Algoritmi sortiranja

U današnjem svetu konstrukcije raznovrsnih softverskih sistema, jedan od glavnih problema i zadataka sa kojima se susretne svaki programer jeste pretraživanje podataka. Imajući u vidu to i činjenicu da se pretraživanje podataka u više od 80 posto slučajeva vrši nad uređenim nizovima (izuzev kad se radi nad malim skupovima podataka), problem pretraživanja podataka se u potpunosti svodi na prethodno sortiranje, u najgeneralnijem slučaju, zadatog niza podataka. Sortiranje predstavlja proces preuređivanja skupa podataka po uređenom poretku. Matematički gledano, sortiranje niza u neopadajućem ili nerastućem poretku podrazumeva nalaženje jedne permutacije elemenata niza u kojoj se elementi pojavljuju u neopadajućem to jest nerastućem poretku. Zadatak ovog projekta jeste da pored priloženih demonstracija (simulacija) algoritama predoči i neke od njihovih osnovnih prednosti i mana kao a takođe i istakne vremensku i prostornu složenost za svaki od navedenih algoritama.

Razmatrani su sledeći algoritmi:

1) Selection sort (direktna selekcija)

2) Bubble sort (direktna zamena)

3) Radix exchange sort (pobitno razdvajanje)

4) Pigeonhole sort

5) Coutnig sort (sortiranje brojanjem)

6) Square (Friend) sort (selekcija-particionisanje)

7) Shaker (Cocktail) sort (direktna zamena)

8) Quick sort (zamena-particionisanje)

9) Merge sort (sortiranje spajanjem-varijanta direktnog spajanja)

Svi algoritmi sortiranja pripadaju jednoj od grupa metoda koje se zasnivaju na: umetanju, selekciji, zameni ili spajanju. Na početku analize svakog od navedenih algoritama ćemo naznačiti kojoj grupi prethodno navedene podele on pripada. Za osnovne indikatore performansi prostorne složenosti isticaćemo koliko dodatnog prostora (broj pomoćnih polja za tip integer) za pomoćne strukture je potrebno za algoritam koji se analizira. Sa druge strane za osnovne indikatore performansi vremenske složenosti (koje smatramo daleko važnijim od prostornih) uzimaćemo:

a) Broj koraka algoritma da bi se došlo do rešenja (za elementarni korak uzimamo jednu iteraciju while ili for petlje)

b) Broj poređenja ključeva

c) Broj premeštanja zapisa

1. *Selection sort* (direktna selekcija, metoda izbora)

Selection sort ili direktna zamena je glavni predstavnik metoda selekcijom. Iako je ovaj algoritam krajnje jednostavan za implementaciju on se ipak zasniva na programerski rečeno „gruboj sili“. Algoritam prolazi kroz dve ugneždene petlje i u prvom prolazu upoređuje prvi element sa svim ostalima i zamenjuje ih ukoliko se ne nalaze u odgovarajućem poretku (rastućem ili opadajućem).

Složenost ovog algoritma ako posmatramo broj zamena je uvek n-1 (ako uvedemo optimizaciju da se sa tekućim elementom spoljašnje petlje menja samo najmanji element preostalih nesortiranih elemenata) a sumiranjem broja koraka i poređenja ključeva dobijamo u oba slučaja da je složenost:

$$\sum_{i=0}^{n-1}(n-i) = n(n-1)/2 = O(n^2)$$

Dakle, složenost selection sorta je u najboljem, najgorem i prosečnom slučaju uvek:

$$\text{MIN} = O(n^2)$$

$$\text{MAX} = O(n^2)$$

$$\text{AVG} = O(n^2)$$

U kodu napisanom u jeziku C++ projekta *AlgortimiSortiranja* ovaj algoritam relaziovan je pozivanjem metode *sortiraj( )* klase *MetodaIzbora*. Postupan prikaz algoritma dat je na slici 1.

