**Алгоритам за сортирање**

**Quick Sort (Брзо сортирање)**



Предметни професор: Студент:

Алекса Срданов Милош Стојановић бр.инд. 281005

**Садржај**

**Увод ………………………………………………………………………………………...3**

**Обрасци коришћења меморије и сортирање индекса .........................................……3**

**Класификација....................................................................................................................4**

**Брзо сортирање....................................................................................................................5**

**Увод........................................................................................................................................5**

**Devide and Conqure (Раздвоји и освоји)...........................................................................5**

**Преднсоти..............................................................................................................................5**

**Имплементација...................................................................................................................6**

**Избор ПИВОТ-a....................................................................................................................7**

**Поновљени елементи...........................................................................................................7**

**Паралелизација.....................................................................................................................8**

**Анализа...................................................................................................................................8**

**Пример...................................................................................................................................10**

**Пример псеудо кода.............................................................................................................11**

**Увод**

Не постоји савршен метода сортирања, свака од њих има своје предности и мане, тако да дају боље резултате од других на одређеним подацима и уређајима.

Не постоји “најбољи ” начин за сортирње, постоје много “најбољух“ начина у зависности од тога шта желимо да сортирамо и у ком окружењу.Битно је познавати карактеристике сваке методе како би наш избор био оптималан.

Алгоритме за сортирање представља алгоритам који пoставља елементе неке листе по одређеном реду.Ефикасно сортирање је битно за оптимизацију алгоритама који захтевају да унос буде у сортираној листи,често се корсти за добијање резултата који треба да буде читљив за човека.Шта више резултат мора задовољити два услова:

* Резултат је у не опадајућем редоследу.
* Резултат представља пермутацију уноса.

Од пoстојања рачунарства, проблем сортирања је привукао многа истраживања, вероватно због сложености ефикасног решавања упркос једноставном захтеву.

**Обрасци коришћења меморије и сортирање индекса**

Кад величина низа који треба сортирати достигне или премаши доступну предвиђену меморију, тако да диск(спорија меморија) или нека друга спорија меморају буду упошљене, образац за коришћење меморије алгоритма за сортирање постаје веома битан, и алгоритам који бе вероватно био довољно ефикасан док је низ стајао у RAM постаје непрактичан.У оваквом сценарију укупан број поређења постаје релативно мање битан, и број потребних копирања сектора меморије на диск или са њега постаје доминантан у исказивању перформанси алгоритма.Тако да број пролаза и локализација поређења постају битнији од самог броја поређења, од како се поређење приближних елемената извршава на брзини системске магистрале или са кеширањем чак брзином процесора, што је у поређењу са брзином диска, тренутна.

На пример, Quicksort алогритам обезбеђује разумне перформансе са адекватним RAM-ом, али због рекурзивног копирања делова низа ,алгоритам постаје све мање практичан кад низ више не стаје у RAM меморију.У том случају би можда прикладнији био неки други алгоритам, иако би захтевао више поређења.

Један од начина решавања овог проблема, што је веома корисно код сложених записа али са малим бројем елемената, је да се креира индекс у низу, и онда сортирати индекс, а не цео низ.Индекс је много мањи од целог низа и лако стаје у меморију.Ова метода се назива „tag sort“.

**Класификација**

Алгоритми за сортирање се обично класификују по:

* **Рачунарска сложеност** (најгору, средњу, најбољу) поређења елемената под условима величине листе O( n ).За уобичајене серијске алгоритме за сортирање добро понашање је O(n log n),за паралелне алгоритме је O(log2 n),где је лоше понашање O(n2).Њбоље понашање за серијско сортирање је O( n ),али то није могуће у о просечним случајевима.Оптимално паралелно сортирање је O(log n).
* **Рачинарска сложеност размене.**
* **Коришћење меморије** ( и осталих рачунарских ресурса).Постоје алгоритми за сортирање који су „in place“ .Стриктно „in place“ алгоритму је потребно само O(1) меморије изван елемената који се сортирају, понекад O(log n) додатне меморије се сматра „in place“.
* **Рекурзија**. Алгоритми могу бити рекурзивни или не, а неки оба(Merge sort).
* **Стабилност.**Стабилни алгоритми одржавају релативни ред записа са истим кључем.
* **Сортирање поређењем**. Сортирање поређивањем испитује податке само поређивањем два елемента оператором.
* **Опште методе** : уметање, размена, селекција, спајање итд.

