Quicksort

Quicksort je algoritam za sortiranje čija vremenska slozenost, u najgorem slučaju, iznosi $O(n^2)$, ali i uprkos tome je algoritam za sortiranje koji se najčešće koristi. Njegova prednost leži u tome što mu je očekivano vreme sortiranja $O(n \log n)$, gde je konstanta kod $n \log n$ veoma mala.

Quicksort koristi jedan od osnovnih algoritamskih principa - **podeli pa vladaj**, koji se zasniva na sledećoj ideji:

1. **Podeli** - ukoliko je problem koji rešavamo veoma mali, reši ga koristeći brute force. U suprotnom podeli problem na više potproblema (uglavnom 2), uglavnom istih veličina.

2. **Vladaj** - Koristeći rekurziju reši svaki potproblem

3. **Kombinuj** - kombinujući rešenja svakog od potproblema, i tako reši prvobitni problem

Glavna ideja quicksort-a je u tome da izaberemo jedan element u nizu, poznat kao '**pivot'**, neka to bude element x i podelimo niz na dva dela gde se u jednom delu nalaze svi elementi $\leq x$, isključujući pivota, a u drugom delu svi elementi $> x$. Zatim rekurzivno sortirajmo ova dva dela (u rekurziji stajemo kada imamo da sortiramo niz koji sadrži samo jedan element) i na kraju dobijamo sortirani niz tako što spojimo deo sa elementima $\leq x$, pa pivot, pa deo sa elementima $> x$.

Moguća implementacija quicksort-a je prikazana u *Algoritam 1*, gde funkcija $partition(a, left, right)$ posle izvršavanja vraća poziciju pivota i razvrstava elemente niza a tako da svi elementi pre pivota budu manji ili jednaki od njega, a svi elementi posle pivota budu veći od njega.

```
================================================================================
            funkcija: quicksort
            ulaz:  a      - niz brojeva
                   left   - indeks prvog elementa u delu niza koji posmatramo
                   right  - indeks poslednjeg elementa u delu niza koji posmatramo

            nakon izvršavanja funkcije quicksort(a, left, right), podniz
            a[left..right] će biti sortiran
--------------------------------------------------------------------------------
      Function quicksort(a : int array, left : int, right : int)
             01     if left < right then // imamo više od jednog elementa
             02         p = partition(a, left, right)
             03         quicksort(a, left, p - 1)
             04         quicksort(a, p + 1, right)
================================================================================
```

Algoritam 1. Pseudo kod za quicksort

Kako implementirati funkciju $parition$? Uzmimo poslednji element posmatranog dela niza za pivota. Krenimo od prve pozicije kroz posmatrani deo niza i ukoliko naiđemo na broj koji je manji ili jednak od pivota, zamenimo ga sa prvim posećenim elementom koji je veći od pivota (ukoliko ne postoji takav element, samo nastavimo dalje). Na taj način ćemo pošto prođemo sve elemente, imati da su na početku svi elementi manji ili jednaki od pivota, a iza njih svi elementi veći od pivota, i na kraju pivot. Da bismo završili particiju, samo zamenimo pivota (koji je na poslednjem mestu) sa prvim brojem u nizu većim od pivota. Implementacija ove ideje se nalazi u *Algoritam 2*.

```
================================================================================
            funkcija: partition
            ulaz:  a      - niz brojeva
                   left   - indeks prvog elementa u delu niza koji posmatramo
                   right  - indeks poslednjeg elementa u delu niza koji posmatramo
            izlaz: pozicija pivota nakon particije dela niza a[left..right]
                   nakon izvršavanja funkcije partition(a, left, right) elementi
                   pre pivota će biti manji ili jednaki od njega, a elementi posle
                   pivota će biti veći od njega.
--------------------------------------------------------------------------------
      Function partition(a : int array, left : int, right : int)
            01     x = a[right] // uzmimo poslednji element kao pivot
            02     p = left - 1 // poslednji element manji ili jednak od pivota
            03     for i = left to right - 1 do
            04         if a[i] <= x then
            05             p = p + 1
            06             tmp = a[i]
            07             a[i] = a[p]
            08             a[p] = tmp
            09
            10     // stavi pivota na dobru poziciju u nizu
            11     p = p + 1
            12     tmp = a[p]
            13     a[p] = a[right]
            14     a[right] = tmp
            15
            16     return p
================================================================================
```

Algoritam 2. Pseudo kod za partition

Fina osobina quicksort-a je da sortira dati niz '**u mestu**' što znači da zahteva dodatne $O(1)$ memorije, ne računajući $O(\log n)$ memorije koje zahteva na steku (prilikom pozivanja rekurzije).

