Osnove

Uloga algoritama u računarstvu

Algoritmi

Algoritam je strogo definisana kompjuterska procedura koja uzima vrednost ili skup vrednosti, kao **ulaz** i proizvodi neku vrednost ili skup vrednosti, kao **izlaz**. Drugim rečima, algoritam je niz računskih koraka koji transformišu ulaz u izlaz.

Algoritam se takođe može posmatrati i kao alat za rešavanje dobro definisanih kompjuterskih problema. U opštem slučaju postavka problema definiše željenu relaciju između ulaza i izlaza. Algoritam zapravo definiše specifičnu proceduru za dobijanje relacije između ulaza i izlaza.

Uzmimo, na primer, da je potrebno urediti niz brojeva u neopadajući raspored. Ovaj problem se često javlja u praksi i dobar je primer za izučavanje različitih programerskih tehnika i alata za analizu. Evo kako možemo formalno definisati problem uređivanja (sortiranja) niza brojeva:

Ulaz: niz od *n* brojeva ($a_1, a_2, ..., a_n$)

Izlaz: permutacija ($a_1', a_2', ..., a_n'$) niza na ulazu, tako da je ispunjen uslov $a_1' \le a_2' \le ... \le a_n'$

Na primer, za zadati ulazni niz (31, 41, 59, 26, 41, 58), algoritam za sortiranje daje kao rezultat izlazni niz (26, 31, 41, 41, 58, 59). Ovakav ulazni niz se naziva **primerkom** (instancom) problema sortiranja. U opštem slučaju, **instanca problema** se sastoji od ulaza (koji zadovoljava ograničenja koja nameće definicija problema) potrebnog da se odredi rešenje problema.

Sortiranje je fundamentalna operacija u kompjuterskim naukama, pa je zbog toga razvijen veliki broj dobrih algoritama za sortiranje. Koji je algoritam najbolji u datom slučaju najviše zavisi od broja elemenata koje je potrebno sortirati, nivoa do koga su elementi već sortirani, eventualnih ograničenja vrednosti elemenata i vrste uređaja na kome su podaci smešteni (radna memorija, disk, traka, ...).

Za neki algoritam kažemo da je **korektan** ukoliko za svaki primerak ulaza daje korektan izlaz. Onda kažemo da korektan algoritam rešava dati kompjuterski problem. Ukoliko algoritam ne može u potpunosti da se izvrši za neke primerke ulaza ili daje netačne rezultate, za njega kažemo da je nekorektan. Iako se to ne bi očekivalo, nekorektni algoritmi nekada mogu biti veoma korisni, ukoliko postoji mogućnost kontrole nivoa greške koju prave.

Algoritam se može predstaviti na različite načine: šematski, na engleskom, srpskim ili nekom drugom jeziku, korišćenjem nekog programskog jezika itd. Jedini zahtev koji opis mora da ispuni je da daje precizan opis kompjuterske procedure koju je potrebno izvršiti.

Algoritmi kao tehnika

Pretpostavimo da su kompjuteri beskonačno brzi i da je kompjuterska memorija besplatna. Da li bi imalo razloga izučavati algoritme. Odgovor je da, ako ni zbog čega drugog, a ono zbog toga što je potrebno dokazati da je algoritam moguće izvršiti do kraja i to sa tačnim rešenjem.

Ukoliko su kompjuteri beskonačno brzi, svaki korektni algoritam će dati tačno rešenje. U takvim slučajevima bi se mogao izabrati dobro projektovan i dokumentovan algoritam, ali bi najverovatnije bio izabran algoritam koji je najlakše primeniti. Naravno, kompjuteri mogu biti brzi, ali ne beskonačno brzi. Takođe, memorija može biti jeftina, ali nikada besplatna. Iz tog razloga su brzina izvršavanja algoritma i količina memorije koju on koristi ograničavajući faktori. Ovi resursi moraju biti mudro korišćeni, a algoritmi koji su efikasni u pogledu korišćenja vremena i prostora u mnogome mogu pomoći.