**Брзо сортиранје**

**Историја**

Алгоритам брзог сортирања (Quicksort) је 1960. године развио Тонy Хоаре на државном универзитету у Москви .Брзо сортирање је добило широку примену, користило се у Unix-у као подразумевана библиотека функције ,као и у програмском језуку C и Java.

Брзо сортирање је касније, обимно анализирао Robert Sedgewick и извршио одређене оптимизације и побољшања.

Robert Sedgewick-ова је докторска дисертација 1975. године, сматра се прекретницом у истраживању Quicksort-а где је решено више отворених проблема који се односе на анализу различитих шема за избор пивота, укључујући Samplesort, прилагодљиво партиционисање по Ван Емдену, као и извођење очекиваног броја поређења и размена.

Bentley и McIlroy су уградити разна побољшања за употребу у програмским библиотекама, укључујући и технике које се баве једнаким елементима и шему за одабир пивота познатом као псеудомедиан од девет, где је узорак од девет елемената подељен у групе од по три елемента, а затим се бира средња вредност од три средње вредности датих подгрупа.

Jon Bentley је описао још једну једноставнију и компактију шему за дељење у својој књизи Programming Pearls коју приписују Nico Lomuto.

У 2009. години, Vladimir Yaroslavskiy предложио је нову примену Quicksort са двоструким пивотом.У Јава библиотекама, покренуо је дискусију у којој тврди је његов нови алгоритам супериоран у односу на методу за сортирање runtime библиотеке, који је у то време била базирана на широко коришћене и пажљиво подешене варијанте класичног Quicksort по Bentley и McIlroy.Yaroslavski-јев квиксорт је изабран као нови подразумевани алгоритам за сортирање у Oracle-овим Java 7 библиоекама.

**Увод**

Брзо сортирање спада у алгоритме који раде по принципу раздвоји и освоји (Devide and Conqure).

**Devide and Conqure (Раздвоји и освоји)**

**Историја**

Рани примери ових алгоритама су, пре свега, decrease and conquer (смањење и освоји) - оригинални проблем је сукцесивно смањен на појединачне манје проблеме, где заиста може да се реши итеративно.

Devide and Conqure је дизајн алгоритма који је засниван на разгранатости рекурзије.DC алгоритам ради тако што рекурзивно разлама проблем на два мања проблема истог или сличног типа, док они не постану довољно једноставни да би се директно решили.Онда се солуције за проблеме комбинују и дају решење првобитног проблема.

**Преднсоти**

**Ефикасност алгоритма**: пример често помаже у откривању ефикасних алгоритама.Био је кључ у алгоритмима Quicksort и Mergesort.У оба примера је DC прилаз водио и до побољшања по асимптотичну цену ове могућности.На пример ако основни случај има константу величину,посао растављања и комбиновања парциалних солуција је пропорцијалан величини проблема *n*, и постоји број подпроблема *p* ,где у свакој фази постоје ~*n*/*p*, тиме ће цена алгоритма бити O( n log n).

**Располагање меморијом:** DC алгоритми природно теже ка ефикасном коришћењу кеш меморије.Разлог је то што кад једном подпроблем постане довољно мали,он и сви негови подпроблеми могу се и решити унутар кеша,без приступања некој споријој меморији.Алгоритам који је дизајниран да на овакав начин приступа кеш меморији се назива „кеш-несвестан“, разлог је то што не садржи одређен параметар о величини кеша.Шта више DC алгоритми могу бити дизајнирани тако да на оптимални начин користе кеш у асимптотичком смилсу, без обзира на величину кеша.