Slika 1

2. *Bubble sort* (direktna zamena)

Bubble sort je takođe jedan od najprostijih i najpopularnijih (za početnike) metoda sortiranja koji pripada grupi metoda zamene. Algoritam više puta sekvencijalno prolazi kroz niz i pritom upoređuje svaki element sa narednim u nizu, pa ako ova dva elementa ne zadovoljavaju u startu definisani poredak, zamene im se mesta. Na taj način će u prvom koraku najveći element sigurno postavi na svoje, u rastućem poretku poslednje mesto, a zatim se u svakom sledećem prolazu takođe po jedan element postavi na svoje finalno mesto i polako dobijamo sortirani niz.

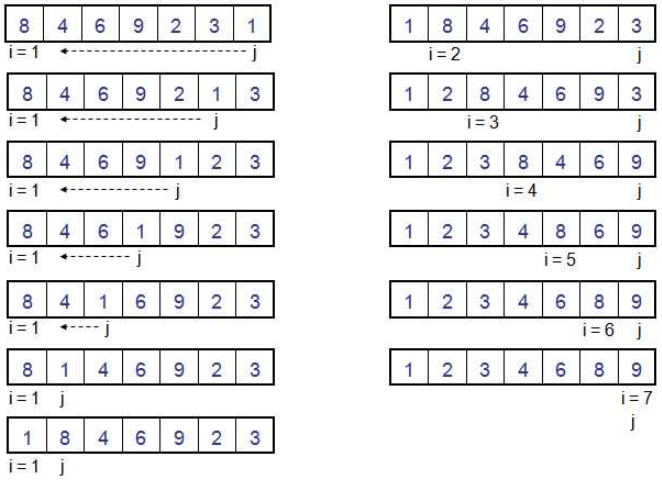
Najbolji slučaj metoda direktne zamene je kada je ulazni niz već sortiran. Tada se na osnovu specijalnog (bound) markera algoritam završava u jednom koraku dok je broj poređenja jednak 0. Najgori slučaj sa najvećom složenošću je kada je niz sortiran u obrnutom poretku od željenog. Tada je potrebno tačno n-1 koraka da svaki element dođe na svoje mesto i tačno *BrPor=n*(n-1)/2* poređenja i BrZam= *n*(n-1)/2*. Dakle, bilo u srednjem (kada izvršimo usrednjavanje po svim slučajevima) ili najgorem slučaju složenost će biti kvadratna. Odnosno:

$$\text{MIN}= O(n)$$

$$\text{MAX}= O(n^2)$$

$$\text{AVG}= O(n^2)$$

U kodu napisanom u jeziku C++ projekta *AlgortimiSortiranja* ovaj algoritam relaziovan je pozivanjem metode *sortiraj( )* klase *Bubble.* Postupan prikaz algoritma dat je na slici 2.



Slika 2

3. *Radix exchange sort* (pobitno razdvajanje)

Radix exchange sort je algoritam koji se samo u zavisnosti od svoje implementacije može svrstati u neku od 4 navedene grupe. U našem projektu napisanom u C++ radix exchange sort implementiran je nad jednim nizom podataka (in situ-na mestu se sortira) i sličan princip razdvajanja kao u Quick sortu, zbog toga ćemo našu implementaciju svrstati u metodu selekcije.

Sam algoritam zasniva se na tome što na osnovu binarne predstave brojeva, svi brojevi u zadatom nesortiranom nizu stavljaju na početak ili na kraj niza. Odluka o tome koji se brojevi stavljaju na početak a koji na kraj niza donosi se na osnovu posmatranog bita težine(u prvoj iteraciji najveće težine). Kada se posmatrani niz prilikom prve iteracije sortira po najvecem bitu, prelazi se na sledeći bit i takav postupak se ponavlja sve dok se ne izvrši i sortiranje po bitu najmanje težine, nakon čega dobijamo finalno sortirani niz.