Efikasnost

Različiti algoritmi namenjeni rešavanju jednog istog problema se često mogu znatno razlikovati u pogledu njihove efikasnosti. Ove razlike mogu imati mnogo veći značaj od razlika u hardveru i softveru.

Kao jedan od primera u narednom poglavlju videćemo dva algoritma za sortiranje. Prvi, poznat kao **Insertion sort**, troši približno $c_1 n^2$ vremena za sortiranje n elemenata, pri čemu je c_1 konstanta koja ne zavisi od n. To praktično znači da se za sortiranje troši vremena proporcionalno n^2 (na primer, za deset puta više elemenata je potrebno 100 puta više vremena). Drugi algoritam, **merge sort**, troši približno $c_2 n \log_2 n$, pri čemu je c_2 konstanta koja ne zavisi od n. Insertion sort obično ima manji konstantni faktor od merge sorta ($c_1 < c_2$). Videćemo da konstatni faktori znatno manje utiču na vreme izvršavanja od veličine ulaza n. Dok merge sort ima faktor $\log_2 n$ u izrazu za vreme izvršavanja, Insertion sort ima faktor n, što je znatno više. Iako je Insertion sort obično brži od merge sorta za male veličine ulaza, kada veličina ulaza postane dovoljno velika, prednost merge sorta ($\log_2 n$ protiv n), će znatno prevazići razliku u konstantnim faktorima. Bez obzira koliko je c_1 manje od c_2, uvek će postojati prelomni trenutak u kome će merge sort postati brži.

Razmotrimo konkretan primer u kome brži kompjuter (kompjuter A) izvršava Insertion sort, dok sporiji kompjuter (kompjuter B) izvršava merge sort i da svaki od njih treba da sortira niz od milion brojeva. Pretpostavimo da kompjuter A izvršava milijardu instrukcija u sekundi, a kompjuter B izvršava samo deset miliona instrukcija u sekundi, što znači da je kompjuter A 100 puta brži od kompjutera B. Da razliku učinimo još dramatičnijom, pretpostavimo da programski kod za Insertion sort na mašinskom jeziku najboljeg programera na svetu zahteva $2n^2$ instrukcija za sortiranje n brojeva ($c_1 = 2$). Sa druge strane, merge sort za kompjuter B je napisao prosečan programer korišćenjem jezika višeg nivoa i neefikasnog kompajlera, tako da rezultujući kod zahteva $50 n \log_2 n$ instrukcija ($c_2 = 50$). Za sortiranje jednog miliona brojeva kompjuter A će potrošiti

$$\frac{2 \cdot \left(10^6\right)^2 \text{ instrukcija}}{10^9 \text{ instrukcija/sekundi}} = 2000 \text{ sekundi}$$

dok kompjuteru B treba

$$\frac{50 \cdot 10^6 \log_2 10^6 \text{ instrukcija}}{10^7 \text{ instrukcija/sekundi}} \approx 100 \text{ sekundi}$$

Korišćenjem algoritma čije vreme izvršavanja raste sporije, čak i sa lošim kompajlerom kompjuter B izvršava isti zadatak 20 puta brže od kompjutera B. Prednost merg sort algoritma još više dolazi do izražaja prilikom sortiranja deset miliona brojeva. Dok je Insertion sortu potrebno približno 2.3 dana, merge sort isti zadatak izvrši za manje od 20 minuta. Generalno, sa povećanjem veličine problema povećava se i relativna prednost merge sorta.

Algoritmi i druge tehnologije

Prethodni primer pokazuje da su i algoritmi, kao i kompjuterski hardver, takođe **tehnologija**. Ukupne performanse sistema zavise od izbora efikasnih algoritama isto toliko kao i od izbora brzog hardvera. Baš kao što je pravljen ekstremno brz napredak u razvoju drugih kompjuterskih tehnologija, pravljen je napredak i u algoritmima.

