZadnjiElement, 0);
          StackPNPop(StackNZnakova);
          ZadnjiElement := ZadnjiElementNiza(StackNZnakova);
        end;
        StackPNPush(StackNZnakova, StackPocetni[i].Znak, 0);
      end;
      zPuta, zPodeljeno: begin
        ZadnjiElement := ZadnjiElementNiza(StackNZnakova);
        if ZadnjiElement in [zPuta, zPodeljeno] then begin
          StackPNPush(StackNUkupni, ZadnjiElement, 0);
          StackPNPop(StackNZnakova);
        end;
        StackPNPush(StackNZnakova, StackPocetni[i].Znak, 0);
      end;
    end;
  IzvrsiZagradu(StackNZnakova, StackNUkupni);
  Result := StackNUkupni;
end;
```

Функција **IzracunajPN** рачуна израз записан у постфиксној нотацији:

```
function IzracunajPN(const ulNiz: TPNStack): Extended;
var
  i, j: integer;
  Niz: TPNStack;
begin
  SetLength(Niz, Length(ulNiz));
  for i := 0 to Length(ulNiz)-1 do Niz[i] := ulNiz[i];

  i := 2; //krecemo od treceg elementa (pre njega ne moze biti
operacija)
  while i<Length(Niz) do begin
    if Niz[i].Znak<>zBroj then begin
      Niz[i-2].Broj := IzracunajVrednost(Niz[i-2].Broj, Niz[i-1].Broj,
                                         Niz[i].Znak);
      for j := i+1 to Length(Niz) do Niz[j-2] := Niz[j];
      SetLength(Niz, Length(Niz)-2);
      i := i-1;
    end
    else Inc(i);
  end;
  Result := Niz[0].Broj;
end;
```

Функција **PNToString** преводи израз, записан на стеку у постфиксној нотацији, у форму стринга у инфиксној нотацији, за приказ кориснику. Може се видети да ради веома слично функцији **IzracunajPN**, разлика је једино та што се уместо извршавања операција и памћења резултата, записује стринг састављен од операнада и операције између њих:

```
function PNToString(Niz: TPNStack): String;
var
  i, j: integer;
  S: string;
  Znak: Char;
  CurArray: array of string;
begin
  SetLength(CurArray, Length(Niz));
  for i := 0 to Length(Niz)-1 do
    case Niz[i].Znak of
      zPlus:      CurArray[i] := '+';
      zMinus:     CurArray[i] := '-';
      zPuta:      CurArray[i] := '*';
      zPodeljeno: CurArray[i] := ':';
      zBroj:      CurArray[i] := FloatToStr(Niz[i].Broj);
    end;
  i := 2; //prva dva elementa se preskacu posto su obavezno brojevi
  while i<Length(CurArray) do begin
    Znak := CurArray[i][1];
    if Znak in ['+','-','*','×',':'] then begin
      S := CurArray[i-2]+Znak+CurArray[i-1];
      if i<Length(CurArray)-1 then S := '('+S+')';
      CurArray[i-2] := S;
      for j := i+1 to Length(CurArray)-1 do CurArray[j-2] :=
CurArray[j];
      SetLength(CurArray, Length(CurArray)-2);
      i := i-1;
```

```
    end
    else Inc(i);
  end;
  Result := CurArray[0];
end;
```

Функција **NajbliziBroj** је најкомпликованија и најдужа функција у програму. Она тражи најбоље решење задатог проблема и враћа стринг са изразима и резултатом. Користи се неколико помоћних функција: на пример **Asoc** и **Dist** за проверу асоцијативности, односно дистрибутивности сабирања / одузимања према множењу; затим **PushResult**, функција која додаје израчунати израз на стек (при чему води рачуна о неким правилима, на пример израз се не памти ако је његов резултат 0, или уколико резултат није цео број а корисник је искључио коришћење разломака) – а уз то проверава и да ли је добијен нови најбољи резултат; затим функција **Ispisuj** која исписује израз и његову вредност у одговарајуће променљиве, а такође може и аутоматски да испразни стекове са изразима (позива се на крају извршавања); и најзад, функција **calc** која извршава задату математичку операцију над два задата израза.

