1. Osnovni algoritmi nad listama i njihova svojstva

Na prethodnim predmetima iz programiranja ste se već upoznali sa najosnovnijim algoritmima nad nizovima u programskom jeziku C (ili nad listama u Pythonu). Ovdje ćemo ponoviti osnovni algoritam linearne pretrage i vidjeti dodatni algoritam binarne pretrage za nizove.

U najgeneralnijem smislu listu definiramo kao skup elemenata u određenom poretku. Iako ćemo o elementima, nizovima i listama pričati detaljnije u idućim poglavljima, ovdje ćemo se sa listom uglavnom referirati na obične nizove brojeva da bi ilustrirali različite algoritme. Svejedno, da bi opisali osnovnu ideju svakog algoritma, možemo koristiti apstraktni tip podatka lista. S takvim apstraktnim tipom podatka možemo ignorirati mnoge detalje oko stvaranja liste ili pristupa pojedinima elementima i koncentrirati se na algoritam. Svaki algoritam ćemo zatim implementirati za C nizove. U idućem poglavlju će se neki od tih algoritama primijeniti i za dinamički vezane liste.

Linearna i binarna pretraga

Zadatak pretrage je pronaći nekakav element (broj) unutar neke liste elemenata (niza brojeva). Linearna pretraga se sastoji od jednostavnog prolaska kroz niz i usporedbe sa svakim članom niza. Algoritam može biti implementiran idućim kôdom:

```
int linearna_pretraga(int b, int *niz, int n) {
      for (int i = 0; i < n; i++) {
            if (niz[i] == b)
                  return 1;
      }
      return 0;
}
```

Slično kao u primjeru *trazi_nulu()*, možemo vidjeti da će algoritam prolaziti kroz niz doke ne pronađe broj ili dođe do kraja niza. Možemo izračunati da pripada u *O(n)* klasu složenosti odnosno da će vrijeme izvođenja rasti linearno sa dužinom niza.

Algoritme nad nizovima često možemo napisati bolje ako je niz poredan prema nekom pravilu, npr. ako su brojevi u nizu sortirani prema veličini. Binarna pretraga je jedan takav primjer gdje, ako je niz sortiran, možemo uštediti puno vremena odnosno smanjiti klasu složenosti sa O(n) na O(log(n)). Ideja je slična kao pretraga rječnika za određenim pojmom. Kako su pojmovi u rječniku sortiran leksikografski (abecedno) možemo krenuti tražiti otprilike od sredine rječnika, zatim opet ići na pola puta prema početku ili kraju, i tako dalje. Ono što nas vodi naprijed-nazad po rječniku je usporedba traženog pojma sa pojmom na koji smo naišli.

Slično, na metodičan način, možemo napraviti i sa nizom brojeva. Prvi korak je pogledati broj na sredini niza i usporediti ga sa traženim brojem. Kako su brojevi sortirani, možemo odmah eliminirati jednu polovicu niza. Sada preostalu polovicu možemo opet tretirati kao niz i pogledati broj po sredini. Cijeli algoritam se može implementirati sa idućim kôdom:

```
int binarna_pretraga(int b, int *niz, int n) {
	int l = 0;
	int r = n;
	while (l < r) {
		int c = (l + r) / 2;
		if (b < niz[c])
			r = c;
		else if (b > niz[c])
			l = c + 1;
		else
			return 1;
	}
	return 0;
}
```

Analizom opisanog algoritma (ili analizom implementacije) možemo vidjeti da je složenost binarne pretrage O(log(n)). Niz koji ulazi u algoritam ima dužinu *n* i na svakom koraku odbacimo polovicu niza dok ili ne pronađemo traženi broj ili dužina preostalog niza ne postane 0. Zbog toga će glavna petlja algoritma najviše iterirati x puta, gdje je $n = 2^x$ odnosno $x = \log_2(n)$ i $T(n) = O(\log(n))$. To je ujedno i najgora složenost algoritma jer ne postoji ulaz za koji će petlja iterirati više od $\log_2(n)$ puta.