Iako postoje aplikacije koje ne zahtevaju eksplicitno algoritamski sadržaj na aplikacionom nivou (na primer, jednostavne web aplikacije), većina njih zahtevaju izvestan stepen algoritamskog sadržaja. Na primer, posmatrajmo web-bazirani servis koji određuje putanju kako doći od jednog mesta do drugog (ili GPS uređaji za navigaciju). Njegova implementacija bi se zasnivala na brzom hardveru, grafičkom korisničkom okruženju, mrežom i najverovatnije nekom orijentacijom objekta. Međutim, ova aplikacija bi zahtevala takođe algoritme za određene operacije, kao što su pronalaženje puta (korišćenjem algoritma za određivanje najkraćeg puta), senčenje mapa i interpolaciju adresa. Sada je jasno da se i aplikacije koje ne zahtevaju algoritamski sadržaj na aplikativnom nivou zapravo oslanjaju upravo na moćne algoritme.

Napredovanjem računara oni se koriste za rešavanje sve većih problema. Kao što smo mogli videti u prethodnom tekstu, sa povećanjem veličine problema razlika između algoritama postaje sve očiglednija. Posedovanje dobrog algoritamskog znanja i tehnika je jedna od karakteristika koja razdvaja vrhunske

programere od početnika. Korišćenjem modernih kompjuterskih tehnologija moguće je izvršiti određene zadatke bez previše poznavanja algoritama, ali sa dobrim znanjem algoritama se može uraditi mnogo više.

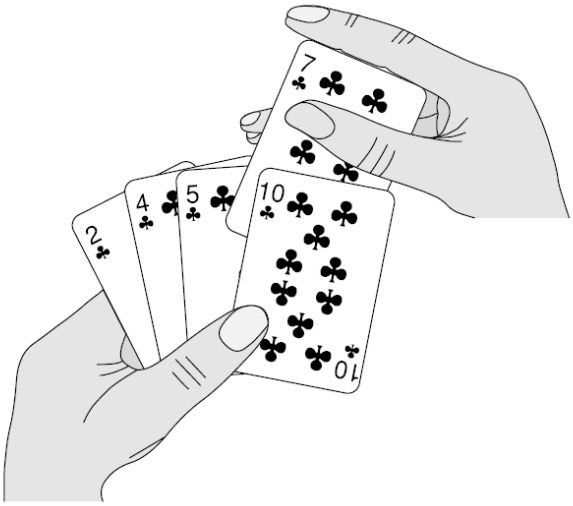
Analiza i dizajn algoritama

Insertion sort

Prvi algoritam koji ćemo razmatrati, Insertion sort, se koristi za rešavanje problema uređivanja niza u neopadajući raspored. Kao što je već pomenuto, ulaz u algoritam je niz od n brojeva, a izlaz permutacija tih brojeva takva da važi relacija $a'_1 \leq a'_2 \leq ... \leq a'_n$. Brojevi koje želimo da sortiramo se još nazivaju i ključevi.

Sve algoritme koje budemo analizirali pisaćemo u **pseudokodu**, koji je sličan programskim jezicima C i Pascal. Za razliku od stvarnog koda napisanog u nekom programskom jeziku, u pseudokodu se mogu koristiti bilo kakve metode izražavanja, koje jasnije i konciznije predstavljaju dati algoritam. Ponekad je delove algoritma najlakše objasniti na Srpskom jeziku, koji se može kombinovati sa stvarnim programskim jezikom. Druga razlika između pseudokoda i stvarnog koda je u tome što pseudokod najčešće ne ulazi u detalje softverskog inženjeringa u cilju što kraćeg i jasnijeg predstavljanja algoritma.

Krenimo od **Insertion sort**-a, koji je efikasan algoritam za sortiranje malog broja elemenata. Insertion sort funkcioniše slično kao što većina ljudi sortira karte za igranje u ruci. Počinjemo sa praznom levom rukom i špilom karata okrenutih licem ka stolu. Uzimamo jednu po jednu kartu sa stola i umećemo je na odgovarajuće mesto u levoj ruci. Da bismo našli odgovarajuće mesto, upoređujemo datu kartu sa svim kartama koje su već u ruci, sa desna na levo, kao što je prikazano na Slici ###. Sve vreme, karte u levoj ruci su sortirane i to su zapravo karte sa vrha špila na stolu.