**Roundoff control :** У рачунању са кружном аритметиком,на пример са floating point бројевима,DC алгоритам може донети прецизније резултате од неке површне еквивалентне итерактивне методе.

**Имплементација**

**Рекурзија:** DC алгоритми су природно имплементирани као рекурзивни поступци.У том случају,подпроблеми који воде до проблема који се тренутно решава се аутоматски чувају у STACK-у.

**Експлицитан стек :** DC алгоритми се такође могу имплементирати у не рекурзивне програме,који подпроблеме чувају у експлицитним скуповима података ,као што је STACK.Такав приступ допушта више слободе у избору подпроблема који ће се следећи решити, својство које је битно у неким аплиакцијама.Овај приступ је такође стандардна солуција у програмским језицима који не подржавају принципе рекурзивних поступка.

**Величина STACK-а :** У рекурзивним имплементацијама DC алгоритма,мора се обезбедити довољно алоциране меморије за рекурзију стога,у супрутном операција може неуспети због прокорачења STACK-а.Нажалост DC алгоритми који су времеснки ефикасни, рекурзивно нису дубоки.На пример Quicksort алгоритам може бити тако имплементиран да никад не захтева више од О(log2 n) уклопљених позива рекурзије за сортирање *n* елемената.Прекорачење стека се некад тешко може избећи када се користе рекурзивне методе,од како многи компајлери претпостављају да је рекурзивни стек гранично подручије меморије,и неки алоцирају фиксну количину мемроије.Компајлери такође чивају више информација него што је стриктно потребно,као што су повратне андресе,непорменљиви параметри.Ризик прекорачења стека се може смањити смањењем броја параметара и интенрих променљивих.

**Избор основних случајеве:** У сваком рекурзивном алгоритму, постоји значајна слобода у избору основних случајева, мали подпроблеми да се решавају директно како би се прекинула рекурзија.

Биранје најмањих или најједноставнијх могућих основних случајева је елегантније и обично доводи до једноставнијих програма, јер има мање случајева које треба узети у обзир и њих је лакше решити. На пример, FFT (Брза Фуријеова Трансформација) алгоритам може да заустави рекурзију када је улаз један узорак, а алгоритам quicksort-та сортирање би могао зауставити када је улаз је празна листа, у оба примера постоји само један основни случај да се размотри, и не захтева обраду.

С друге стране, ефикасност се често поправља када се рекурзија заустави код релативно великог броја основних случајева, и ти случајеви се решавају без рекурзије, из чега поизилази хибридни алгоритам.Ова стратегије избегава сувише велики број рекурзивних позива који решавају врло посал или чак никаквог, и дозвољава коришћење специјализованих не-рекурзивних алгоритма,који су за дате случајеве ефикаснији од експлицитне рекурзије.

**Дељење поновљених проблема:** За неке проблеме разграната рекурзија може се свести на вредновању истог подпроблема више пута.У том случају може бтити од користи идентификовати и сачувати солуцију за те понављајуће проблеме.

Quicksort алгoритам прво раздваја огромну листу на две сличне подлисте:

-ниске елементе

-високе леменете

Quicksort онда може рекурзивно да сортира подлисту.Кораци за то су следећи:

1. Изабере се елемент из листе ни он се зове **ПИВОТ.**

2.Листа се преуређује тако да сви елементи чија је вредност мања од ПИВОТ-а се постављају пре њега,а сви елементи са већом вредности се постављају после њега (исте вреднсоти се могу поставити са било које стране).После раздељивања ће се пивот наћи на својој коначној позицији.

3.Рекурзивно примењивати пређашње кораке на подлисту са мањом вредношћу , и одвајањем на пдолисту елементата са већом вредношћу.