```
procedure NajbliziBroj2(cilj: Integer; brojevi: array of Integer;
                        var optVal: string; var optSol: string);
var
  nn: Integer;
  rn: Integer;
  i, j: Integer;
  rez: double;
  tmp: Integer;
  obs: string;
begin
  Tacan := cilj;
  MinRazlika := 99999999;
  SetLength(MinIzrazi, 0);
  for i := 1 to 6 do SetLength(iz[i], 0);
  CALC_OP := 0;
  for i := 0 to 4 do begin
    nn := i;
    for j := i+1 to 5 do
      if brojevi[j] > brojevi[nn] then nn := j;
    tmp:=brojevi[i]; brojevi[i]:=brojevi[nn]; brojevi[nn]:=tmp;
  end;
  for i := 0 to 5 do begin
    PushResult(1, nil, nil, opNone, brojevi[i], 1 shl i);
    if (brojevi[i]=Tacan) and (not NADJI_SVA_RESENJA) then begin
      Ispisuj(@iz[1][i], optVal, optSol);
      exit;
    end; //ako nalazi sva resenja, ne prekidaj
  end;
  for nn := 2 to 6 do
  for rn := 1 to nn div 2 do begin
    for i := 0 to Length(iz[nn-rn])-1 do
    for j := 0 to Length(iz[rn])-1 do begin
      if (iz[nn-rn][i].numList and iz[rn][j].numList) <> 0
      then continue; //koristi se isti broj vise puta!
      if (rn = nn-rn) and (j<=i) then continue; //komutativnost
      //A+B
      if (iz[nn-rn][i].sign<>opSub)
```

```
      and ((iz[rn][j].sign<>opAdd) and (iz[rn][j].sign<>opSub))
      then begin
        if not Dist(iz[rn][j], iz[nn-rn][i])
        and not Asoc(iz[nn-rn][i], iz[rn][j], opAdd) then begin
          rez := calc(iz[nn-rn][i].val, iz[rn][j].val, opAdd);
          PushResult(nn, @iz[nn-rn][i], @iz[rn][j], opAdd, rez,
                     iz[nn-rn][i].numList or iz[rn][j].numList);
          if (rez=Tacan) and (not NADJI_SVA_RESENJA) then begin
            Ispisuj(@iz[nn][Length(iz[nn])-1], optVal, optSol); exit;
          end;
        end;
      end;
      //A-B ili B-A (zavisi koji je veci)
      if iz[nn-rn][i].val > iz[rn][j].val then begin
        if ((iz[rn][j].sign<>opAdd) and (iz[rn][j].sign<>opSub))
        then begin
          if not Dist(iz[rn][j], iz[nn-rn][i])
          and not Asoc(iz[nn-rn][i], iz[rn][j], opSub) then begin
            rez := calc(iz[nn-rn][i].val, iz[rn][j].val, opSub);
            PushResult(nn, @iz[nn-rn][i], @iz[rn][j], opSub, rez,
                       iz[nn-rn][i].numList or iz[rn][j].numList);
          end;
        end;
      end //kraj A-B
      else if iz[nn-rn][i].val < iz[rn][j].val then begin
        if ((iz[nn-rn][i].sign<>opAdd) and (iz[nn-rn][i].sign<>opSub))
        then begin
          if not Dist(iz[rn][j], iz[nn-rn][i])
          and not Asoc(iz[rn][j], iz[nn-rn][i], opSub) then begin
            rez := calc(iz[rn][j].val, iz[nn-rn][i].val, opSub);
            PushResult(nn, @iz[rn][j], @iz[nn-rn][i], opSub, rez,
                       iz[nn-rn][i].numList or iz[rn][j].numList);
          end;
        end;
      end;
      if (rez=Tacan) and (not NADJI_SVA_RESENJA) then begin
        Ispisuj(@iz[nn][Length(iz[nn])-1], optVal, optSol); exit;
      end;
      //A*B
      if (iz[nn-rn][i].val<>1) and (iz[rn][j].val<>1)
      and (iz[nn-rn][i].sign<>opDiv)
      and ((iz[rn][j].sign<>opMul) and (iz[rn][j].sign<>opDiv))
      then begin
        if not Asoc(iz[nn-rn][i], iz[rn][j], opMul) then begin
          rez := calc(iz[nn-rn][i].val, iz[rn][j].val, opMul);
          PushResult(nn, @iz[nn-rn][i], @iz[rn][j], opMul, rez,
                     iz[nn-rn][i].numList or iz[rn][j].numList);
          if (rez=Tacan) and (not NADJI_SVA_RESENJA) then begin
            Ispisuj(@iz[nn][Length(iz[nn])-1], optVal, optSol);
            exit;
          end;
        end;
      end;
      //A/B
      if (iz[rn][j].val<>1)
      and ((iz[rn][j].sign<>opMul) and (iz[rn][j].sign<>opDiv))
      then begin
        if not Asoc(iz[nn-rn][i], iz[rn][j], opDiv) then begin
```

