**Алгоритми за брзо сортирање**

**Quick Sort**



Предметни професор: Студент:

Алекса Срданов Милош Стојановић бр.инд. 281005

**Садржај**

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

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

Класификација …………………………………………………………………………………..4

Стабилност ………………………………………………………………………………………4

Брзо сортирање..…...…………………………………………………………………………….5 Увод ………………………………………………………………………………………………5

Devide and Conqure (Раздвоји и освоји) ………………………………………………………..6

Предности ………………………………………………………………………………………..6

Имплементација …………………………………………………………………………………6

Избор ПИВОТ-a …………………………………………………………………………………8

Поновљени елементи ……………………………………………………………………………9

Паралелизација …………………………………………………………………………………10

Анализа …………………………………………………………………………………………10

Пример ………………………………………………………………………………………….12

Пример псеудо кода ……………………………………………………………………………14

Пример рада алгоритма ………………………………………………………………………..15

Варијанте Quicksort-a ………………………………………………………………………….18

Веза са другим алгоритмима ………………………………………………………………….19

**Увод**

Не постоји савршен метод сортирања. Свака од њих има своје предности и мане. Тако, неке дају боље резултате од других на одређеним подацима и уређајима док друге то показују са неким другим подацима и на другим уређајима.

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

Метода за сортирање представља алгоритам који пoставља елементе неке листе по одређеном реду. Најкоришћеније ређање је по нумеричком или ликсикогафском реду. Ефикасно сортирање је битно за оптимизацију алгоритма, јер примена често захтева да унос буде у сортираној листи, осим тога, очекује се да добијени резултат буде читљив за човека. Шта више, резултат мора задовољити два услова:

* Резултат је у унапред одређеном редоследу.
* Резултат представља пермутацију уноса.

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

Међу првим ауторима раних алгоритама за сортирање (1951.) нашла се Betty Holberton

ENIAC-у (Electronic Numerical Integrator and Computer) и UNIVAC-у. Bubble sort је анализиран још 1956. године. Алгоритми за сортирање упоређивањем имају основни захтев од O(nlogn) поређења. Алгоритми који се на заснивају на поређењу, као што је counting sort, могу имати и боље перформансе. И ако се проблем сортирања сматрао решеним (асимптотички оптимални алгоритми познати су од средине 20. века) и даље се јављају нови методи. Корисни алгоритми се непрекидно формирају. Алгоритми за сортирање преовладавају на часовима увода у рачинарске науке, где се упркос постојања мноштва алгоритама за овај проблем, по правилу, пружа само благо упознавање са основним концептима компјутерских алгоритама.

**Обрасци коришћења меморије и сортирање индекса**

Када величина низа који треба сортирати достигне или премаши доступну предвиђену меморију, тако да диск (спорија меморија) или нека друга спорија меморија буде упошљена, образац за коришћење меморије алгоритма за сортирање постаје веома битан, и алгоритам који би вероватно био довољно ефикасан, док је низ стајао у RAМ-у, 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. Поређење приближних елемената заснива се на брзини системске магистрале или са кеширањем, чак брзином процесора, што је у поређењу са брзином диска, тренутна брзина.

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, и ако 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 елементи низа замене индексима, и онда се сортирају индекси, а не цео низ. Индекси заузимају много мање простора од целог низа и требало би да лакше стаје у меморију. Ова метода се назива „tag sort“.

**Класификација**

Алгоритми за сортирање се обично класификују по следећим критеријумима:

* **Рачунарска сложеност** (најгору, средњу, најбољу) поређења елемената под условима величине листе O(n). За уобичајене серијске алгоритме за сортирање добро понашање је O(nlogn), за паралелне алгоритме је O(log2n),где је лоше понашање O(n2). Идеално понашање за серијско сортирање је O(n), али то није могуће у већини случајева. 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.

**Стабилност**

Када се врши сортирање некаквих података, само се део података испитује када се одређује редослед за сортирање. Подаци који се сортирају могу се представити као неки запис (ентитет) или као пар неких вредности, део тог податка, тј. вредност која се користи за сортирање назива се **кључ**. Алгоритам за сортирање је стабилан када се појаве два записа са истим кључем нпр. Р и С, и Р се појави пре С у оригиналном низу, тада се Р такође мора појавити пре С и у сортираном низу.