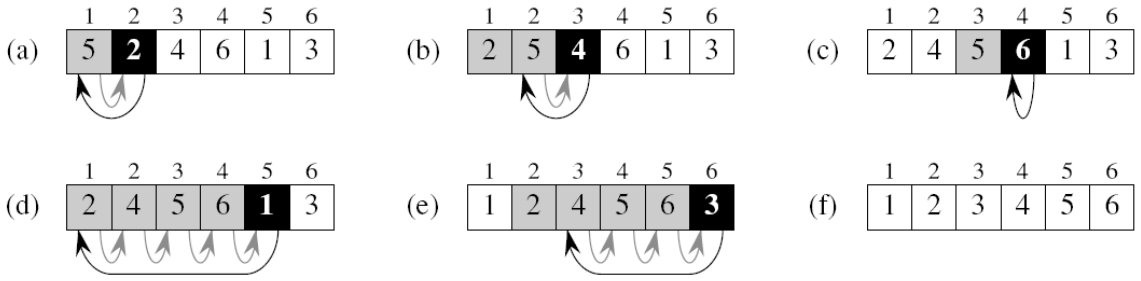
Slika ###. Sortiranje karata korišćenjem Insertion sorta

Pseudokod za Insertion sort je predstavljen funkciom InsertionSort, koja kao parametar uzima niz A dužine n, koji je potrebno sortirati. Ulazni niz se sortira **u mestu**, što znači da se vrši samo premeštanje brojeva unutar niza A, bez korišćenja dodatnog niza. Po završetku funkcije, u nizu A se nalaze brojevi sortirani u neopadajućem redosledu.

```
InsertionSort(A)
   for j = 2, duzina(A)
      key = A[j]
      // Umetanje broja A[j] u sortirani podniz A[1..j-1]
      i = j-1
      while i>0 and A[i]>key
         A[i+1] = A[i]
         i = i-1
      A[i+1] = key
```

Invarijante petlje i korektnost Insertion sorta

Na Slici ### je prikazano kako ovaj algoritam funkcioniše za $A=(5,2,4,6,1,3)$. Indeks j označava trenutnu kartu koja se umeće u ruku. Na početku svake iteracije spoljašnje **for** petlje, indeksirane sa j, podniz $A[1..j-1]$ se sastoji od trenutno sortiranih karata, a elementi $A[j+1..n]$ odgovaraju špilu karata koje su još uvek na stolu. Elementi $A[1..j-1]$ su zapravo elementi koji su originalno bili na pozicijama od 1 do $j-1$, ali sada u sortiranom redosledu. Ovu osobinu podniza $A[1..j-1]$ uzimamo kao **invarijantu petlje (**važi u svakoj iteraciji petlje**)**: *Na početku svake iteracije **for** petlje, podniz $A[1..j-1]$ se sastoji od elemenata koji su originalno bili u podnizu $A[1..j-1]$, ali u sortiranom rasporedu.*



Slika ###. Operacije pri Insertion sortu na nizu $A=(5,2,4,6,1,3)$. U svakoj iteraciji spoljašnje for petlje, crni pravougaonik čuva ključ preuzet iz $A[j]$, koji se zatim poredi sa vrednostima u sivim pravougaonicima sa njegove leve strane. Sive strelice pokazuju vrednosti koje se pomeraju za po jedno mesto u desno, a crne strelice pokazuju gde se umeće ključ.

Invarijante petlje koristimo da bismo lakše razumeli zašto je neki algoritam korektan. Da bismo dokazali korektnost algoritma, moramo pokazati tri osobine invarijanti petlje:

Inicijalizacija: Tačna je pre prve iteracije petlje.