Algoritam možemo napisati i rekurzivno. To nije standardan način za implementaciju algoritma, ali je dobar primjer izračuna složenosti za rekurzivnu funkciju.

```
int binarna_pretraga(int b, int *niz, int n) {
      if (n == 0)
            return 0;
      int c = (l + r) / 2;
      if (b < niz[c])
            return binarna_pretraga(b, niz, c);
      else if (b > niz[c])
            return binarna_pretraga(b, niz + c + 1, r - c);
      else
            return 1;
}
```

Iz kôda se može vidjeti da će se svaki poziv funkcije izvesti u nekom konstantom vremenu, ako ignoriramo vrijeme potrebno za rekurzivan poziv. Rekurzivan poziv će proslijediti jednu polovicu niza. Zbog toga T(n) možemo napisati kao:

$$T(n) = c + T(n/2)$$

gdje T(n/2) označava složenost funkcije binarne pretrage za polovicu niza. Kako je $T(n/2) = c + T(n/4)$, izraz se može raspisati dalje:

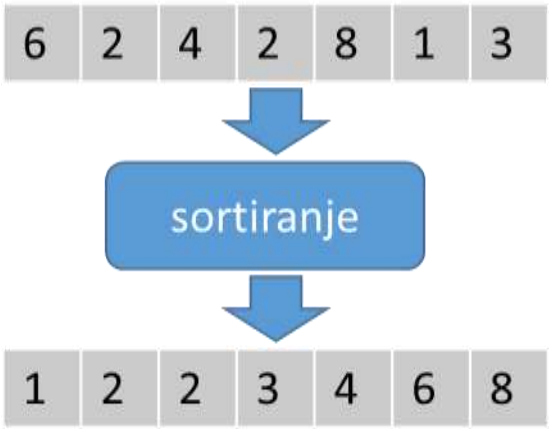
$$T(n) = c + c + T(n/4) = c + c + c + \ldots + c + O(1)$$

gdje je *O(1)* na kraju konstantna složenost kada se funkcija više ne poziva rekurzivno i vrati 1 ili 0. Koliko će konstanti *c* biti odnosno koliko će se puta funkcija rekurzivno pozvati? Kako se svaki put *n* prepolovi, opet dolazimo da će ih biti najviše $\log_2(n)$ i složenost funkcije je O(log(n)) i u srednjem i u najgorem slučaju.

Zašto se preferira prva (iterativna) implementacija algoritma? Što je vremenske složenosti tiče, razlike nema, a u praksi je petlja možda malo brža od rekurzivnih poziva. Ako pogledamo memorijsku složenost, možemo vidjeti i teoretsku razliku između dvije implementacije. U prvom slučaju je vremenska složenost O(1) jer deklariramo nekoliko varijabli što zauzima točno određenu količinu prostora u memoriji. Kod rekurzivnog slučaja, svaki rekurzivni poziv će zauzeti dodatnu memoriju na programskom stogu. Zbog toga će memorija rasti sa brojem rekurzivnih poziva odnosno biti će O(log(n)). To sigurno neće uzrokovati problem u praksi (da iscrpimo dostupnu memoriju računala) jer će nam puno više memorije trebati za samo predstavljanje podataka. Svejedno, prvo rješenje je i teoretski bolje.

Algoritmi sortiranja

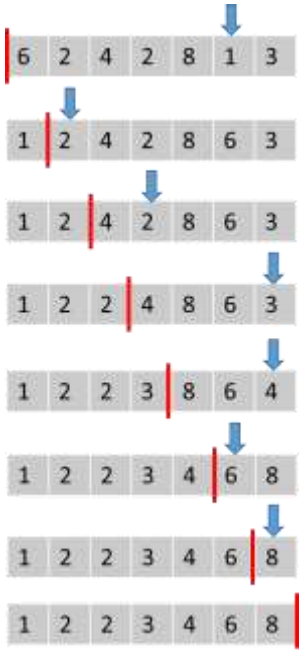
Zadatak algoritama sortiranja je da ulaz u obliku niza (ili generalnije, liste) poreda po nekom jasno definiramo poretku. Sve primjere ćemo prikazati na nizu brojeva, osim kada budemo opisivali svojstvo stabilnosti algoritma sortiranja gdje su nam potrebne složenije strukture podataka.