**Избор ПИВОТ-а**

У раним верзијама Quicksorta-а ,који је најдаљи лево,тaj би се бирао као пивот.Нажалост то проузрокује најгори случај понашања код већ сортираних низова.Проблем је био лако решен тако што је биран случајан (random) члан ,средњи члан или последњи члан.Ово правило је одговор код проблема већ сортираних низова .Бирање пивота је такође компликовано постојањем премашења интиџера (integer overflow) ,ако је граница индекса под низа који се сортира превише велика ,коришћење средњег индекса ће проузроковати премашење и обезбедити неприхватљив индекс пивот-а

**Поновљени елементи**

Овакав алгоритам сортирања даје лоше резултате за уносе који садрже више истих елемената.Када су елементи једнаки ,при свакој рекурзији лева страна је празна ,а десна страна буде смањена само за један елемент.Стога алгоритму је потребно квадратно време да сортира низ истих елемената.

Да би се решио овај проблем qуицксорт-а,уводи се алтернативно времеснки линеарно дељење,низ се дели на три групе,на мање од пивот-а,на елементе једнаке пивот-у и на веће од пивот-а.Вреднсоти једнаке пивот-у су већ сортиране,тако да остају да е рекурзивно соритају вреднсоти веће и мање од пивот-а.

**Паралелизација**

Qуицксорт-ова раздели и освоји формулација га чини покорним паралелизму.Корак раздељивања је постигнут коришћењем алгоритма који се назива паралелна префиx сума(параллел префиуx сум) да би се прорачунао индеx сваког члана у свом делу раздељеног низа.За дату величину низа н ,кораци раздељивања ће се извршити (н) за време од (лог н) и захтева (н) додатних места.После дељења низа ,два дела низа се могу рекурзивно сортирати паралелно.Претпоставимо идеалан избор пивот-а,qуицксорт ће сортирати низ он н чланова у (н лог н) корака за време од (лог2 н) и захтева н додатног простора.

Qуицксорт има исте недостатке као и слични алгоритми за сортирање(мерге сорт),сто компликује постизање ефикасне паралелизације.Дубина qуицксрот-овог раздели и освоји(Д&Ц) дрвета утиче на скалабилност алгоритма, и да дубина је у великој мери зависна од избора пивот-а.Сем тога ,веома је дешко паралелизовати корак раздељивања у месту „ин плаце“.

**Анализа**

**Просечан случај користећи дискретну вероватноћу**

Да би се сортирао низ он н елемената ,очекивано време решавања је (н лог н),са просечно н! пермутација н елемената са једнаком вероватноћом.У најнеуједнаценијем слуцају, свакоб пута када извршавамо раздељивање ми раздвајамо листу да два под низа са величинама 0 и н-1.То значи да сваки рекурзивни позив обрађује лситу мању за један од предходне.Стога моземо извршавати рекурзију н-1 пута док не достигнемо величину листе једнаку јединици.То значи да дрво позивања (цалл трее) представља линарни ланац од н-1 позива.Овде че и-ти елемент иамти (н-1) корака да изврши дељење,тако да је у том случају потребно (н2) времена.То је најгори случај.

У најуједначенијем случају,када вршимо раздељивање ми делимо листу на две под лсите приближно исте величине.То значи да сваки рекурзивни позив обрађује листу преполовљене величине.Стога можемо извршити само лог2 н позива пре него сто достигнемо листу величине 1.То значи да је дубина дрвета позива(цалл трее) једнака лог2 н .Сада је сваком нивоу позива потребно (н) времена све укупно(сваки позив има исти константни премашај(оверхеад),али од како има само 0(н) позива за сваки ниво ,ово се може подвести у 0(н) фактор ).Резултат тога је да ра решење тј. Сортирање низа је потребно 0(н лог н) времена.

У овом случају није потребно да низови буду потпуно изједначени,чак и ако пивот издели низове на 75% на јенду страну и 25% на другу ,дубина позива је и даље ограничена на лог4/3 н тако да је коначно време извршавања н лог н.

У просечном случају,ако пивот има ранг негде око 50% ,то је између 25% и 75% ,тада раздваја елементе са најмење 25% на једној и највисе 75% на другој страни.Кад би могли константно да бирамо пивот-а на гранци педесетог процента ,мнорали би највосе лог4/3 н пута пре ного што се достигне величина лсите од 1.