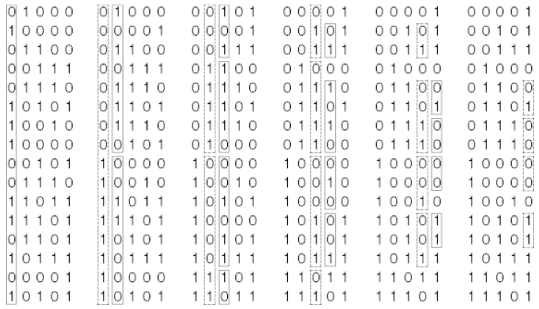
Gore navedeni postupak se implementira tako što idući sa leve strane i sa desne strane niza uporedno vršimo ispitivanje da li je posmatrani tekući bit 0 ili 1. Ako se sa leve strane nalazi bit koji je 1 a sa njegove desne strane bit koji je 0 ta dva broja zameniće mesta. Nakon toga postupak za sledeći niži bit se nastavlja u dve novostvorene particije koje počinju sa 0 ili sa 1.Neka je broj bitova u predstavi najvećeg broja u nizu koji se sortira m, a broj koraka polovljenja (u opštem najboljem slučaju) jednak $n/2^k$, što daje k=log(n). Pošto je ukupan broj elemenata u svim particijama u određenom koraku svakako manji od n gornja granica broja poređenja je O(kn)=O(n*log(n)). Očekivani broj zamena na istom nivou je približno n/6 tako da je ukupan broj zamena takođe reda O(n*log(n)). Prema tome u najboljem slučaju vremenska složenost je O(n*log(n)). Najgori slučaj predstavlja podela na particije neravnomerne veličine i to tako da se u prvom koraku donja particija sadrži od n-1 elemenata, u drugom od n-2 itd. Tada je složenost algoritma jednaka $O(n^2)$. Detaljnijom analizom i usrednjavanjem po svim slučajevima dobija se da je srednji slučaj mnogo bliži najboljem nego najgorem slučaju, odnosno reda je O(n*log(n)). Dakle:

$$MIN=O(n*\log(n))$$

$$AVG=O(n*\log(n))$$

$$MAX=O(n^2)$$

U kodu napisanom u jeziku C++ projekta *AlgortimiSortiranja* ovaj algoritam relaziovan je pozivanjem metode *sortiraj( )* klase *Radix.* Postupan prikaz algoritma dat je na slici 3.



Slika 3

4. *Pigeonhole sort*

Pigeonhole sort je algoritam koji je veoma sličan Counting sort algoritmu. Jedina razlika je u tome što se u pomoćnom nizu u kome se ranije čuvao broj pojavljivanja elementa sa datim indeksom, sada čuva zaglavlje liste. Sinonimi se stavljaju u ulančanu listu sa zaglavljem koje odgovara indeksu datog elementa u pomoćnom nizu.

Na prvi pogled bi se moglo reći da ova modifikacija radi istu stvar, uz dodatan trošak u prostoru. Međutim, ovaj metod favorizuje manipulaciju složenim tipovima podataka. Ukoliko recimo imamo tačke koje se sastoje od 2 koordinate koje je potrebno sortirati po koordinati X, na ovaj način se ne gubi podatak o koordinati Y podvođenjem pod isto X.

Samo funkcionisanje algoritma podrazumeva alociranje niza zaglavlja dovoljne veličine kako bi se svi elementi početnog niza smestili u konstantnoj složenosti (nalik heširanju). Dimenzija se najefikasnije određuje kao razlika najvećeg i najmanjeg elementa niza uvećana za jedan. Zatim se ti elementi vade od početka do kraja uz smeštanje natrag u početni niz. Vremenska složenost nije narušena u odnosu na Counting sort, jer se same operacije dodavanja i uklanjanja iz liste odvijaju u kontantnom vremenu.

$$MIN=(2n)$$

$$AVG=(2n+k)$$

$$MAX=(2n+k)$$

U kodu napisanom u jeziku C++ projekta *AlgortimiSortiranja* ovaj algoritam realizovan je pozivanjem metode *sortiraj( )* klase *PigeonHole.*

5. *Counting sort* (sortiranje brojanjem)