```
          rez := calc(iz[nn-rn][i].val, iz[rn][j].val, opDiv);
          PushResult(nn, @iz[nn-rn][i], @iz[rn][j], opDiv, rez,
            iz[nn-rn][i].numList or iz[rn][j].numList);
          if (rez = Tacan) and (not NADJI_SVA_RESENJA) then begin
            Ispisuj(@iz[nn][Length(iz[nn])-1], optVal, optSol);
            exit;
          end;
        end;
      end;
      //B/A
      if(iz[nn-rn][i].val<>1) {and (iz[nn-rn][i].val<>iz[rn][j].val)}
      and ((iz[nn-rn][i].sign<>opMul) and (iz[nn-rn][i].sign<>opDiv))
      then begin
        if not Asoc(iz[rn][j], iz[nn-rn][i], opDiv) then begin
          rez := calc(iz[rn][j].val, iz[nn-rn][i].val, opDiv);
          PushResult(nn, @iz[rn][j], @iz[nn-rn][i], opDiv, rez,
            iz[nn-rn][i].numList or iz[rn][j].numList);
          if (rez = Tacan) and (not NADJI_SVA_RESENJA) then begin
            Ispisuj(@iz[nn][Length(iz[nn])-1], optVal, optSol);
            exit;
          end;
        end;
      end;
    end;
  end;
  if not NADJI_SVA_RESENJA then begin
    Ispisuj(@(MinIzrazi[0]), optVal, optSol, True); exit;
  end;
  optSol := '';
  for i := 0 to Length(MinIzrazi)-1 do begin
    Ispisuj(@(MinIzrazi[i]), optVal, obs, false);
    if i>0 then optSol := optSol+#13#10;
    optSol := optSol+optVal+' = '+obs;
  end;
  for i := 1 to 6 do SetLength(iz[i], 0); //sad sam isprazni stekove
end;
```

Још једна функција чији ће код бити наведен је **PisiIzraz** (коју позива функција **Ispisuj**). Ова функција преводи израз у стринг, а из оног облика у коме га чува функција **NajbliziBroj**:

```
function PisiIzraz(exp: PProstIzraz): string;
var
  lexp, rexp: PProstIzraz;
  s1, s2, ops: string;
begin
  if exp^.sign = opNone then begin
    result := Format('%.0f', [exp^.value]); {ceo broj}  exit;
  end;
  lexp := exp^.leftExp;
  rexp := exp^.rightExp;
  case exp^.sign of
    opAdd: begin
      ops := '+';
      s1 := PisiIzraz(lexp);
      s2 := PisiIzraz(rexp);
    end;
```