Када је унос случајан(рандом),нема гаранције да ће ранг пивот-а бити око педесетог процента.У сваком рекурзивном позиву ,пивот има случајан ранг у лсити.тако да је у пола случајева у граници педесетог процента.То је довољно добро.Qуицксорт-ова рекурзија ће се зауставити на дубини од 2лог4/3 н .

**Просторна сложеност**

Простор који ће утрошити(заузети) Qуицсорт зависи од верзије.У верзији „in place“ Quicksort-ова просторна сложенсот износи O(log n) ,чак и у најгорем случајау ,када се користе следеће стратегије:

• Користи се „in place“ раздељивање.Нестабилно раздељивање захтева 0(1) простора.

• Након раздељивања ,део са најмање елемената се први сортира и захтева највише О(log n) места.

Quicksort са „in place“ и нестабилним раздељивањем користи константну количину додатног простора пре него што се изврши икакав рекурзивни позив.Quicksort мора сачувати константну количину информација за сваки рекурзивни позив.Како најбољи случај извршава највише О(log n) позива ,тако користи и O(log n) простора.

**Пример**

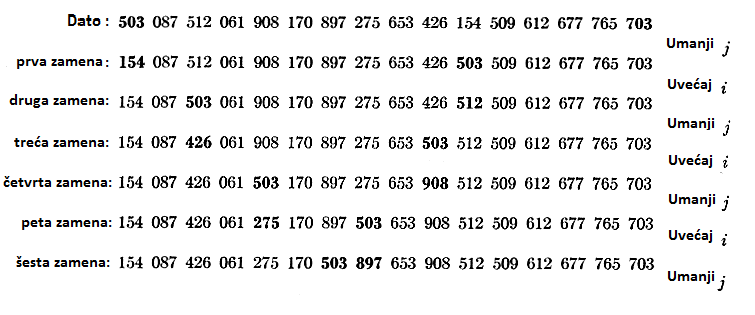
Суштинска идеја ове методе је да се узме један запис,рецимо Р1,и да се он премести на коначну позицију коју би требало заузети у сортираном фајлу, рецимо позиција С.Док се одређује ова коначна позиција,ми ће мо такође преуредити остале записе тако да неће бити ниједан са већим кључем од позиције С,и ни један са мањим кључем десно. Фајлови ће бити подељени на начин тако да првобитни проблем сортирања буде смањен на једноставније пробеме ,наиме да се сортирају Р1 ... Рс-1 и сортирати Рс+1 .. Рn.

Можемо применити исту технику на сваки од подфајлова појединачно док се посао не обави.Постоје више начина да се постигне такво дељење на леви и десни подфајл.

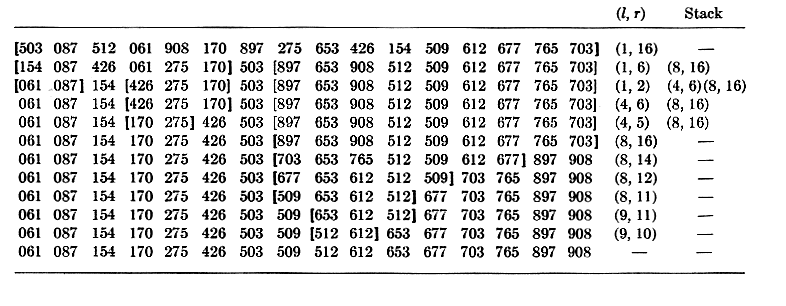
**-Sedwick:** Држати два показивача и са i=2 и ј=N почетно.Ако ће Рi бити део левог подфајла после дељења( то можемо утврдити упоређивањем Кi са К1),повећати i за 1,и наставити док се не сретне Рi који припада десном подфајлу.

Слично смањити j за 1 док се не сретне запис Рј који припада левом подфајлу.Ако је i<ј, замени Рi са Рј , затим наставити са процесом на следећим записима на исти начин ,док i >= ј.