Postoje desetci algoritama sortiranja i njihovih varijanti. Razni algoritmi imaju različita svojstva, mane i prednosti, ali njihova osnovna značajka je klasa složenosti, tipično $O(N^2)$ ili O(NlogN). Ovdje ćemo vidjeti i usporediti četiri algoritma sortiranja koji spadaju u dvije klase složenosti. Kasnije ćemo vidjeti i peti algoritam koji se temelji na posebnoj strukturi podataka.

Selectionsort

Jedna moguća strategija za sortiranje liste brojeva bi mogla biti da iz liste uzimamo uvijek najmanji broj i prebacimo ga na kraj nove liste. Kako svaki put na kraj nove liste dodajemo broj koji je veći ili jedna svim prethodnim brojevima, očigledno će ta lista na kraju biti sortirana uzlazno. Ovakav algoritam bi u implementaciji imao nekoliko problema od kojih je najbitniji da nam je potrebna dodatna memorija za novu list koju kreiramo. Zbog toga možemo logički podijeliti originalni niz na dva dijela: sortirani i nesortirani. Na početku će sortirani dio imati dužinu 0, a svi brojevi se nalaze u nesortiranom dijelu dužine *n*.



Kako prebacujemo jedan po jedan najmanji broj iz nesortiranog dijela u sortirani dio, dužina sortiranog dijela će rasti kako dužina nesortiranog dijela pada, a ukupna dužina će uvijek biti *n*. Time ćemo izbjeći dodatnu memoriju (osim nekoliko varijabli za indekse i zamjenu) i cijeli algoritam može biti efikasno implementiran sa idućim kôdom:

```
void selectionsort(int *niz, int n) {
	// i nam je granica između sortiranog i nesortiranog dijela
	for (int i = 0; i < n - 1; i++) {
		// tražimo indeks najmanjeg elementa
		int mini = i;
		for (int j = i + 1; j < n; j++) {
			if (niz[j] > niz[mini])
				mini = j;
		}
		// zamijena elemenata na mini i i indeksima
		int tmp = niz[mini];
		niz[mini] = niz[i];
		niz[i] = tmp;
	}
}
```

Složenost možemo procijeniti ili čitajući implementaciju ili analizom opisanog algoritma i procjenom složenosti pojedinih operacija. Svako traženje najmanjeg broja će nas koštati koliko i linearna pretraga niza, odnosno O(n) gdje je n dužina nesortiranog dijela niza. Kako svaki put prebacujemo po jedan broj iz nesortiranog u sortirani dio, dužina nesortiranog dijela će biti *n*, pa *n-1*, *n-2*, … dok ne ispraznimo nesortirani dio. Tako da, ignorirajući konstante, na kraju imamo zbroj:

$$T(n) = n + (n-1) + (n-2) \ldots + 1$$

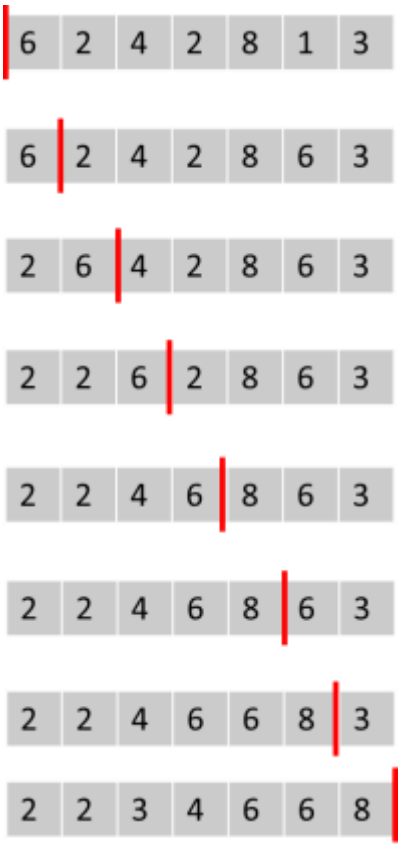
$$T(n) = n * (n + 1) / 2$$

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

Tvrdnju da je $n + (n-1) + (n-2) \ldots + 1 = n * (n + 1) / 2$ se može lako dokazati indukcijom.