Када се елементи не могу разликовати, као што је случај са целим бројевима (integers), или генерализованије, било који податак где један цео елемент представља кључ, стабилност не предстаља проблем. Стабилност такође није проблем када су сви кључеви различити.

Нестабилни алгоритми за сортирање се могу посебно имплементирати на начин на који би били стабилни. Један од начина је да се вештачки прошири упоређивање, тако да се упоређивања два објекта са једнаким кључевима одлучује на основу провобитног редоследа.

**Брзо сортирање**

**Историја**

Алгоритам брзог сортирања (Quicksort) је 1960. године развио Тонy Хоаре на државном универзитету у Москви. У то доба Хоаре је радио на пројекту машинског превода за Националну лабораторију физике. Као део процеса превођења, он је морао да сортира речи руских реченица пре него што их потражи у руско-енглеском речнику који је већ сортиран по абецедном реду на магнетној траци. Након што је препознао да ће његова прва идеја сортирања бити спора, брзо је дошао до нове идеје која је преточена у Quicksort.

Написао је програм у Mercury Autocode за партицију, али није могао да напише програм да би одговорио на листу несортираних сегмената. По повратку у Енглеску, од њега је затражено да напише код за Shellsort као део новог посла. Где је представио бржи алгоритам на коме је радио.

Брзо сортирање је добило широку примену, користило се у Unix-у као подразумевана библиотека функција, 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 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 уградили разна побољшања за употребу у програмским библиотекама, укљућујући и технике које се баве једнаким елементима и шему за одабир пивота познатим 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 компактнију шему за дељење у својој књизи Programming Pearls.

У 2009. години, Vladimir Yaroslavskiy предложио је нову примену Quicksort-а са двоструким пивотом. У ЈАVА библиотекама, покренуо је дискусију у којој је тврдио да је његов нови алгоритам супериоран у односу на методу за сортирање “runtime” библиотеке, која је у то време била базирана на широко коришћеној и пажљиво подешеној варијанти класичног Quicksort-а по Bentley и McIlroy. Yaroslavski-јев Quicksort је изабран као нови подразумевани алгоритам за сортирање у Oracle-овим Java 7 библиотекама.

**Увод у Брзо Сортирање**

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 (умањи и освоји) - оригинални проблем је сукцесивно смањен на појединачне мање проблеме, где заиста може да се реши и итеративно.

Devide and Conqure је дизајн алгоритма који је засниван на разгранатости рекурзије. Devide and Conqure 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 различите солуције проблема комбинују и дају решење првобитног проблема.

**Предности**

**Ефикасност алгоритма**: Пример често помаже у објашњењу ефикасних алгоритама. Пример је био и кључ у алгоритмима Quicksort и Mergesort. У оба примера је прилаз Devide and Conqure водио и до побољшања и до асимптотске процене ове методе. На пример, ако основни случај има константу величину, посао растављања и комбиновања парцијалних солуција је пропорционалан величини проблема *n*, и постоји број подпроблема *p*, где у свакој фази постоје ~*n*/*p*, тиме ће брзина алгоритма бити O(nlogn).

**Располагање меморијом:** Devide and Conqure 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 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, Devide and Conqure алгоритми могу бити дизајнирани тако да на оптимални начин користе кеш у асимптотичком смислу, без обзира на величину кеша.

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

**Имплементација**

**Stack:** Стек у рачунарској архитектуи представља област меморије где се подаци дадају или уклањају на начин LIFO (Last in, first out). У већини савремених компјутерских система, свака нит (thread) има резервисани регион у мемеорији који се назива њиховим стеком. Када се функција изврши, може додати неке од своjих података о стању на врху стека, када функција излази, она је одговорна за уклањање тих података из стека. У најмању руку, стек неке нити се користи за чување локације позива функције да би се повратни извод вратио на исправну локацију. Програмери и даље могу експлицитно користити стек. Ако нека меморијска локација лежи на стеку,каже се да је “алоцирана на стеку“. Пошто се меморија на стеку додаје и уклања по LIFO принципу, алоцирање меморије по стеку је једноставније и брже од алоцирање меморије на хипу. Меморија се аутоматски и веома ефикасно уклања са стека када се нека функција заврши, што може бити од користи ако ти подаци програмеру више нису потребни. Додељени стек некој нити може бити и реда неколико бајтова на неким процесорима. Алоцирати више меморије него што је доступно води до премашења стека и прекида извршења програма. Неке породице процесора као x86, имају посебна упутства за манипулацију са стеком зa тренутно извршавајуће нити. Друге породице процесора укључујући PowerPC и MIPS, немају експлицитну подршку за стек, већ се ослањају на конвенцију и делегирају управљање стеком бинарном интерфејсу (ABI) апликације оперативног система.