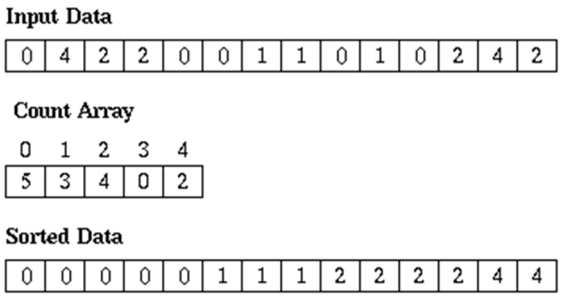
Counting sort ili sortiranje brojanjem je specifičan i skup metod za sortiranje ali daje odličnu vremensku složenost. On ne spada ni u jednu od četiri navedene grupe metoda, već se zasniva na brojanju ponavljanja svakog elementa u nizu tako što prođe jedanput kroz ceo niz a u pomoćnom nizu za svaki različiti element ima broj puta koliko se taj element pojavljivao. Nakon toga algoritam prolazi kroz novonastali niz i od prvog elementa niza koji će predstavljati sortiran poredak počinje da ređa elemente iz novonastalog niza (i to onoliko od svakog koliko se puta on ponavljao). Na taj način dobićemo sortiran poredak brojeva. Vremenska složenost ovog algoritma uvek je uvek O(n+k) gde je k=maxel-minel. Prostorna složenost je poprilično kritična, zato što je potreban niz od k elemenata a to može biti jako veliki broj. Dakle ukoliko je k manje od n a pretpostavlja se da jeste složenost algoritma je:

$$\text{MIN}= O(n+k)$$

$$\text{MAX}= O(n+k)$$

$$\text{AVG}= O(n+k)$$

U kodu napisanom u jeziku C++ projekta *AlgortimiSortiranja* ovaj algoritam realizovan je pozivanjem metode *sortiraj( )* klase *Counting_Up.* Postupan prikaz algoritma dat je na slici 4.



Slika 4

6. *Square (Friend) sort* (selekcija)

Kvadratna selekcija (*square sort*) je prvo poboljšanje metoda sortiranja selekcijom koje je patentirao Frend (*Friend*). Specifično je kao jedini algoritam za sortiranje koji ima složenost reda O(n$\sqrt{n}$).

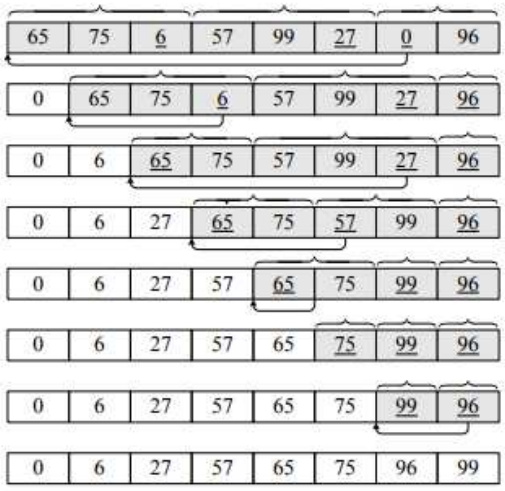
Ovo se postiže tako što se prvo podeli niz na $\sqrt{n}$ particija. Zatim se prođe jednom kroz niz i izvuku lokalni minimumi (maximumi, u daljem tekstu će se sortiranje opisivati neopadujuće, a u klasi *Square* se vrši nerastuće). Složenost ovog postupka je O(n). Dalje se od lokalnih minimuma nađe minimum i stavlja na početak pomoćnog niza i nalazi se ako postoji novi lokalni minimum iz njegove particije. Lokalni minimumi (maksimumi) se stavljaju u prioretni red (realizovano hipom (heap)). Složenost tog postupka je O($log\sqrt{n}$) a kako ima n elemenata ukupna složenost postupka je O($nlog\sqrt{n}$). Međutim, posle prvog koraka treba treba u $\sqrt{n}$ particija dohvatiti $\sqrt{n}-1$ minimuma. Složenost postupka je $\sqrt{n}$ O($\sum_{i=1}^{\sqrt{n}} i$)=O($n\sqrt{n}$) sa konstantnim faktorom $\frac{1}{2}$. Budući da algoritam ne može da ispituje da li je niz sortiran ista je složenost u najgorem, najboljem i prosečnom slučaju:

MIN= O($n\sqrt{n}$)

AVG= O($n\sqrt{n}$)

MAX= O($n\sqrt{n}$)

U kodu napisanom u jeziku C++ projekta *AlgortimiSortiranja* ovaj algoritam realizovan je pozivanjem metode *sortiraj( )* klase *Square.* Postupan prikaz algoritma dat je na slici 5.



