Složenost algoritama

Kako nisam znao da ću predavati složenost (ustvari, nisam znao ni da ću predavati...), predavanje nije ispalo kako je trebalo... Ali, već sam tamo rekao da ću napraviti nekakav tutorial za određivanje složenosti algoritama (ako uhvatim vremena (tj. koliko mi fiz dopusti) napravim i tutorial o hash-u... mislim da bi to bilo daleko korisnije od ovoga o složenosti) :).

Prije nego krenete čitati tutorial, morate znati:
1°
Složenost algoritma se određuje da bi **od prilike** dobili informaciju o upotrebljivosti našeg programa.
Npr. ima li smisla pustiti program u rad ako mu za obradu dnevnih podataka trebaju 2 dana?

2°
Primarno nas zanima **big O notation** (način bilježenja složenosti algoritma).
Određivanje asimptotske složenosti se samo tražila u blicu (žao mi je što nisam ovo uspio napraviti pred blic... ali Čupićev domaći (Java) mi je uzeo cijeli četvrtak i petak ujutro :(... a na massinstrukcijama nije bilo dovoljno vremena da objasnim detalje (a i bez detalja sam uzeo previše vremena :( )).

3°
Da bi odredili koliko će se od prilike izvoditi neki program na nekom računalu za bilo koje ulazne parametre, vi morate:

- Odrediti složenost programa (njegovih algoritama)
- Izvrtiti program za neki točno određeni ulazni parametar
- Izvesti aposteriori analizu (programski izvedena analiza koja "štopa" koliko se izvodio neki program)

Kad imate te podatke, lako je uvrštavanjem novih parametara, bez ponovnog pokretanja programa, odrediti koliko će se od prilike program izvoditi na tom računalu za te nove parametre.

4°
Sve se ovo radi **OD PRILIKE**. Rade se grube aproksimacije...
Jer... npr. ako imate n^5 + n^3, a n vam je neki jako veliki prirodni broj, tad će vam n^3 biti zanemarivo mala veličina u odnosu na n^5.
Ako pogledate grafove vremena izvršavanja u odnosu na ulazni podatak, biti će vam jasnije zašto se sve može zaokružiti (čak i koeficijent uz vodeći član polinoma koji opisuje složenost funkcije).

5°
Složenost određujete u ovisnosti samo o parametru koji utječe na različit broj izvršavanja funkcije.
Ako imate

Kod:

```
void f(int n, int k, int x) {
        int i;
        for (i = 0; i < k; i++) {
                printf("%d\n", n+i);
                if (x + i == 10) break;
        }
}
```

Znači, složenost ovisi o k, ali broj izvršavanja može ovisiti i o varijabli x.

Znači:
U najgorem slučaju je O(k) (npr. x > 10)
U prosječnom je O(k/2) (npr. x+k/2 = 10)
U najboljem je O(1) (x = 10)

6°
I meni je ovo novo... tako garantiram da će u nastavku biti grešaka (da se bar malo ogradim od kritika ;D). Zamolio bi nekoga sa 2. godine da prođe malo ovaj tutorial i javi o greškama...

POČETAK tutoriala:
Što se tiče reda složenosti, vrijedi:

$$O(1) < O(\log n) < O(\log_2 n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

Vas ne zanima koliko operacija (zbrajanja, množenja, ispisivanja u file, čitanja, ...) program obavi, ili red veličine rezultata, već samo **koliko se puta neka petlja (ili rekurzija) izvršila**.

O(1):
Algoritam se izvodi trenutno za bilo koji parametar.
Neke funkcije složenosti O(1):
Kod:

```
void f(int n) {
        int a;
        n += 5;
        a = n * sqrt(n) - pow(n,5);
        printf("manje bitno...");
        return;
}

int neparan(int broj) {
        return broj % 2;
}

void ispisi100puta(char *tekst) {
        int i;
        for (i = 1; i <= 100; i++) {
                printf("%s\n", tekst);
        }
}
```

Prepoznajete je po: Tome što se program vrti samo jednom, ili konstantan broj puta.