**Рекурзија:** Devide and Conqure 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 :** Devide and Conqure алгоритми се такође могу имплементирати у не рекурзивне алгоритме, који подпроблеме чувају у експлицитним скуповима података, као што је STACK. Такав приступ допушта више слободе у избору подпроблема који ће се следећи решити. Ово својство је битно у неким апликацијама. Овај приступ је такође стандардно решење у програмским језицима који не подржавају принципе рекурзивних поступка.

**Величина STACK-а:** У рекурзивним имплементацијама Devide and Conqure алгоритма, мора се обезбедити довољно алоциране меморије за рекурзију стека, у супрутном операција може „не успети“ због прокорачења STACK-а. На жалост, Devide and Conqure алгоритми који су временски 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. Многи компајлери претпостављају да је рекурзивни стек гранично подручије меморије, и неки алоцирају фиксну количину меморије. Компајлери такође чивају више информација него што је стриктно потребно, као што су повратне адресе, непроменљиви параметри. Ризик прекорачења стека се може смањити смањењем броја параметара и интерних променљивих.

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

Бирање најмањих или најједноставнијих 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 би сортирање могао зауставити када је улаз 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 произилази хибридни алгоритам. Ова стратегије избегава сувише велики број рекурзивних позива који решавају врло мало послa 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 вредновање истог подпроблема више пута. У том случају може бити од користи идентификовање и чување решења за те понављајуће проблеме.

Quicksort алгoритам прво раздваја огромну листу на две сличне подлисте:

-ниске елементе

-високе елементе

На тај начин Quicksort може рекурзивно да сортира подлисту. Кораци за то су следећи:

1. Изабере се елемент из листе и он се назове **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 вредношћу се постављају после њега (исте вредности се могу поставити са било које стране). После раздељивања ће се пивот наћи на својој коначној позицији.

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

**Избор ПИВОТ-а**

У раним верзијама Quicksorta-а, који је најдаљи лево, тaj би се бирао као пивот. На жалост, 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 опсега целих бројева (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. С тога алгоритму је потребно квадратно време да сортира низ истих елемената.

**Kвадратно време:** У рачунарским наукама, временска комплексност неког алгоритма квантификује тј. одређује количину времена утрошену за извршавање као функције у зависности од дижине уноса тј. улазног параметра.

Временска комплексност се обично одређује бројањем основних операција изршених од стране алгоритма, где је за једну основну (елементарну) операцију потребана непроменљива количина времена да се изврши. Како се перформансе алгоритма могу мењати у зависности од раличитог улаза исте величине, алгоритам обично користи временску сложеност најгорег случаја означеног као Т(n), и дефинисана је као максимални износ времена утрошеног на било који улаз величине n.

Обично се ређе, а поготово не екплицитно, наводи вредност просечног случаја временске комплексности неке методе.

Да би се решио овај проблем Quicksort-а, уводи се алтернативно временски линеарно дељење, низ се дели на три групе, на мање од пивот-а, на елементе једнаке пивот-у и на веће од пивот-а. Вредности једнаке пивот-у су већ сортиране, тако да остају да се рекурзивно соритају вредности веће и мање од пивот-а.

Псеудокод је приказан у следећем примеру:

algorithm quicksort(A, lo, hi) is

if lo < hi then

p := pivot(A, lo, hi)

left, right := partition(A, p, lo, hi)

quicksort(A, lo, left)

quicksort(A, right, hi)

Најбољи случај се јавља када су сви елементи једнаки. У овом случају модификовани алгоритам ће извршити највише два рекурзивна позива на празне поднизове и њих ће завршити временски линеарно.