Održanje: Ukoliko je tačna pre neke iteracije petlje, ona ostaje tačna i pre sledeće iteracije.

Okončanje: Kada se petlja okonča, invarijanta daje korisno svojstvo koje pomaže dokazivanju da je algoritam korektan.

Kada su prve dve osobine ispunjene, invarijanta petlje je tačna pre početka svake iteracije. Kao i u slučaju matematičke indukcije, da bismo dokazali da invarijanta važi, pokazujemo da invarijanta važi pre prve iteracije, kao osnovni slučaj, i da se invarijanta održava od iteracije do iteracije. Za razliku od matematičke indukcije gde se induktivni korak koristi beskonačno, u ovom slučaju indukcija se prekida kada se završi petlja, što je zadatak treće osobine.

Razmotrimo tačnost ovih osobina u slučaju Insertion sort-a.

Inicijalizacija: Pokažimo prvo da invarijanta petlje važi pre prve iteracije, kada je $j=2$. U tom slučaju podniz $A[1..j-1]$ se sastoji samo od jednog elementa $A[1]$, koji je zapravo originalni element $A[1]$. Naravno, niz od jednog elementa je svakako sortiran, tako da je dokazano da invarijanta petlje važi pre prve iteracije.

Održanje: Sada treba pokazati da svaka iteracija održava tačnost invarijante petlje. U svakoj iteraciji se elementi $A[j-1]$, $A[j-2]$, $A[j-3]$ i tako dalje, pomeraju za po jedno mesto u desno, sve dok se ne pronadje odgovarajuća pozicija za element $A[j]$, nakon čega se vrši njegovo umetanje u niz. Na ovaj način podniz $A[1..j-1]$ i dalje ostaje sortiran, pa samim tim i druga osobina invarijante petlje važi.

Okončanje: Konačno, razmotrimo šta se dešava kada se petlja završi. Kod Insertion sorta, spoljašnja **for** petlja se završava kada j prekorači dužinu niza n, t.j. kada je $j = n+1$. Zamenom j u definiciji invarijante sa $n+1$, dobijamo da se podniz od $A[1..n]$ se sastoji od elemenata koji su originalno bili u $A[1..n]$, ali sada u sortiranom redosledu. S obzirom da je $A[1..n]$ ceo niz, to znači da je ceo niz sortiran, pa je samim tim i algoritam korektan.

Analiza algoritama

Analiza algoritma podrazumeva predviđanje koje algoritam zahteva. Ono što najčešće želimo da izmerimo je vreme izvršavanja algoritma, a nešto ređe su u pitanju resursi kao što je memorija, protok u komunikacijama ili kompjuterski hardver. Analizom nekoliko algoritama pomoću kojih se može rešiti određeni problem, dolazi se do zaključka koji od njih je najefikasniji u datom slučaju.

Analiza algoritma Insertion sort

Vreme potrebno za izvršavanje funkcije InsertionSort zavisi od ulaza, tako da sortiranje hiljadu brojeva uzima više vremena nego sortiranje tri broja. Štaviše, InsertionSort može različito trajati u zavisnosti od toga u kojoj meri su elementi niza već sortirani. U opštem slučaju vreme potrebno za izvršenje nekog algoritma raste sa porastom veličine ulaza, pa se najčešće vreme izvršavanja izražava kao funkcija veličine ulaza. Da bismo to učinili, neophodno je da pažljivo definišemo termine "vreme izvršavanja" i "veličina ulaza".

Veličina ulaza zavisi od problema koji se posmatra. Za mnoge probleme, kao što su sortiranje i računanje diskretne Furijeove transformacije, najprirodnija mera je broj podataka na ulazu (na primer, dužina niza koji se sortira n). Za mnoge druge probleme, kao što je množenje dva cela broja, najbolja mera veličine ulaza je ukupan broj bitova potreban da se ulazni podaci predstave u binarnom obliku. Nekada je pogodnije veličinu ulaza predstaviti pomoću dva broja. Na primer, ukoliko je ulaz u neki algoritam graf, veličina ulaza može biti predstavljena kao broj čvorova i broj veza u grafu.