O(log*n*):
Logaritamska složenost. Funkcija logaritamske složenosti svakom iteracijom (izvedbom/"izvrčenjem" ;D) ulazni problem smanji za red veličine baze.
Neke funkcije složenosti O(logn):
Kod:

```
int zbrojZnamenki (int broj) {
        long zbroj = 0;
        while (broj > 0) {
                zbroj += broj % 10;
                broj /= 10;
        }
}

/* Složenost je O(log(broj)) sa bazom 10.
   Svakom novom iteracijom, funkcija je složenost ulaznog problema
smanjila za 10 veličina */
```

Prepoznajete je po: dijeljenju ulaznog parametra sa nekom konstantom.

O(*n*):
Linearna složenost. Funkcija izvede oko n iteracija.
Neke funkcije složenosti O(n):
Kod:

```
void linearno(int k) {
        int i;
        for (i = 1; i <= k; i++) {
                printf("nešto\n");
        }
}
/* Složenost je O(k) */

// rekurzivna izvedba gornjeg programa
void linearno_r(int k) {
        if (k == 0) return;
        printf("nešto\n");
        linearno_r(k - 1);
}

// izvedba gornjeg programa beskonačnom while petljom
void linearno_w(int k) {
        while (1) {
                printf("nešto\n");
                k--;
                if (k == 0) break;
        }
}
void lin_zbroj (int s) {
        int i;
        for (i = 1; i <= s; i++) {
                printf("nešto ");
        }

        for (i = 1; i <= s; i++) {
                printf("novo\n");
        }
}
/* Ovdje se složenosti zbrajaju. Znači, funkcija će se izvesti 2*s puta,
ali gledajući big O notaciju, to je opet O(s) */
```

Prepoznajete je po: Jednoj for petlji / rekurziji kojoj je konvergencija (n+-1).
Uostalom... ovo je jednostavno i provjeriti na datom primjeru... samo izvrtite program (u glavi/na papiru) i linearnost je brzo uoči.

O(*n*log*n*):
"Logaritamsko-linearna" složenost.
Neke funkcije složenosti O(n*logn):
Kod:

```
/* Funkcija koje ispisuje zbroj znamenaka svakog broj od 1 do n */
void ispis_zbroja_znam(int n) {
        int zbroj = 0, i;
        if (n != 0) ispis_zbroja_znam(n - 1);
        while (n != 0) {
                zbroj += n % 10;
                n /= 10;
        }
        printf("%d\n", zbroj);
}

/* Analiza složenosti:
Funkcija se rekurzivno vrti n puta (konvergencija je n-1; n->0).
Složenost unutarnje while petlje je logn (baza je 10).
Pri prvom izvođenju n će biti jednak 1 (rekurzija prvo propada do n=0, a
za n=0 while petlja neće izvoditi, te se petlja izvodi tek kad n dođe u
1),
znači petlja se izvodi log(1) puta.
n = 1 -> petlja se izvodi log(1) puta.
n = 2 -> petlja se izvodi log(2) puta.
n = 3 -> petlja se izvodi log(3) puta.
...
n = n -> petlja se izvodi log(n) puta.
Budući da to možemo zbrojiti jer su petlje međusobno neovisne, dobivamo:
log(1) + log(2) + log(3) + ... log(n) =
= log(1*2*3*...*n) = log(n!), a kako wikipedia kaže:
"From Stirling's approximation we know that log(n!) is asymptotically
equal to n*log(n)" :).
Analiza je gotova :D. Istina, wikipediu nećete moći koristiti na MI ;D,
ali većinu stvari iz analize složenosti možete bubnuti,
napisati ono što vam "se čini točnim" (najozbiljnije... jer su
aproksimacije tako grube da je vjerojatnost da pogriješite minimalna).
*/
```

Dalje ne mogu naći ni jedan smislen primjer, a da je dovoljno jednostavan...
Prepoznajete je po: Tome da imate jednu for petlju (ili while, ili rekurziju) koja će se odviti n puta, a pri svakom izvođenju će raditi neke operacije čija je ukupna složenost O(logn).