Insertionsort

Ideja insertionsorta je na neki način suprotna od selectionsorta. Ovdje redom izvlačimo elemente iz originalne liste. Svaki element stavljamo u novu listu na mjesto određeno poretkom odnosno tražimo njegovo mjesto prema veličini. Nova lista će uvijek biti sortirana i kada se isprazni originalna lista, algoritam je gotov.



U implementaciji sa nizovima će se ponovno koristiti trik sa dijeljenjem niza na sortirani i nesortirani dio. Na kraju cijeli algoritam može biti implementiran sa idućim kôdom:

```
void insertionsort(int *niz, int n) {
	// i određuje granicu sortiranog i nesortiranog dijela
	for (int i = 1; i < n; i++) {
		// tražimo mjesto elementu u sortiranom dijelu niza
		int j = i;
		while (j > 0 && niz[j - 1] > niz[j]) {
			// zamijena elemenata
			int tmp = niz[j - 1];
			niz[j - 1] = niz[j];
			niz[j] = tmp;
			j--;
		}
	}
}
```

Ova implementacija traži mjesto u sortiranom dijelu od kraja prema početku. Pri tom novi element mijenja mjesto sa prethodnim elementom i efektivno „izgura“ sve elemente veće od elementa prema kraju sortiranog dijela. Zbog toga nas svaka operacija ubacivanja u sortirani dio dužine *n* košta O(n). Dužina sortiranog dijela će biti 1 prvi put, zatim 2, 3, 4, … do n. ako da, ignorirajući konstante, na kraju imamo zbroj:

$$T(n) = 1 + 2 + 3 + \ldots + (n-1) + n$$

$$T(n) = n * (n + 1) / 2$$

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

Iako oba algoritma (selectionsort i insertionsort) na kraju spadaju u $O(n^2)$ klasu složenosti, razlike postoje ako se detaljnije analizira implementacija. Iz implementacija je jasno da će insertionsort izvesti puno više zamjena elemenata u odnosu na selectionsort. Tako da relativna brzina izvođenja algoritama ovisi o stvarnoj cijeni zamijene elemenata u odnosu na operaciju usporedbu elemenata. Na primjer, ako sortiramo na sporom mediju poput diska, vjerojatno ćemo preferirati selectionsort da bi minimizirali zamjene. To nas neće toliko brinuti ako je niz u radnoj memoriji. Pri izboru algoritma, uvijek je dobro uzeti u obzir stvarnu cijenu pojedinih operacija i koliko ih često algoritam upotrebljava. Postoji još cijeli niz detalja koji se mogu uzeti u obzir, ali na kraju je najpouzdanija mjera testiranje različitih algoritama na zadanom problemu. To je takozvana *a-posteriori* analiza.

Mergesort

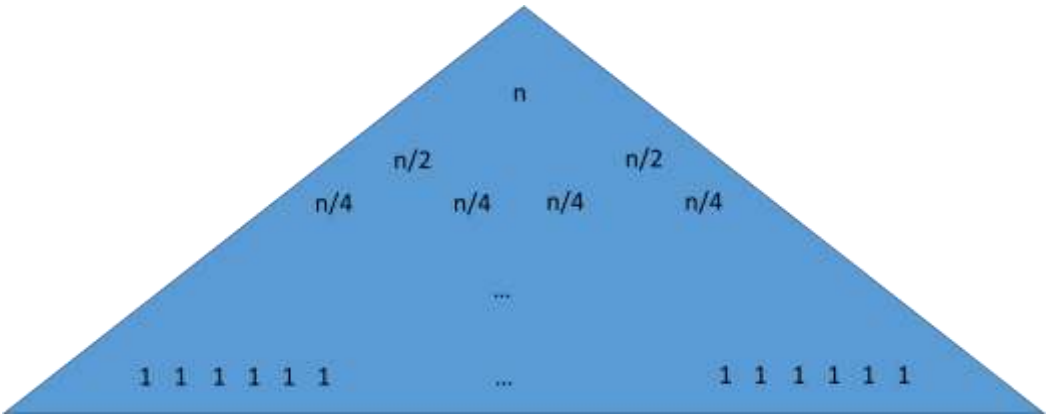
Pri dizajnu algoritama ponekad može pomoći podijeliti problem na manje dijelove koji će se kasnije kombinirati u rješenje. Taj pristup je poznat pod imenom „podijeli pa vladaj“ i može se primijeniti i na algoritme sortiranja. Ako uzmemo jedan od prethodna dva algoritma sortiranja, možemo vidjeti da je vremenska složenost algoritma $O(n^2)$ gdje je *n* dužina niza koji sortiramo. Zamislimo algoritam koji podijeli niz na dva podjednaka dijela, sortira posebno svaki dio i zatim kombinira dva dijela u jedan sortirani niz. Možemo izračunati njegovu vremensku složenost kao zbroj složenosti svih koraka: dijeljenja, sortiranja svake polovice i kombiniranja. Dijeljenje i kombiniranje možemo izvesti sa složenosti manjom od sortiranja dva dijela, tipično u O(n) i to ćemo pretpostaviti za daljnju analizu. Onda možemo napisati da je:

$$T(n) = O(n) + O((n/2)^2) + O((n/2)^2) + O(n)$$

$$T(n) = 2 * O(n) + 2 * O(n^2/4)$$

$$T(n) = O(n + n^2/2)$$

Ovaj trik ne bi bilo previše koristan jer je klasa složenosti i dalje $O(n^2)$, ali možemo ga nastaviti primjenjivati na svaku polovicu niza posebno, da bi na kraju došli do pojedinih elemenata gdje je složenost sortiranja O(1). Tada bi ukupna vremenska složenost bila sastavljena samo od operacija dijeljenja i kombiniranja dijelova niza. Cijeli proces možemo vizualizirati neformalno kao piramidu.



Na svakoj razini piramide moramo dijeliti i kombinirati sve manje dijelove niza, pa je složenost ukupne obrade svake razine u piramidi O(n). Slično kao kod binarne pretrage, niz možemo dijeliti samo log(n) puta dok ne dođemo do pojedinih elemenata, tako da je visina piramide log(n) i konačna složenost takvog algoritma je O(nlog(n)).

Najjednostavniji algoritam koji koristi „podijeli pa vladaj“ je mergesort. Karakteristika mergesorta je da je dijeljenje niza jednostavno, a većina posla se obavlja pri kombiniranju dijelova niza. Možemo ga opisati rekurzivno sa idućim koracima: podijeli niz po sredini, primijeni mergesort na svakoj polovici, kombiniraj rezultat pomoćnom *merge* funkcijom. Algoritam neće napraviti ništa ako primi niz dužine manje od 2 i to je uvjet zaustavljanja dijeljenja niza.

Ilustrirani algoritam se može implementirati kao rekurziju na nizovima uz pomoć dodatnih funkcija za izradu kopije niza *duplicate()* i kombiniranje nizova *merge()*.

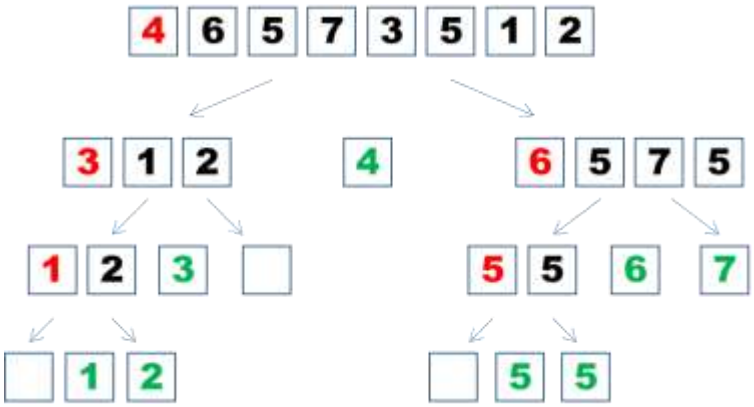
```
void mergesort(int *niz, int n) {
	// zaustavljamo se ako niz ima 0 ili 1 element
	if (n < 2)
		return;
	// dijelimo niz po sredini
	int half = n / 2;
	int* left = duplicate(niz, half);
	int* right = duplicate(niz + half, n - half);
	// rekurzivno pozivamo mergesort na polovicama
	mergesort(left, half);
	mergesort(right, n - half);
	// kombiniramo dvije polovice
	merge(left, half, right, n - half, niz);
	// oslobađa se memorija za polovice
	free(left);
	free(right);
}
```