Vreme izvršavanja nekog algoritma za određeni ulaz predstavlja broj primitivnih operacija ili koraka koje je potrebno izvršiti. U našim razmatranjima pretpostavićemo da je za svako izvršavanje *i*-te linije u našem pseudokodu potrebno c_i vremena, pri čemu je c_i konstantno.

Predstavimo pseudokod funkcije InsertionSort zajedno sa vremenima potrevnim za izvršenje svake od linija pseudokoda. Neka je t_j ($j = 2,3,...,n$) broj koji označava koliko puta se izvršava provera uslova **while** petlje u linija 5 za datu vrednost j. Kada se **for** i **while** petlja koriste na uobičajeni način, proveravanje uslova se izvršava jedan put više od izvršavanja tela petlje. Komentari u pseudokodu se ne izvršavaju, tako da je vreme njihovog izvršavanja 0.

	InsertionSort(A)	vreme izvršavanja	broj izvršavanja
1	`for j = 2, duzina(A)`	c_1	n
2	`key = A[j]`	c_2	n-1
3	`// Umetanje broja A[j] u sortirani podniz A[1..j-1]`	0	n-1
4	`i = j-1`	c_4	n-1
5	`while i>0 and A[i]>key`	c_5	$\sum_{j=2}^{n} t_j$
6	`A[i+1] = A[i]`	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	`i = i-1`	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	`A[i+1] = key`	c_8	n-1

Vreme izvršavanja algoritma je jednako sumi vremena izvršavanja svake linije. Ukoliko je za izvršenje jedne linije potrebno c_i vremena i ako se izvršavanje ponavlja n puta, onda ta linija učestvuje sa $c_i n$ vremena u ukupnom vremenu izvršavanja algoritma. Da bismo izračunali vreme izvršavanja funkcije InsertionSort, sabiramo proizvode kolona koje označavaju vremena i broj izvršavanja linija, pri čemu dobijamo

$$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n-1)$$

Iako vreme izvršenja algoritma prvenstveno zavisi od veličine ulaza, čak i za istu veličinu ulaza vreme izvršenja može se razlikovati u zavisnosti od podataka koji su uneti. Na primer, kod Insertion sorta najbolji slučaj se javlja kada je niz već sortiran. Tada, pošto i ima inicijalnu vrednost $j-1$, u liniji 5 za svako $j = 2,3,...,n$ važi da je $A[i] \le key$. Samim tim za svako $j = 2,3,...,n$ broj proveravanja uslova unutar **while** petlje je $t_j = 1$, a **vreme izvršavanja u najboljem slučaju**

$$\begin{aligned} T(n) &= c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 (n-1) + c_8 (n-1) = \\ &= (c_1 + c_2 + c_4 + c_5 + c_8) n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Ovako izračunato vreme izvršavanja se može izraziti kao $an + b$, pri čemu su a i b konstante koje zavise od c_i. To zapravo znači da je u najboljem slučaju vreme izvršavanja Insertion sorta linearna funkcija veličine ulaza n.

Ako je niz sortiran u obrnutom redosledu (opadajućem), dobija se najgori slučaj, pri kome moramo porediti svaki element $A[j]$ sa svim elementima u sortiranom podnizu $A[1..j-1]$, tako da je $t_j = j$ za svako $j = 2,3,...,n$. U tom slučaju sume u izrazu za ukupno vreme izvršenja imaju sledeći oblik

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

i

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

a vreme izvršavanja funkcije InsertionSort

$$T(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5\left(\frac{n(n+1)}{2}-1\right)+c_5\left(\frac{n(n-1)}{2}\right)+c_7\left(\frac{n(n-1)}{2}\right)+c_8(n-1)=$$

$$=\left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2+\left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n-(c_2+c_4+c_5+c_8)$$

Dakle, vreme izvršavanja u najgorem slučaju se može predstaviti kao an^2+bn+c, pri čemu su a, b i c konstante koje zavise od c_i, što je praktično kvadratna funkcija veličine ulaza n.