**Оптимизације**

Постоје још две веома битне оптимизације, такође преложене од стране Sedgewick-а и имају широку примену.

* Осигурати да највише O(logn) простора буде искоришћено, што значи обрадити прво мањи део подељеног низа, а затим финално прећи на другу половину.

**Паралелизација**

Quicksort-ова идеја раздели и освоји чини га покорним паралелизму. Корак раздељивања је постигнут коришћењем алгоритма који се назива паралелна префикс сума (paralel prefix sum) да би се прорачунао индекс сваког члана у свом делу раздељеног низа. За дату величину низа n, кораци раздељивања ће се извршити (n) за време од (logn) и захтева (n) додатних места. После дељења низа, два дела низа се могу рекурзивно сортирати паралелно. При претпоставци о идеаланом избору пивот-а, Quicksort ће сортирати низ од n чланова у (nlogn) корака за време од (log2n) и захтева n додатног простора.

Quicksort има исте недостатке као и слични алгоритми за сортирање (меrge sort), што компликује постизање ефикасне паралелизације. Дубина Quicksort-овог раздели и освоји (D&C) дрвета утиче на скалабилност алгоритма, и да је дубина у великој мери зависна од избора пивот-а. Сем тога, веома је тешко паралелизовати корак раздељивања у месту „in place“.

**Анализа просечаног случаја коришћењем дискретне вероватноће**

Да би се сортирао низ од n елемената, очекивано време решавања је (nlogn), са просечно n! пермутација n елемената са једнаком вероватноћом. У најнеуједначенијем случају, сваког пута када извршавамо раздељивање ми раздвајамо листу да два под низа са величинама 0 и n-1. То значи да сваки рекурзивни позив обрађује листу мању за један од предходне. Стога можемо извршавати рекурзију n-1 пута док не достигнемо величину листе једнаку јединици. То значи да дрво позивања (call tree) представља линарни ланац од n-1 позива. Овде ће n-ти елемент имати (n-1) корака да изврши дељење, тако да је у том случају потребно (n2) времена.То је најгори случај.

У најуједначенијем случају, када вршимо раздељивање ми делимо листу на две под листе приближно исте величине. То значи да сваки рекурзивни позив обрађује листу преполовљене величине. Стога можемо извршити само log2n позива пре него што достигнемо листу величине 1. То значи да је дубина дрвета позива (call tree) једнака log2n.

Сада је сваком нивоу позива потребно (n) времена све укупно (сваки позив има исти константни премашај (оверхеад), али од како има само 0(n) позива за сваки ниво, ово се може подвести у 0(n) фактор). Резултат тога је да решење тј. да је за сортирање низа потребно 0(nlogn) времена.

У овом случају није потребно да низови буду потпуно изједначени, чак и ако пивот издели низове на 75% на једну страну и 25% на другу, дубина позива је и даље ограничена на log4/3n тако да је коначно време извршавања nlogn.

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 највише 75% на другој страни. Кадa би могли константно да бирамо пивот-а на гранци педесетог процента, морали би највише log4/3n пута пре нeго што се достигне величина листе од 1.

Када је унос случајан (random), нема гаранције да ће ранг пивот-а бити око педесетог процента. У сваком рекурзивном позиву, пивот има случајан ранг у листи. Тако да је у пола случајева у граници педесетог процента. То је довољно добро. Quicksort-ова рекурзија ће се зауставити на дубини од 2log4/3n.

**Просторна сложеност**

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

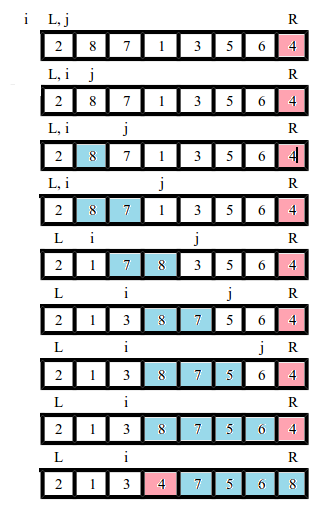
• Користи се „in place“ раздељивање. Нестабилно раздељивање захтева О(1) простора.