Pomoćna funkcija *duplicate()* kreira kopiju proslijeđenog niza i zbog toga ih je potrebno osloboditi zauzetu memoriju. Funkcija merge() kombinira dva sortirana niza u jedan sortirani niz i ključno je da se to spajanje izvede efikasno. Najbrži način je da se dva indeksa postave na početak svake polovice i u treći niz se prebacuje po jedan element iz jedne od polovica. Ako uvijek biramo manji element između dva indeksirana elementa, rezultirajući niz će biti sortiran. Vremenska složenost cijele operacije kombiniranja je *O(n)*, gdje je n ukupna dužina rezultirajućeg niza. Implementacija same funkcije je ostavljena za vježbu. Ukupna vremenska složenost je *O(NlogN)* jer će se kombiniranje obavljati u dijelovima za cijeli niz na svakoj razini „piramide“, odnosno rekurzije. To je u klasi najboljih algoritama za sortiranje, ali postoji jedna mana zbog koje se često odabire neki drugi algoritam sortiranja. Mana je memorijska složenost mergesorta. Problem je što *duplicate()* pomoćna funkcija zauzima dodatnu memoriju. Kroz rekurzije memoriju zauzimamo i oslobađamo po potrebi, ali memorijska složenost je gornja granica potrebne memorije, bilo kada u tijeku algoritma. Ta granica će se dostići kada rekurzija dođe od uvjeta zaustavljanja. U tom trenutku će svaka razina zauzeti memoriju dovoljnu za cijeli primljeni niz, odnosno ukupna memorija će biti složenosti $O(n + n/2 + n/4 + ... + 1)$ što je efektivno $O(n)$. To znači da će nam, uz memoriju potrebnu za ulazni niz, biti potrebno još toliko memorije za izvođenje algoritma. Ako nam je to prihvatljivo, mergesort ima i neke prednosti. U najgorem slučaju, složenost će i dalje biti O(NlogN) jer niz uvijek dijelimo na pola, što ne mora biti istina za druge algoritme sortiranja.

Quicksort

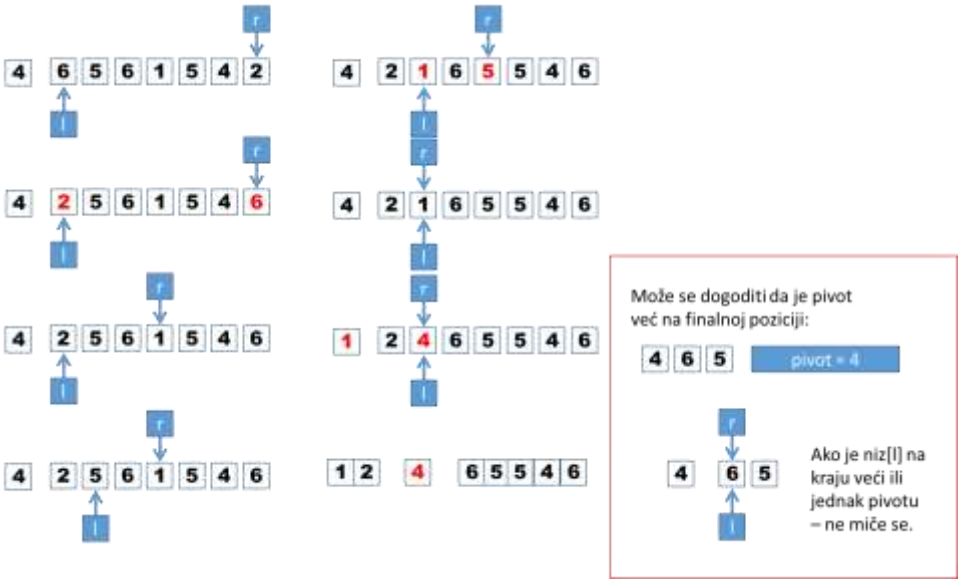
Quicksort je još jedan primjer algoritma prema principu podijeli-pa-vladaj. U ovom algoritmu se većina posla obavlja prilikom dijeljenja niza na dva dijela, a kombiniranje dijelova je trivijalno.