Дељење ће коначно бити завршенои када се замене Рј и Р1

.

Следећа табела показује како се комплетно сортира овим приступом у 11 фаза.Заграде указују на подфајлове које је и даље потребно сортирати .Дупле заграде показују подфајлове важећег интереса .Унутар компјутера, важећи подфајл може бити предсављен граничним вредностима (l , р), и друге подфајлове представљене гомилом (STACK) додатних парова (lk , pk).Кад је неки фајл подељен на мање делове, ми стављамо дужи подфајл на гомилу(STACK) и почињемо посао на краћем подфајлу,док се не добију тривијално кратки фајлови.Ова метода гарантује да гомила(STACK) никад неће имати(садржати) више од logN уноса.



**Пример псеудо кода**

Једноставан пример примене куицксорт алгоритам на низ А ,где се сортирају елементи од и-тог до к-тог елемента.

quicksort(A, i, k):

if i < k:

p := partition(A, i, k)

quicksort(A, i, p - 1)

quicksort(A, p + 1, k)

Сортирање целог низа се извршава позивањем функције quicksort(А, 1, length(А)).Лево је индекс од елемента који се налази скроз на левој страни подниза.Десно је индеx који се налази скроз на десној страни подниза.Број елемената у низу је једнак десни индеx мање леви и плус један.

partition(array, left, right)

pivotIndex := choosePivot(array, left, right)

pivotValue := array[pivotIndex]

swap array[pivotIndex] and array[right]

storeIndex := left

for i from left to right - 1

if array[i] < pivotValue

swap array[i] and array[storeIndex]

storeIndex := storeIndex + 1

swap array[storeIndex] and array[right] //Коначна позиција пивот-а

return storeIndex

**Пример програма у коме се користи алогритам Quicksort-a**

На почетку је приказана извршна функција програма.И у њој се позива(корсити) класа PoccessHandler и њена статичка метода mainProccess().Ова класа ,тј. функција је задужена за покретање и ток операција које се обављају током извршавања и рада програма.

#include <iostream>

#include "QuicksortClass.h"

#include "DisplayHandler.h"

#include "Settings.h"

#include "ProccessHandler.h"

#include <array>

using namespace std;

int main(){

ProccessHandler::mainProccess();

return 0;

}

**Статичка метода** је метода класе и може се позивати и користии без инстанцирања класе тј. креиранја објекта.

**Опис и приказ класе ProccessHandler**

У ово класи је приказан глвани ток програма и све операције које се у њему извршавају.Следећи код приказује опис тј. декларацију класе(у C++ h фајл).

using namespace std;

class ProccessHandler

{

public:

static void mainProccess();

static vector<int> generateRandomVector(int &size);

ProccessHandler(void);

~ProccessHandler(void);

};

На почетку кода сеу позване библиотеке које су непходен за рад програма.Далје следи декларација public метода .Public методе се могу користити и изван класе,као поруке које се упућују објекту.Даље ће се приказати дефиницја свих метода класе.

using namespace std;

ProccessHandler::ProccessHandler(void){}

ProccessHandler::~ProccessHandler(void){}

void ProccessHandler::mainProccess(){

int array\_param = 0;

int array\_size = 0;

int delay = 0;

vector<int>my\_vec;

vector<int>def\_vec;

DisplayHandler::displayRequestInputType(array\_param);

if(array\_param == Settings::COSTUM\_ARRAY){

DisplayHandler::displayRequestArraySize(array\_size);

DisplayHandler::displayRequestDelay(delay);

my\_vec = DisplayHandler::displayRequestArrayInput(array\_size);

QuicksortClass quickSort = QuicksortClass();

quickSort.inputData(my\_vec);

quickSort.setArraySize(array\_size);

quickSort.setDelay(delay);

quickSort.sortData();

DisplayHandler::displayOnSortFinish();

}

if(array\_param == Settings::DEFAULT\_ARRAY){

QuicksortClass quickSort = QuicksortClass();

DisplayHandler::displayRequestArraySize(array\_size);