Slika 5 – kvadratna selekcija(in-place iako u stvarnosti koristi pomoćni niz velicine n.)

7. *Shaker(Cocktail) sort* (direktna zamena)

Šejker (*Shaker*) algoritam za sortiranje je unapređenje *bubble* algoritma. Glavna razlika po kojoj je i dobio ime je što u spoljnoj petlji u svakom koraku menja pravac (kao šejker pri pravljenju koktela). Time se poboljšava prosečni slučaj *bubble* algoritma za sortiranje jer će i manji i veći ključevi u svakoj odgovorajućoj alternaciji biti bliži svojoj krajnjoj pozicji. Nažalost, ovo smanjuje samo konstanti faktor. Najogri slučaj je isti, kada je niz obrnuto sortiran, gde u i-toj iteraciji mora da se izvrši n-i-1 zamena (i poređenja). Kada se sumira:

$$\text{MIN}= O(n)$$

$$\text{MAX}= O(n^2)$$

$$\text{AVG}= O(n^2)$$

U kodu napisanom u jeziku C++ projekta *AlgortimiSortiranja* ovaj algoritam realizovan je pozivanjem metode *sortiraj( )* klase *Shaker*. Postupan prikaz algoritma dat je na slici 6.

Slika 6 – Koktel algoritam za sortiranje

8. Quick sort (zamena-particionisanje)

Quick sort – brzo sortiranje je tipičan rekurzivni algoritam koji u svakom prolasku deli niz na dva dela (particije), jednu gde se nalaze elementi niza manji od pivota i drugu gde se nalaze elementi veći od pivota i potom se poziva ponovo za jednu od particija sve dok particjia ne postane veličine jednog elementa. Radi efikasnog rada, algoritam se implementira pomoću korisničkog steka iterativno, gde se da bi stek bio u najgrem slučaju reda O(log n) uvek uzima manja particija (jer se ona brže sortira). Složenost u najboljem sličaju je red O(log n) i dobija se kada je pivot uvek medijana niza. Tada se svaki put prepolovljava veličina particije u kojoj se pivot stavlja na mesto. Tada se za particiju veličine n sortira 1 element veličine $\frac{n}{2}$ 2 elementa, $\frac{n}{4}$ 4 u $\frac{n}{2^i}$ 2^ielemenata. onda je kraj kada je $2^i = n$, odosno $i = log\ n$. budući da se za svako od tih sortiranja morao proći ceo niz onda je složenost najboljeg slučaja O(*nlogn)*. U prosečnom slučaju se onda samo menja konstantni faktor. Međutim u najgorem kada je pivot najmanji (najveći) element u particiji particija se ne deli nego samo ostatak niza postaje nova particija, a ceo niz se prolazi i poredi sa pivotom. Tada je složenost reda O($\sum_{i=1}^{n} i$)=O(n^2). Zato bi bilo idealno da je pivot medijana. Kako je pronalaženje medijane reda O(k^2) gde je k veličina particije, složenost postupka bi bila lošija od reda O(n^2). Zato je najbolje rešenje uzeti srednju vrednost, a ne element niza za pivot. Tada se gotovo garantuje složenost reda O(*nlogn)*, mada konstanti faktor u veoma ekstremnom slučaju kada je svaki element veći od sume svih prethodnih elemenata (za veće nizove čak i neostvarivo na savremenim računarima) opet O(n^2). Radi jednostavnosti, oslanjajući se na verovatnoću, urađenoj implementaciji se za pivot koristi pseuodslučajni element dobijen Knutovim linearnim generatorom.