Analiza najgoreg i prosečnog slučaja

U analizi Insertion sorta, razmatrali smo najbolji slučaj, kada je niz na ulazu već sortiran, ali i najgori slučaj, kada je ulazni niz sortiran u obrnutom redosledu. U daljim analizama razmatraćemo samo vreme izvršavanja pri najgorem mogućem slučaju, tj. na najduže vreme izvršavanja za datu veličinu ulaza n. Tri najbitnija razloga za ovakav pristup su:

- Vreme izvršavanja u najgorem slučaju je gornja granica vremena izvršavanja za bilo koji ulaz. To nam garantuje da se algoritam ni u kom slučaju neće izvršavati duže od tog vremena.
- Za neke algoritme najgori slučaj se javlja veoma često. Na primer, prilikom pretraživanja neke baze podataka, najgori slučaj algoritma za pretraživanje će se javljati uvek kada u bazi ne postoji tražena informacija.
- **Prosečni slučaj** je često približno loš isto toliko kao i najgori slučaj. Pretpostavimo da slučajno biramo n brojeva i primenjujemo Insertion sort. Koliko će trajati određivanje mesta u podnizu $A[1..j-1]$ gde treba umetnuti element $A[j]$. U proseku, polovina elemenata podniza $A[1..j-1]$ je manja, a polovina veća od $A[j]$. To znači da se u proseku proverava polovina niza $A[1..j-1]$, što znači da je broj proveravanja $t_j = j/2$. Ako izračunamo vreme izvršenja u prosečnum slučaju, dobićemo da je ono kvadratna funkcija veličine ulaza n, baš kao i u najgorem slučaju.

Red rasta

U analizi Insertion sort algoritma, utvrdili smo da se u najgorem slučaju vreme izvršenja može predstaviti kao an^2+bn+c, pri čemu konstante a, b i c zavise od c_i. U pojednostavljenom razmatranju posmatramo samo vodeće članove (u ovom slučaju an^2), s obzirom da su članovi nižeg reda relativno beznačajni. Imajući u vidu da su za velike ulaze konstantni koeficijenti znatno manje bitni od rasta funkcije, takođe ignorišemo i koeficijent ispred vodećeg člana. Sada možemo da kažemo da je vreme izvršenja Insertion sorta u najgorem slučaju $\Theta(n^2)$.

Obično smatramo da je algoritam efikasniji ukoliko ima manji red rasta. Zbog zanemarivanja konstantnih koeficijenata i članova nižeg reda, ova pretpostavka može biti pogrešna za male veličine ulaza. Međutim, za dovoljno velike ulaze, $\Theta(n^2)$ algortiram će u najgorem slučaju raditi mnogo brže od $\Theta(n^3)$ algoritma.

Rast funkcija

Red rasta vremena izvršavanja algoritma definisan u prethodnoj sekciji nam daje jednostavan način za opisivanje efikasnosti algoritma i mogućnost da izvršimo njegovo poređenje sa drugim alternativnim algoritmima. Kada veličina ulaza n postane dovoljno velika, Merge sort sa svojim vremenom izvršavanja u najgorem slučaju $\Theta(n\log_2 n)$ postaje superioran u odnosu na Insertion sort, čije je vreme izvršavanja u najgorem slučaju $\Theta(n^2)$. Iako ponekad možemo izračunati tačno vreme izvršavanja nekog algoritma (kao što je to urađeno za Insertion sort), to najčešće nije vredno truda. Za dovoljno velike ulaze, konstante i članovi nižeg reda se mogu zanemariti u odnosu na član najvišeg reda. Kada posmatramo samo dovoljno velike ulaze, da bismo odredili red rasta vremena izvršavanja, mi zapravo proučavamo **asimptotsku**

efikasnost algoritama. U opštem slučaju, algoritam koji ima veću asimptotsku efikasnost je najbolji izbor u svim slučajevima osim za veoma male ulaze.