DisplayHandler::displayRequestDelay(delay)

quickSort.inputData(generateRandomVector(array\_size));

quickSort.setArraySize(array\_size);

quickSort.setDelay(delay);

quickSort.sortData();

DisplayHandler::displayOnSortFinish();

}

}

vector<int> ProccessHandler::generateRandomVector(int &size){

vector<int>temp;

for(int i=0; i < size; i ++){

int num = (rand() % 100);

temp.push\_back(num);

}

return temp;

}

На самом почетку ће се корисник питати да ли жели да крости неки случајан низ одређене величине,или да сам унесе жељени низ.На основу уноса(одговара)корисника, даље ће се извршавати програм.Тај упит се врши у следећој функцији.

void DisplayHandler::displayRequestInputType(int &param){

string temp = "0";

system("cls");

cout<<"------------------------------------------------"<<endl;

cout<<"| " << Settings::COSTUM\_ARRAY << " Uneti novi niz "<< Settings::DEFAULT\_ARRAY <<"

Koristi postojeci niz |"<<endl;

cout<<"------------------------------------------------ "<<endl;

cout<<"Uneti odgovor : ";

cin>>temp;

if(isInputOK(temp)){

switch (stoi(temp)){

case Settings::COSTUM\_ARRAY :

param = Settings::COSTUM\_ARRAY;

break;

case Settings::DEFAULT\_ARRAY:

param = Settings::DEFAULT\_ARRAY;

break;

default:

wrongAnswer();

displayRequestInputType(param);

break;

}

}else{ wrongInput();

displayRequestInputType(param);

Парамтери за унос су дефинисани у класи Settings која садржи статичке променљиве као параметре и подешавања програма.

class Settings{

public:

static const int DEFAULT\_ARRAY = 2;

static const int COSTUM\_ARRAY = 1;

static const int EXIT\_PROGRAM = 1;

static const int RESTART\_PROGRAM =2;

static const int MAX\_ELEMENTS = 3000;

static const int MSG\_DELAY = 2000;

static const int NO\_DELAY = 1;

static const int MEDIUM\_DELAY = 2;

static const int HIGH\_DELAY = 3;

static const int NO\_DEALY\_VALUE = 0;

static const int MEDIUM\_DELAY\_VALUE = 1000;

static const int HIGH\_DELAY\_VALUE = 2000;

Settings(void);

~Settings(void);

};

Такође се врши контрола уноса корисника ,тј. провера да ли је корисник унео погрешан знак или не постојећи параметар.Провера да ли је унос исправан се обављ у функцјији isInputOk(string& s).Ако је унос исправан функција враћа **true** у супротном **false.**

bool DisplayHandler::isInputOK(string& s){

string::const\_iterator it = s.begin();

while (it != s.end() && isdigit(\*it)) ++it;

return !s.empty() && it == s.end();

}

Ако је унос исправан програм наставља са извршавањем ,а у случају да није исправан унос ,исписаће се порука и затражити поновни унос.Функција за испис грешке:

void DisplayHandler::wrongInput(){

system("cls");

cout<<"Uneli ste pogresan znak,pokusajte ponovo"<<endl;

Sleep(Settings::MSG\_DELAY);

}

Након успешног одабира једне од могућности ,од корисника ће се захтевати унос величине низа који жели да сортира.

void DisplayHandler::displayRequestArraySize(int &size){

string str\_s = "0";

system("cls");

cout<<"-----------------------------------------------"<<endl;

cout<<"| Unesite broj elementata |"<<endl;

cout<<"-----------------------------------------------"<<endl;

cin>>str\_s;

if(isInputOK(str\_s)){

if(stoi(str\_s) < Settings::MAX\_ELEMENTS){

size = stoi(str\_s);

system("cls");

cout<<"------------------------------------------------"<<endl;

cout<<"| Broj usepsno unesen elementata (" <<size<< ")|"<<endl;

cout<<"------------------------------------------------"<<endl;

Sleep(2000);

}else{

maxSize();

displayRequestArraySize(size);