• 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(logn) простора.

**Пример**

Пример имплементације алгоритма рекурзијом. Приказан је на слици 1.1.



*Слика 1.1*

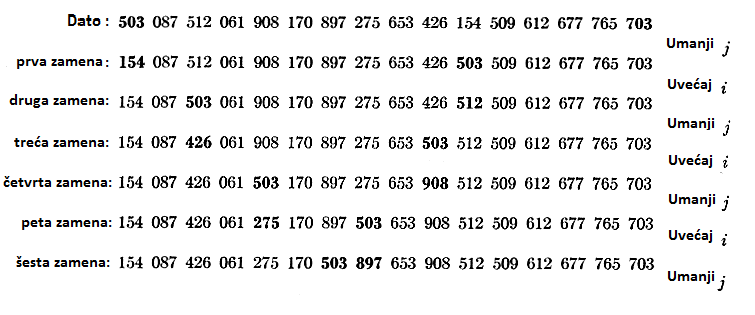
Суштинска идеја ове методе је да се узме један запис, рецимо Р1, и да се он премести на коначну позицију коју би требало заузети у сортираном фајлу, рецимо позиција С. Док се одређује ова коначна позиција, ми ћемо такође преуредити остале записе тако да неће бити ниједан са већим кључем од позиције С, и ни један са мањим кључем десно.

Фајлови ће бити подељени на начин тако да првобитни проблем сортирања буде смањен на једноставније проблеме, 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 на почетку. 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, све док је i >= ј.

Дељење ће коначно бити завршенои када се замене Рј и Р1.

*Слика 1.2*

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 представљен 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 Quicksort алгоритам на низ А, где се сортирају елементи од i-тог до k-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 индекс који се налази скроз на десној страни подниза. Број елемената у низу је једнак: десни индекс 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

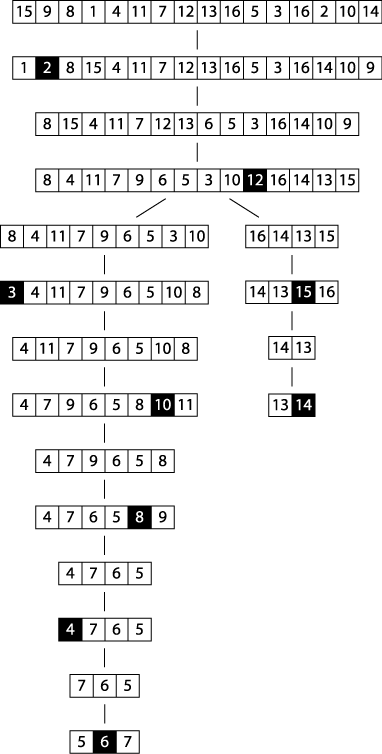
return storeIndex

**Хоаре шема за партиционисање**

Хоаре користи два индекса који почињу на крајевима низа који се дели, онда индекси иду један ка другоме док не наиђу на инверзију: пар елемената, један који је већи или једнак пивоту, а један мањи или једнак и који су у погрешном редоследу релативно један према другом. Ти елементи се онда инвертују. Када се ти индекси сретну, алгоритам се зауставља и враћа коначни индекс.