```
    opSub: begin
      ops := '-';
      s1 := PisiIzraz(lexp);
      if (rexp.sign = opAdd) or (rexp.sign = opSub)
      then s2 := '('+PisiIzraz(rexp)+')'
      else s2 :=     PisiIzraz(rexp);
    end;
    opMul: begin
      ops := '×';
      if (lexp.sign = opAdd) or (lexp.sign = opSub)
      then s1 := '('+PisiIzraz(lexp)+')'
      else s1 :=     PisiIzraz(lexp);
      if (rexp.sign = opAdd) or (rexp.sign = opSub)
      then s2 := '('+PisiIzraz(rexp)+')'
      else s2 :=     PisiIzraz(rexp);
    end;
    opDiv: begin
      ops := ':';
      if (lexp.sign = opAdd) or (lexp.sign = opSub)
      then s1 := '('+PisiIzraz(lexp)+')'
      else s1 :=     PisiIzraz(lexp);
      if (rexp.sign <> opNone)
      then s2 := '('+PisiIzraz(rexp)+')'
      else s2 :=     PisiIzraz(rexp);
    end;
  end;
  result := s1+ops+s2;
end;
```

5. ЗАКЉУЧАК. РЕЗУЛТАТИ

Два проблема која су се решавала током израде програма су:

- Како израчунати вредност израза који корисник састави у програму; и
- Како добити најбоље решење, односно од свих израза које је могуће саставити, онај чија је вредност најближа могућа тачном броју?

Први проблем је било могуће решити на неколико начина, а изабрани начин је тај, да се цео израз претходно преведе у *постфиксну нотацију*, у коме га је веома лако израчунати. Постојали су бржи начини за рачунање вредности израза, који не би два пута пролазили кроз цео израз; међутим пошто сви изрази у програму ионако имају највише 6 бројева, било који метод би радио брзо те није било битно убрзати рачунање резултата.

Други проблем је био много тежи и он представља главну сврху израде програма. Овај задатак већ није лак и захтева много рачунања и испробавања како би се дошло до одговора. Управо због временске захтевности, циљ је био да се осмисли метод који би, за што краће време, пронашао неко најбоље решење.

За тражење најбољег решења постојало је много начина, а ови су званично тестирани:

- Класични brute-force search алгоритам, без икаквих оптимизација („heuristics“); био је рекурзиван и састављао је све могуће изразе од 6 бројева и операција, директно у постфиксној нотацији и затим рачунао њихове вредности. Време рада на рачунару коришћеном за тестирање било је веома кратко у доста лакших случајева (мање од пола секунде), међутим у неким случајевима је то трајало и по неколико секунди, док је у најгорим ситуацијама (када није могао да се добије тачан број ни на један могући начин) програм радио и до једног минута.
- Након brute-force алгоритма, у кога није ни убацивано много оптимизација (јер је уочено да ни оне неће довољно убрзати програм), испробан је нерекурзивни начин који је, користећи већ добијене резултате, састављао изразе са по једним више бројем, почевши од свих комбинација са по 2 броја; овај алгоритам је првобитно тестиран без икаквих убачених оптимизација, како би се упоредила његова ефикасност у односу на brute-force search. Резултати су били задовољавајући – уочено је да је овај алгоритам и до петнаест пута бржи него brute-force.
- Прва оптимизација убачена у нови, нерекурзивни алгоритам била је провера комутативности приликом сабирања и множења. Уз ову оптимизацију програму је требало свега нешто мање од две секунде да нађе све комбинације. Укупан број израчунатих израза, тј извршених математичких операција, процењује се на око милион и по.
- Следеће оптимизације су биле: избегавање множења и дељења са 1, и игнорисање свих израза који дају резултат 0. Ово последње је постигнуто тако

што се одузимање није ни извршавало у случају да су обе стране израза једнаке. Време рада је смањено на око једну секунду.