O(n^2):
Kvadratna složenost. Funkcija izvede oko n^2 iteracija.
Neke funkcije složenosti O(n^2):
Kod:

```
void kvadratno(int k) {
        int i, j;
        for (i = 1; i <= k; i++)
                for (j = 1; j <= k; j++)
                        printf("nešto\n");
}

/* Složenost je O(k) */

// rekurzivna izvedba gornjeg programa
void kvadratno_r(int k) {
        int j;
        if (k == 0) return;
        for (j = 1; j <= k; j++)
                printf("nešto\n");
        kvadratno_r(k - 1);
}

void red_unatrag(int k) {
        int i, j, suma;
        for (i = 1; i <= k; i++) {
                suma = 0;
                for (j = i; j <= k; j++) {
                        suma += j;
                }
                printf("%d + ", suma);
        }
}
/* za k = 7 ispisuje:
7 + 6 + 5 + 4 + 3 + 2 + 1
Znači, unutarnja petlja se izvrtila prvo k puta, zatim k-1 puta, zatim k-
2 puta, čija nam suma daje:
k*(k-1)/2 ~ (k^2)/2, pri čemu koeficijent zanemarujemo i dobivamo O(k^2).
```

Prepoznajete je po: Dvije for petlji / rekurziji i jednoj for petlji kojoj je konvergencija (n+-1).

O(2^n):
Eksponencijalna složenost. Ovu složenost će te teško dobiti bez rekurzije (iako se svaka rekurzija može zapisati nerekurzivno... ali... to nema smisla ni pokušavati...).
Rezultat izvođenja obično možete zapisati grafički u obliku (govorimo o 2^n... 3^n ili nešto više je naravno teže za nacrtati) (potpuno binarno stablo (tj. binarno stablo koje može ne imati potpunu popunjenost samo u najnižoj razini)):
Kod:

```
       14
      /  \
     2    11
    / \   / \
   1  3  10  30
         /  /
        7  40
```

Neke funkcije složenosti $O(2^n)$:
Kod:

```
/* Rekurzivno računanje sume reda:
f(0) = 1
f(1) = 1
f(2) = 5
f(n) = f(n-1)*(1/2)*f(n-3) - f(n-2)
*/
int f(int n) {
        if (n == 0) return 1;
        else if (n == 1) return 1;      // sve sam raspisao radi jasnijeg
prikaza
        else if (n == 2) return 5;
        else return f(n-1)*(1/2)*f(n-3) - f(n-2);
}


/* Mali Ferko (;D) želi saznati imena svojih predaka unazad nekoliko
generacija.
Da bi pomogli Ferku, mladi, nadobudni FER-ovci su stvorili ogromnu bazu
predaka (ključ pretrage je ime djeteta, i recimo da su imena unikatna,
poput Ferka ;D), za čiju pretragu se koriste 2 funkcije:
char *mama(char *ja);   // Vraća ime mame od *ja
char *tata(char *ja);   // Vraća ime tate od *ja
Koje pretragu rade trenutno, tj. u složenosti O(1).
Zatim su napisali program koji ispisuje pretke (ne ide se po nekakvom
prirodnom redoslijedu, jer bi za neki uređeniji ispis program bio daleko
složeniji)
*/

void ispisi_pretke(char *ja, int dubina) {
        if (dubina == 0) return;
        printf("%s - %s", mama(ja), tata(ja));
        ispisi_pretke(mama(ja), dubina - 1);
        ispisi_pretke(tata(ja), dubina - 1);
}

/* Složenost ovog algoritma je
2^(dubina)
Jer, rekurzija linearno konvergira (dubina->0).

Ovime bi mogli nacrtati stablo koje bi izgledalo od prilike ovako (jer
nismo ispisali korijen (tj. samog Ferka)):
2^0       -             <- Mali Ferko
         /  \
2^1     M1    T1        <- Mama i tata
       / \   / \
2^2   M2  T2 M2 T2      <- Djed i baka

(dubina = 2)
Vidite da broj članova razine eksponencijalno raste.
Tako da bi u k-toj generaciji mali Ferko imao 2^k predaka!
*/
```

Prepoznajete je po: Dva ili više poziva iste rekurzivne funkcije (koja linearno konvergira) unutar nje same (u ovom slučaju imamo 2 poziva).