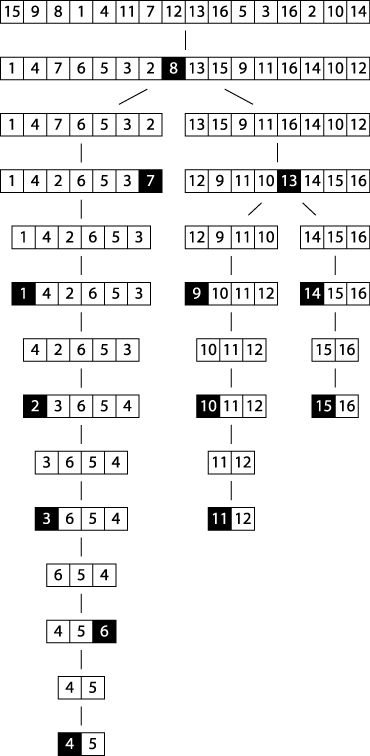
**Пример рада алгоритма**

На слици (слика 1.3) приказује Quicksort у акцији. Сваки од црних квадрата на слици представља одабрани пивот на првом месту партиције. Први одабрани пивот је је број “2“, где се испоставља да је то био веома лош избор од како је произвео два под низа са 1 и 14 елемената. Током следећег рекурзивног позива Quicksort-а на десном под низу, одабран је број “12“ као пивот, из чега произилазе два под низа са по 9 и 4 елемента, што представља бољи избор. Већ се може видети предност коришћења партционисања, од како су последња четири елемента низа највећи бројеви низа. Због природе случајног избора пивота, могућа су различита понашања и исходи.



*Слика 1.3*

У другачијем извршавању, приказаном на слици (слика 1.4), први изабрани пивот боље раздваја проблем у два упоредива задатка.

****

*Слика 1.4*

**Примери имплементације**

Имплементација приказана на слици 1.5 користи стандардно оптимизовани начин коју користи Insertion Sort када величина подниза које треба сортирати падне испод предодређене најмање величине.

/\*\*

\* Sort array ar[left,right] using Quicksort method.

\* The comparison function, cmp, is needed to properly compare elements.

\*/

void do\_qsort (void \*\*ar, int(\*cmp)(const void \*,const void \*),

int left, int right) {

int pivotIndex;

if (right <= left) { return; }

/\* partition \*/

pivotIndex = selectPivotIndex (ar, left, right);

pivotIndex = partition (ar, cmp, left, right, pivotIndex);

if (pivotIndex-1-left <= minSize) {

insertion (ar, cmp, left, pivotIndex-1);

} else {

do\_qsort (ar, cmp, left, pivotIndex-1);

}

if (right-pivotIndex-1 <= minSize) {

insertion (ar, cmp, pivotIndex+1, right);

} else {

do\_qsort (ar, cmp, pivotIndex+1, right);

}

}

/\*\* Qsort straight \*/

void sortPointers (void \*\*vals, int total\_elems,

int(\*cmp)(const void \*,const void \*)) {

do\_qsort (vals, cmp, 0, total\_elems-1);

}

*Слика 1.5*

Избор пивота је направљен помоћу екстерне методе selectPivotIndex(ar, left, right), која обезбеђује елемент низа од кога би се започело партиционисање.

**Варијанте Quicksort-a**

**Quicksort са вишеструким пивотом:** Овај начин имплементације, место дељања првобитног низа на два дела, низ дели на С поднизова, где би број С био произвољан и представњао један пивот. Док се овај метод узимао у обзир током седамдесетих година прошлог века, крајњи резултат оваквог алгоритма у пракси није био бржи од класичне имплементације. Процена из 1999. године, овог алгоритма са променљивим бројем пивота била је да је за његово извршавање потребно чак 20% више процесорских инструкција, али су симулације указивале на то да би се његова ефикасност повећала код великих уноса. Касније је утврђено да се перформансе овог алгоритма огледају у кеширању и експериментални резултати показују да се варијанта са три пивота може много ефикасније извршавати на савременијим машинама.

**Спољни** **Quicksort:** Ова имплементација функционише као и регуларна само што је пивот замењен бафером. Прво се у бафер учитају први и последњи елементи, и они се сортирају. Након тога се чита следећи елемент са почетка или краја да би се успоставила равнотежа уписивања. Ако је следећи елемент мањи од мањег лемента из бафера, онда се тај елемент уписује у слободан простор на почетку низа. Ако је већи од већег елемента у низу, онда се уписује на крају. У супротном се уписује најмањи или највећи из бафера и следећи елемент се смешта у бафер. Рекурзивно се сортира мањи део, и итерира кроз преостале делове.

**Quicksоrt sa trosmernom osnovom:** Овај алгоритам је комбинација радик и Quicksort-а. Изаберите елемент из низа (пивот) и анализирајте сваки члан низа на следећи начин. Све њих треба поделити у три гупе: оне чији је одговарајући карактер мањи, једнак и већи од карактера пивота-а. Рекурзивно сортирајте “мање од“ и “веће од“ групе на истом карактеру.