Asimptotska notacija

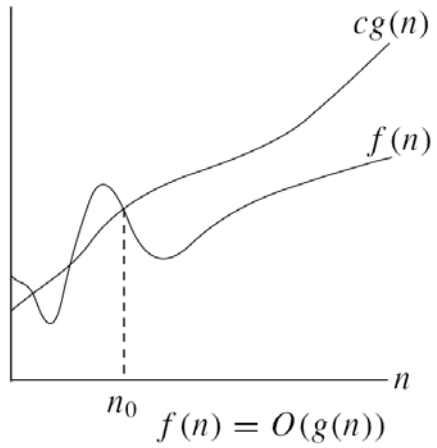
U praksi se najčešće koriste Θ, O i Ω notacija za izražavanje kompleksnosti algoritama. U nastavku ćemo razmotriti samo najbitnije karakteristike *O*-notacije.

O-notacija

Kada posmatramo samo gornju asimptotsku granicu, koristimo *O*-notaciju.

- Za datu funkciju $g(n)$, sa $O(g(n))$ označavamo skup funkcija $f(n)$, ukoliko postoje pozitivne konstante c i n_0 takve da važi $0 \le f(n) \le c \cdot g(n)$ za svako $n \ge n_0$.

O-notacija nam zapravo daje gornju granicu funkcije $f(n)$, kao što je prikazano na Slici ###.



Slika ###. O-notacija daje gornju granicu funkcije u odnosu na konstantan faktor

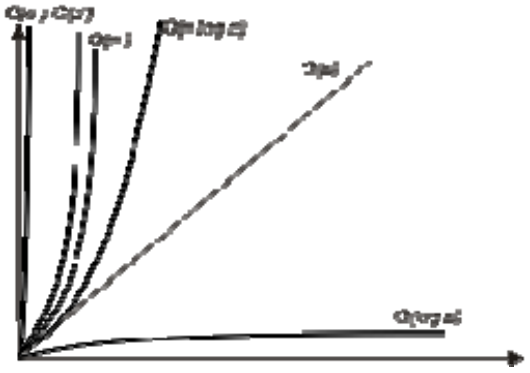
Ukoliko funkcija $f(n)$ pripada skupu $O(g(n))$, onda to znači da je za sve vrednosti n desno od n_0, vrednost funkcije $f(n)$ manja ili jednaka vrednosti funkcije $g(n)$ pomnožene konstantnim faktorom c.

Primer

$n^2, 3n^2, n^2 - 5000, 5n^2 + 3n, \ldots \in O(n^2)$

Korišćenjem *O*-notacije, vreme izvršavanja algoritma često možemo odrediti jednostavnim posmatranjem strukture algoritma. Na primer, dvostruka ugneždena petlja u Insertion sort algoritmu, navode na zaključak da je u najgorem slučaju vreme izvršenja $O(n^2)$. Kao što smo mogli videti u prethodnoj sekciji, vreme izvršavanja Insertion sort algoritma u slučaju kada je ulazni niz već sortiran je $O(n)$. Međutim, s obzirom da vreme izvršavanja zavisi i od ulaza u algoritam, kada kažemo da je "vreme izvršavanja $O(n^2)$", to znači da postoji funkcija $f(n)$ koja pripada skupu $O(n^2)$, takva da za bilo koju vrednost n, vreme izvršavanja za taj ulaz će biti manje od $f(n)$. Zato kažemo da je u najgorem slučaju vreme izvršavanja $O(n^2)$.

Pored stepenih funkcija, kompleksnost algoritama može biti izražena korišćenjem i drugih matematičkih funkcija, kao što su eksponencionalne i logaritmaske. Na Slici ### je prikazan odnos brzina rasta pojedinih funkcija pri povećanju veličine ulaza n.



Slika ###. Odnos brzina rasta funkcija