Ideja je podijeliti niz na dva dijela, tako da se u jednom dijelu nalaze „manji“ brojevi, a u drugom dijelu „veći“ brojevi. Manji i veći u odnosu na neki element niza kojeg odaberemo i naziva se pivot (prag, granica). Za pivot možemo za sada odabrati prvi element u nizu, a kasnije ćemo vidjeti moguće probleme i poboljšanja. Spajanje dva dijela je jednostavno spajanje redom: „manjeg“ dijela, pivota i „većeg“ dijela. Cijeli algoritam se može ilustrirati slikom.



Na slici su pivoti (prvi elementi u listi) označeni crveno kada dijelimo niz i zeleno kada spajamo tri dijela. Ako se pogledaju samo zeleni pivot, vidi se da su poredani po veličini. To je posljedica podijele niza na sve manje i veće elemente. Zbog toga će se pivot automatski naći na „pravom“ mjestu prilikom spajanja. Kao i prije, algoritam ne radi ništa za nizove sa manje od dva elementa. U stvarnosti se dijeljenje niza na manje i veće elemente treba obaviti „u mjestu“, odnosno unutar istog niza koji sortiramo. To možemo napraviti zamjenom mjesta elemenata u nizu i to je najčešće implementirano kao posebna funkcija partiticije.

Funkcija partiticije uzima za pivot element na prvom mjesto u nizu i preuređuje ostatak niza sa dva indeksa - „lijeva i desna ruka“. Desna ruka kreće od kraja niza i traži prvi element koji je manji od pivota. Lijeva ruka kreće od početka (ostatka) niza i traži element veći ili jednak pivotu. Kada su obje ruke pozicionirane, elementi mijenjaju mjesto. Kada se lijeva i desna ruka susretnu, svi elementi lijevi od mjesta susreta su manji od pivota, a desno su veći. Na kraju je još samo potrebno pozicionirati pivot. Ako je pivot veći ili jednak elementu na mjestu susreta, onda se pivot zamijeni sa tim elementom. U suprotnom, pivot ostaje na prvom mjestu. U oba slučaja, pivot je na pravom mjestu u sortiranom nizu i potrebno je samo sortirati lijevi i desni dio niza.



Na kraju, quicksort je jednostavna rekurzivna funkcija koja se oslanja na funkciju partiticije. Složenost quicksort algoritma se može izračunati uz pretpostavku da se niz svaki puta dijeli na približno dva jednaka dijela. U tom slučaju, dobiti ćemo piramidu sličnu kao kod mergesorta. Kako funkcija partiticije ima složenost *O(n)*, na kraju će vremenska složenost biti *O(NlogN)*. Problem nastaje kada pretpostavka o podijeli niza na dva približno podjednaka dijela nije ostvarena. Najbolji primjer za to je niz koji je već sortiran, odnosno kada je pivot u svakom koraku manji (ili veći) od svih drugih elemenata. Tada će se pri svakom koraku quicksorta sortirati samo pivot, a ostatak niza će ići u idući korak (rekurzivni poziv). To je najgori slučaj za quicksort algoritam i efektivno će „piramida“ biti visine *O(n)* tako da će ukupna složenost biti $O(n^2)$. Zbog tog problema se osnovni quickosrt algoritam nadograđuje sa nekoliko dodataka. Jedna mogućnost je birati pivot među nekoliko elemenata, tipično između elemenata na prvom, zadnjem i srednjem indeksu. Uzima se srednji od ta tri elementa i prebacuje se na početak niza. Time se efektivno aproksimira medijan niza, koji bi bio idealan pivot. To rješenje je još uvijek manjkavo, na primjer za niz koji sadrži sve iste elemente, složenost će i dalje biti $O(n^2)$. Moguće je i provjeriti da li je niz (skoro) sortiran prije nego uopće krenemo sa quicksort algoritmom i odabrati drugi algoritam za takav slučaj.