Рекурзивно сортирајте „једнако“ групу следећим знаком. С обзиром да користимо бајтове или речи дужине W битова, најбољи случај је О(КN), а најгори случај О(2kN), дат је за једниствене кључеве N<2k , где је К скривена константа у свим алгоритмима за сортирање са стандардним упоређивањем. Ово је на неки начин тросмерни Quicksort где је средња партиција представља сортиран подниз елемената који се једнаки пивоту.

**Брзо радикс сортирање:** Развијен је од стране Powers-a као О(К) паралелни ПРАМ алгоритам. Ово је опет комбинација радикс и Quicksort-a, али се Quickosrt-ова одлука за партиционисање леве и десне стране заснива на сукцесивним битовима кључа, и стога је О(КN) за N К-битних кључева. Треба имати на уму да сви алгоритми за сортирање путем упоређивања ефективно предпостаљају идеално К од О(logN), када је К мање можемо сортирати О(N) користећи хешну таблицу или сортирање целог броја, ако је К много веће од logN али елементи су јединствени унутар О(logN) бита, преостали битови неће бити разматрани.

**Веза са другим алгоритмима**

Quicksort је оптимизована верзија типа бинарног дрвета. Уместо стављања ставки секвенцијално у ескплицитно дрво, Quicksort их истовремено организује у дрво које подразумева рекурзивне позиве. Алгоритми чине иста упоређивања, али другим редоследом. Пожељна особина неког алгоритма за сортирање јесте стабилност, то је редослед елемената којим се постиже да се упоређивањем елемената њихов природан редослед није променио. Овај захтев омогућава контролу над редоследом више кључних табела. Ову особину је тешко задржати за алгоритам који је “in place“. За варијанте Quicksort-а који укључују додатну меморију, због коришћења показивача, тривијално је очувати стабилност. Сложеније или диск-везане структуре података имају тенденцију повећања времена и искоришћене меморије.

Директни конкурент Quicksort-а је Heapsort.

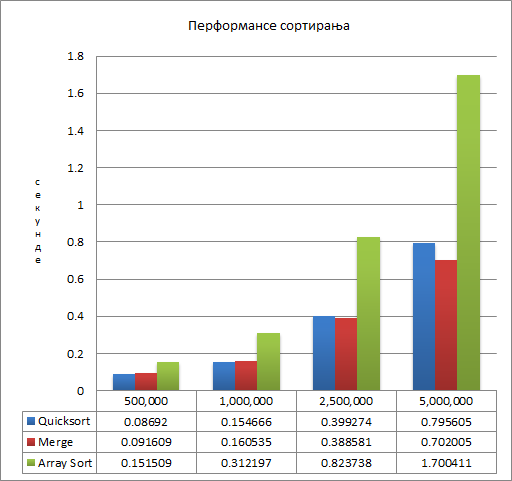
Време рада Heapsort је O(nlogn), али проселно време рада Heapsort-а обично се сматра споријим од Quicksort-а. Овај резултат је дискутабилан.

Introsort је варијанта Quicksort који се пребацује на Heapsort када се открије лош случај како би се избегао најгори сценарио. Quicksort се такође такмичи и са Mergesort-ом. Mergesort је стабилна врста алгоритма, за разлику од иницијалних Quicksort-а и Heapsort-а.

Може се лако прилагодити за рад са повезаним листама. И ако се Quicksort може имплементирати као стабилна врста помоћу повезаних листи, он ће увек трпети због лоших избора пивота без случајног приступа.

Главна мана Mergesort-а је та да када се ради над низом, ефикасне имплементације захтевају О(n) помоћног простора, где Quicksort захтева само малу константну количину тог простора.

На следећој слици (слика 1.6) је приказан дијаграм перформанси у односу на друге алгоритме.



*Слика 1.6*

**Literatura:**

1. <https://en.wikipedia.org/wiki/Quicksort>
2. <https://brilliant.org/wiki/sorting-algorithms/>
3. <https://en.wikipedia.org/wiki/Sorting_algorithm>
4. <https://www.safaribooksonline.com/library/view/algorithms-in-a/9780596516246/ch04s04.html>
5. http://www.geeksforgeeks.org/quick-sort/