- Уследиле су још неке оптимизације: на пример, не одузимати збир и не делити производом. Еквивалент је два пута заредом извршити одузимање, односно дељење. Такође, уведена је и провера асоцијативнонсти, као и дистрибутивности множења (и дељења) према сабирању и одузимању. Време рада смањено скоро 4 пута.

Још неки начини и варијације алгоритма су постојале за тражење најбољег решења, но они нису тестирани. Следи табела са коначним резултатима на неким тест-примерима.

Тест пример	**Једно решење**	**Brute-force**	**Нови алгоритам**	**Без „разломака“**
200 од: 9, 4, 3, 1, 15, 75	(9-1)×75:3	6ms 4286 израза	2ms 6030 операција	< 1ms 3148 операција
551 од: 2, 2, 1, 3, 15, 25	(2+2+3+15)×25+1	30ms 24302 израза	29ms 107217 операција	6ms 32011 операција
999 од: 2, 2, 2, 2, 10, 25	2×2×10×25-2:2	71ms 52910 из.	90ms 377207 оп.	26ms 133785 оп.
963 од: 1, 2, 4, 8, 10, 25	**962** = (25-1)×4×10+2	~ 43.7 секунди 33665406 из.	79ms 306605 оп.	13ms 70945 оп.
434 од: 4, 9, 5, 4, 25, 75	25×5×4+9-75	168ms 131718 из.	13ms 59520 оп.	4ms 21800 оп.
889 од: 5, 6, 5, 2, 25, 50	(25-5-2)×50-6-5	~ 30.7 секунди 22588458 из.	50ms 194462 оп.	12ms 66319 оп.

Табела 1: Времена извршавања алгоритама за тражење најбољег решења

Неки резултати се не могу добити ако се дели само оно што је дељиво без остатка; уколико би се користили и разломци (3/4; 1/5 итд) могло би се добити много више резултата. Замислимо, на пример, ситуацију у којој нешто делимо са 3, затим сабирамо са целим бројем, а онда све то множимо са 6 – коначан резултат би био цео број, међутим тај резултат можда не би могао да се добије на други начин... Због тога је понекад неопходно коришћење разломака. Програм узима у обзир и такве изразе, мада то може да се искључи у програму. Када се „разломци“ не би користили, време извршавања би било још краће, као што се види у табели.

Такође, може се приметити да „тежи“ тест примери не повећавају време извршавања код свих алгоритама – brute-force је у једном примеру спорији него у другом, док је са другим алгоритмом обрнут случај!

Резултати показују да је избегавање рекурзије и памћење резултата много ефикасније него brute-force. При томе је додатно убрзање постигнуто разним оптимизацијама, а највише провером асоцијативности и комутативности – које избацују око 5/6 могућих комбинација у свакој итерацији, што чини укупно убрзање од око 20 пута. Постигнута убрзања су значајна и коначан алгоритам ради око 300 пута брже него првобитни brute-force search.

Програм може да се користи као игра, где се бројеви случајно окрећу, један по један, до клика на „СТОП“, баш као и у квизу; има могућност састављања израза и рачунања његове вредности у сврхе провере резултата играча; међутим, највећи значај овог програма је тај што могу да му се задају бројеви по жељи (на пример, проблеми из самог квиза „ТВ слагалица“) и да се сазна како је могао да се добије тачан број, или који је најближи могући резултат у случају да не може.

Такође, изворни код програма може бити од великог научног значаја и може имати велику примену у проблемима сличног типа. Један пример проблема где би се овај код могао користити (уз мале измене) је проблем добијања одређеног броја коришћењем што мање јединица, или неких других бројева. Такође, проблем „Мој број“ се може лако проширити да ради са произвољним бројем бројева – а не само са 6... Ово су само неке примене, а код се сигурно може употребити за многе друге корисне ствари.

6. ЛИТЕРАТУРА

[1] Brown, Bob (2001): *Postfix Notation Mini-Lecture*. Документ пронађен на Интернету: http://www.spsu.edu/cs/faculty/bbrown/web_lectures/postfix/

[2] *Brute-Force Search*. The Heuristic Wiki: http://greenlightwiki.com/heuristic/Brute-Force_Search

[3] Chang, Shi-Kuo (2003): *Data structures and algorithms*. World Scientific