Dodatni problem je što će u praksi, za male nizove, quicksort biti sporiji od insertionsort ili selectionsort algoritama. To se može riješiti unutar quicksort rekurzije gdje, ako je primljeni niz manji od nekog minimuma (npr. 20 elemenata), umjesto pozivanja iduće rekurzije pozovemo npr. insertionsort.

```
int partition(int *niz, int n) {
	int l, r;
	l = 1;
	r = n - 1;
	while (l < r) {
		if (niz[r] >= niz[0]) {
			r--;
		}
		else if (niz[l] < niz[0]) {
			l++;
		}
		else {
			int tmp = niz[l];
			niz[l] = niz[r];
			niz[r] = tmp;
		}
	}
	if (niz[0] < niz[r]) {
		return 0;
	}
	else {
		int tmp = niz[r];
		niz[r] = niz[0];
		niz[0] = tmp;
		return r;
	}
}

void quicksort(int *niz, int n) {
	if (n < 2)
		return;
	int pi = partition(niz, n);
	quicksort(niz, pi);
	quicksort(niz + pi + 1, n - pi - 1);
}
```

Zadaci

1) Implementirati *merge()* funkciju za mergesort.
2) Kolika je memorijska složenost za pokazane implementacije četiri algoritma sortiranja?
3) Kolika je složenost najboljeg slučaja za pokazane implementacije četiri algoritma sortiranja?
4) Procijeniti klasu složenosti O() algoritma koji računa razliku dva skupa (dva niza dužine N i M sa svim različitim brojevima). Razlika skupova je niz brojeva koji sadrži sve brojeve iz prvog niza koji se ne nalazi u drugom nizu. Algoritam se može napisati na tri načina:
   a) Nijedan niz nije sortiran. Svaki broj u prvom nizu uspoređujemo sa svim brojevima drugog niza.
   b) Drugi niz je sortiran. Svaki broj u prvom nizu tražimo među brojevima drugog niza upotrebom binarne pretrage.
   c) Oba niza su sortirana. Postavimo indeks j na početak drugog niza. Za svaki broj u prvom nizu provjeravamo redom sve brojeve u drugom nizu koji su manji ili jednaki od traženog broja uvećavajući indeks j. Indeks j se nikada ne vraća na početak drugog niza.
5) Procijeniti klasu složenosti O() algoritma koji računa histogram za niz brojeva dužine N. Histogram je vezana lista gdje svaki element sadrži jedan broj iz niza i koliko puta se taj broj pojavljuje u nizu. Ukupno imamo M različitih brojeva u nizu tako da će histogram na kraju imati M elemenata. Algoritam se može napisati na tri načina:

   a) Prolazimo kroz niz i za svaki broj provjerimo da li se već nalazi u listi. Ako da, uvećamo brojač za taj broj. Ako ne, dodamo ga na početak liste.
   b) Sortiramo niz. Nakon toga prolazimo kroz sortirani niz i za svaki novi broj dodamo element na početak liste i brojimo koliko puta se pojavljuje dok ne naiđemo na idući broj.
   c) Prolazimo kroz niz i za svaki broj provjerimo da li se već nalazi u listi. Ako da, uvećamo brojač za taj broj. Ako ne, dodamo ga po redu (veličini broja) u listu.

6) Procijeniti složenost O() algoritma koji računa operaciju za 3 skupa (3 niza brojeva) dužine N, M i K. Algoritam kreira novi skup tako da uzima sve element iz prvog skupa (dužine N) koji se su veći od svih elemenata u drugom skupu (dužine M), ali i manji od svih elemenata u trećem skupu (dužine K). Algoritam se može napisati na dva načina:
   a) Nijedan niz nije sortiran. Prolazimo kroz prvi niz i za svaki broj provjerimo da li postoji veći broj u drugom ili manji broj u trećem nizu.
   b) Pronađemo najveći i najmanji broj u drugom i trećem nizu. Nakon toga iz prvog niza uzmemo sve brojeve veće od najvećeg iz drugog niza i manje od najmanjeg iz trećeg niza.
   c) Sortiramo silazno drugi niz i uzlazno treći niz. Nakon toga iz prvog niza uzmemo sve brojeve veće od prvog broja drugog niza i manje od prvog broja trećeg niza.

Složenost izraziti u N, M i K. Možete pretpostaviti da će pretrage u (a) proći pola niza, a da se za sortiranje u (b) koristi quicksort.