Analizirajmo složenost ovog algoritma. Složenost funkcije $partition$ je $O(right - left)$, tj. linearna u odnosu na veličinu dela niza koji treba da particioniše.

Sada probajmo da odredimo složenost funkcije quicksort. Najpre posmatrajmo idealan slučaj, gde prilikom svakog poziva funkcije $partition$, pivot završi na sredini posmatranog dela niza. Ukoliko sa $f(n)$ označimo složenost funkcije quicksort prilikom sortiranja niza dužine n, dobijamo sledeću rekurentnu relaciju u ovom idealnom slučaju:

$$f(n) = O(n) + 2f(n/2)\text{ , čije je rešenje } f(n) = O(n \log n)$$

Posmatrajmo sada najgori slučaj, kada pivot završi na kraju posmatranog dela niza (kada kao pivota odaberemo najveći ili najmanji element). U tom slučaju rekurentna relacija je:

$$f(n) = O(n) + f(n-1)\text{, čije je rešenje } f(n) = O(n^2)$$

Do ovih procena složenosti možemo da dođemo i na malo intuitivniji način. Primetimo da u svakom koraku rekurzije mi ustvari prolazimo kroz čitav niz (zapravo se broj elemenata smanjuje za 1 pri ulasku u sledeći nivo rekurzije, ali zbog jednostavnosti, recimo da ostaje isti). Dakle složenost quicksort-a, je $f(n) = O(n) * DubinaRekurzije$. Kolika je dubina rekurzije u ova 2 slučaja?

U prvom slučaju, možemo primetiti da, pošto se broj posmatranih elemenata smanjuje 2 puta (na početku je n, pa u sledećem nivou rekurzije $n/2$, pa $n/4$, itd...) dubina rekurzije je $\log n$, pa je složenost quicksort-a $O(n \log n)$.

U drugom slučaju se broj posmatranih elemenata smanjuje za 1, pa dobijamo da je dubina rekurzije n, te je složenost u ovom slučaju $O(n^2)$.

Šta se događa u očekivanom slučaju? Dokaz da je očekivano vreme $O(n \log n)$ ćemo ovde izostaviti. Jedan od načina za procenu složenosti u očekivanom vremenu je da pretpostavimo da pivot sa jednakom verovatnoćom može biti svaki od brojeva u posmatranom delu niza. Ukoliko pretpostavimo da su svi brojevi različiti, jer ukoliko su svi brojevi isti dobijamo složenost $O(n^2)$, možemo da postavimo rekurentnu relaciju za očekivano vreme:

$$f(n) = O(n) + \frac{1}{n}\sum_{i=1}^{n}(f(i-1) + f(n-i))$$

gde posle malo igranja sa ovom relacijom dobijamo da je očekivano vreme $f(n) = O(n \log n)$.

Međutim, pokušajmo intuitivno da vidimo zašto je očekivano vreme bliže najboljem nego najgorem slučaju. Pretpostavimo da pivot uvek završi na $n/100$ od kraja posmatranog dela niza. Na prvi pogled bi se učinilo da će ovde vreme izvršavanja biti blizu $O(n^2)$, međutim, postavimo rekurentnu relaciju:

$$f(n) = O(n) + f(n/100) + f(99n/100)$$

Možemo primetiti da je dubina rekurzije u ovom slučaju $\log_{100/99} n$, pa je složenost opet $O(n \log n)$.

Kao što je napomenjeno u lekciji Složenost algoritama, kako bismo bili sigurni da će quicksort raditi u očekivanom vremenu, pre sortiranja, treba napraviti slučajnu permutaciju niza, pa ga zatim sortirati.