O(*n*!):
Faktorijelna (kombinatorna) složenost (bar mislim da se tako kaže...). Algoritmi ove složenosti uglavnom služe za rješavanje nekih kombinatornih problema, npr. pronalazak svih permutacija nekog skupa (skup S = {a,b,c}; permutacije tog skupa: Pr = {{a,b,c}, {a,c,b}, {b,a,c}, {b,c,a}, {c,a,b}, {c,b,a}} k(Pr) = 6 = 3! = (k(S))!).
Neke funkcije složenosti O(n!):
Kod:

```
/* program ispisuje n! slučajnih kombinacija brojeva od 4 znamenke */
#include <stdlib.h>
$include <time.h>
void fakt_sloz(int n) {
        int i;
        srand(time(NULL));
        if (n == 0) return;
        for (i = n; i >= 1; i++) {
                printf("%d%d%d%d\n", rand() % 10, rand() % 10, rand() %
10, rand() % 10);       // pretpostavimo da funkcija rand() ima složenost
O(1)
                fakt_sloz(n-1);
        }
}
/* Užasno glup program, kojeg nemojte pokretati za neki n > 7, jer će te
se načekati... (nisam pokretao, ali znam da bi već za n=10 bilo
problema...),
ali bitno je da prikazuje primjer faktorijelne složenosti */
```

Prepoznajete je po: Kombinaciji rekurzije i for petlje (napisano samo za ilustraciju... jer, npr. vi svaku for petlju možete prikazati preko while petlje). Vidite da se za iste parametre funkcija poziva višestruko.

<u>Određivanje složenosti u ovisnosti o slučaju</u>:
Ovo je malo teži dio...
Problem je što više ne vrijede tako grube aproksimacije (za nijansu su određenije)... Sve gore navedeno vrijedi za najgori slučaj. Najbolji slučaj je uglavnom trivijalno za odrediti, ali prosječni... malo teže.
Evo rješenja zadatka o složenosti sa međuispita:
Kod:

```
I.
nasao = 0;
for (i=0; i<n; i++) {
   if (polje[i][i] == zadani) {
      nasao = 1;
      break;
   }
}


II.
nasao = 0;
for (i=0; i<n; i++) {
   for (j=0; j<n; j++) {
      if (i==j && polje[i][j]==zadani)
         nasao = 1;
   }
}

Odrediti apriornu složenost oba odsječka za najbolji, najgori i prosječni
slučaj u ovisnosti o broju redaka 2D polja (n). Detaljno objasniti svoje
odgovore.
```

I)
Najbolji slučaj: O(1)
Objašnjenje: U najboljem slučaju, zadani element će se nalaziti na polje[0][0], te će nas break; naredba izbaciti iz petlje.

Prosječni slučaj: O(n/2)
Objašnjenje: U prosjeku ćemo zadani element nalaziti na sredini polja, tj. oko polje[n/2][n/2]

Najgori slučaj: O(n)
Objašnjenje: U najgorem slučaju, zadani element će se nalaziti na polje[n-1][n-1] ili se neće nalaziti na dijagonali. Petlja će se izvrtiti n puta.

II)
Najbolji slučaj: $O(n^2)$
Prosječni slučaj: $O(n^2)$
Najgori slučaj: $O(n^2)$
Objašnjenje: Druga petlja će se uvijek izvrtiti n^2 puta, jer nema naredbe kojom bi joj prekinuli ciklus (druga se vrti n puta za svako izvođenje prve).

Kad se analizira složenost, prvo se gleda kako se program izvodi, tj. zbog čega dolazi do većeg broja iteracija (izvršavanja). Je li to zbog neke for petlje, while petlje, rekurzije, ...