U kodu napisanom u jeziku C++ projekta *AlgortimiSortiranja* ovaj algoritam realizovan je pozivanjem metode *sortiraj( )* klase *Quick.* Postupan prikaz algoritma dat je na slici 7.

Slika 7

MIN = O(*n*logn)*
AVG = O(*n*ln(n))*
MAX = O(*n*logn),* za n>k+1 MAX = O(n^2) za n<=k+1, gde je k broj bitova.

9) *Merge sort* (sortiranje spajanjem)

Algorimti sortiranja spajanjem rade tako što već sortirane delove niza spajaju u nov sortirani niz. Spajanje dva sortirana niza zahteva samo poređenje tekućih elemenata niza i ubacivanje manjeg (većeg) u pomoćni niz. Prostorna složenost je stoga O(n), dok je složenost postupka spajanja i+j, gde su i, j veličine spajanih delova. Kod realiozovane metode direktnog spajanja i=j,

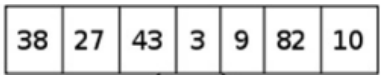
$$i\epsilon\{1,2,4..2^k|k\epsilon N,\ \ k=\max(N)\mid\ 2^k<n\}.$$

U svakom koraku spajaju se dve susedne particije veličine i. Takvih koraka ima k i još jedan korak za spajanje particija veličine 2^k i $n-2^k$. Iz toga sledi da ima *log(n)* koraka veličine n($\frac{n}{i}$ spajanja veličine *n*). Pa je složenost reda O(*n*logn*). Ovo važi u svim slučajevima osim ako se ne uvede marker (*flag*) koji govori da li su dva elementa pri spajanju zamenila mesta i koji propagirajući kroz delove ostaje tačan ako nema nijednog spajanja i svaka particija počinje sa elemtnom većim od prethodnog. U tom slučaju ako je marker tačan moglo bi se ranije završiti. Ovo poboljšanje donosi složenost u najboljem slučaju reda O(n), međutim toliko retko pomaže da je veći konstanti faktor u prosečnom slučaju veči problem. Zato se je složenost realnih implemntacija *merge sort*-a:

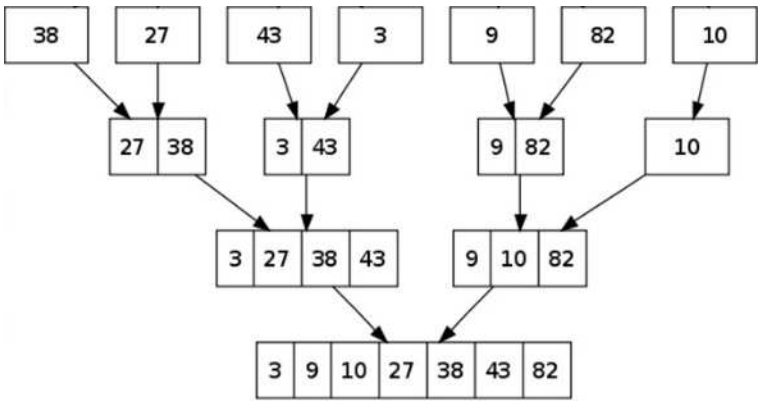
MIN = O(*n*logn)*

AVG = O(*n*logn)*

MAX = O(*n*logn).*



Slika 8: Početni niz



Slika 9: Merge sort

U kodu napisanom u jeziku C++ projekta *AlgortimiSortiranja* ovaj algoritam realizovan je pozivanjem metode *sortiraj( )* klase *Merge.*. Postupan prikaz algoritma dat je na slici 9.

Literatura:

https://stackoverflow.com/

Milo Tomašević „Algoritmi i strukture podataka“

https://www.thecrazyprogrammer.com/2015/04/counting-sort-program-in-c.html

https://www.hackerearth.com/practice/algorithms/sorting/selection-sort/tutorial/

Radili:

Đorđe Nikolašević 0074/2016

Marko Vekarić 0089/2016

Gojko Vučinić 0070/2016