}

}else{

wrongInput();

displayRequestArraySize(size);

}

}

Када се унесе број елемената које корсиник жели да сортира,следи унос елемента ако је кросник изабрао ручни унос чланова.

vector<int> DisplayHandler::displayRequestArrayInput(int size){

string element;

vector<int>vec;

system("cls");

cout<<"----------------------------------------------"<<endl;

cout<<"| Unesite brojeve koje zelite da sortirate |"<<endl;

cout<<"-----------------------------------------------"<<endl;

cout<<endl;

for(int i=0; i < size; i++ ){

cout<<"Unesite element : ";

cin>>element;

while(!isInputOK(element)){

wrongInput();

displayElements(vec);

cout<<"Unesite element : ";

cin>>element;

}

vec.push\_back(stoi(element));

displayElements(vec);

}

return vec;

}

Ова функција тражи од корисника да унесе елементе један по један.Такође се врши контрола уноса.Ако је унос правилан,смештаће се у вектор целих бројева ,који функицуја враћа.Као параметар функције се прослеђује величина низа која је предходно била унесена.

У случају да је кросник одабрао коришћење случајног низа , извршиће се функција која враћа вектор целих случајних бројева.Случајни бројеви могу имати вредност он 0 до 100.За геберисање јендог случајног броја корсти се функција rand().Користећи ову функцију за одређени опсег (rand() % 100).

vector<int> ProccessHandler::generateRandomVector(int &size){

vector<int>temp;

for(int i=0; i < size; i ++){

int num = (rand() % 100);

temp.push\_back(num);

}

return temp;

}

Након уноса елемената за сортирање креираће се инстанца калсе QuicksortClass.Декларација кчасе:

using namespace std;

int const temp\_size = 10;

class QuicksortClass{

private:

vector<int> data;

int size;

int delay;

public:

QuicksortClass(void);

~QuicksortClass(void);

void inputData(vector<int>arr);

void setArraySize(int s);

void setDelay(int d);

void sortData();

long partition(vector<int> &input, int p, int r);

void quicksort(vector<int> &input, int p, int r);

void print(vector<int>input);

};

Пре покретања функције за сортирање низа морају се поставити параметри.Први параметар је величина низа који се сортира.Тај параметар је променљива size.Тој промељивој се додељује вредност функцијом setArraySize(int s).

void QuicksortClass::setArraySize(int s){

size = s;

}

Следећи параметар је кашњење тј. паузе између циклуса сортирања.Пауза се уноси да би приказ резултата сваког циклуса био што разумљивији кориснику.Тај параметар представља целобројна променљива delay.Функција за додељивање вредности променљивој:

void QuicksortClass::setDelay(int d){

delay = d;

}

Последњи параметар класе за сортирање је вектор(низ,листа) елемената који ће се сортирати.Параметар је представљен променљивом vector<int> data, и поставља се функцијом void inputData(vector<int>arr):

void QuicksortClass::inputData(vector<int>vec){

data = vec;

}

Након уношења свих параметара и података који су потртебни ,може се позвати функција за покретање сортирање података.

void QuicksortClass::sortData(){

cout<<"Sort started . . .";

Sleep(1000);

cout<<"Unos: ";

print(data);

quicksort(data, 0, size - 1);

cout << "Sortiran niz: ";

print(data);

}

У овој функцији се исписује поруке о почетку соритрања као и првобтини подаци тј. првобитни облик низа елемената који се требају сортирати,то се извршава функцијом void print(vector<int>input). Након тога се позива сама функција за сортирање .

void QuicksortClass::print(vector<int>input)

{

for ( int i = 0; i < size; i++ )

cout << input.at(i) << " ";

cout << endl;

}

void QuicksortClass::quicksort(vector<int> &input, int p, int r)

{

if ( p < r )

{

Sleep(delay);

int j = partition(input, p, r);

quicksort(input, p, j-1);

print(input);

quicksort(input, j+1, r);

Sleep(delay);

print(input);

}

}