Zatim, se gleda ima li kakva naredba koja će utjecati na tok izvršavanja programa, npr. break naredba, ili određeni return kod rekurzije.

Ukoliko postoji takvo nešto, moguće je da će program imati različita vremena izvođenja za različite podatke iste složenosti (tj. nije svejedno ako se ima sortirano polje od 100 elemenata i nesortirano polje od 100 elemenata).
Tu se onda može promatrati najbolji slučaj, najgori i prosječni (inače su sva tri slučaja iste složenosti).

Najgori slučaj je objašnjen u prvom dijelu tutoriala.

Najbolji slučaj:
Najbolji je ona kombinacija ulaznih parametara određene složenosti koja će nam najbrže vratiti rješenje (standardan primjer je traženje nekog broja u nizu brojeva; ti je najbolji slučaj kad se traženi broj pojavi kao prvi element polja (naravno, ukoliko krećemo tražiti od početka polja)).
To riješite na način da pogledate kod, uvrstite neki parametar za koji će se program izvršiti s najmanje iteracija i taj parametar je najbolji slučaj.
Naravno, ako imate:
Kod:

```
void f(int n, int k) {
        int i, r;
        r = rand() % n;
        for (i = 0; i < n; i++) {
                if (k + i == r) break;
                else printf("%d", i);
        }
}
```

Vi nećete za n uvrstiti 0, već reći da je nastupio slučaj kad je k+0 = r. (rješenje: O(1))

Prosječni slučaj:
Ne znam kako da ovo opišem... ali evo po primjerima:
Kod:

```
void f(int n, int *polje) {
        int i;
        for (i = 0; i < n; i++) {
                if (*(polje + i) != 3) printf("%d\n", *(polje + i));
                else break;
        }
}
/* Ova petlja će se izvršiti ili 0 ili 1 ili 2 ili ... ili n puta.
Broj izvršavanja ovisi isključivo o sadržaju niza *polje (kojeg uzimamo
da je slučajan).
Gledajući taj niz izvršavanja (0, 1, 2, ..., n), možemo uzeti da je
prosječan broj izvršavanja: n/2.
Znači, složenost je O(n/2).
*/


/* Ako imamo dvodimenzionalno polje n x n */
void f(int n, int *polje, int maxstup) {
        int i, j;
        for (i = 0; i < n; i++) {
                for (j = 0; j < n; j++) {
                        if (*(polje + i*maxstup + j) != 3) printf("%d\n",
*(polje + i));
                        else break;
                }
        }
}
/* Sad, prva petlja će se uvijek izvršavati n puta,
ALI, druga će se izvršavati dok god ne naleti na vrijednost 3 unutar
polja.
Ako promatramo drugu petlju, i idemo načinom razmišljanja iz prethodnog
zadatka,
dobivamo da će prosječno vrijeme izvršavanja biti:
n/2 + n/2 + n/2 + n/2 + ... + n/2 (n pribrojnika)
Znači n*(n/2), a to je (1/2)*n^2.
*/


/* Ako funkciju još jednom promjenimo, imamo: */
void f(int n, int *polje, int maxstup) {
        int i, j, kraj = 0;
        for (i = 0; i < n; i++) {
                for (j = 0; j < n; j++) {
                        if (*(polje + i*maxstup + j) != 3) printf("%d\n",
*(polje + i));
                        else {
                                kraj = 1;
                                break;
                        }
                }
                if (kraj) break;
        }
}
/* Sad ćemo u najgorem slučaju imati n^2 iteracija, u najboljem 1, a u
prosječnom n/2.
*/
```

Kod logaritamskih, eksponencijalnih i faktorijelnih složenosti, određivanje prosječne složenosti je malo teži posao za koji nemam ni vremena, ni znanja (ni iz ASP-a, ni iz matematike)...
Tako da... ako vam dođe tako nešto... ako ne želite gubiti vrijeme... bubnite rješenje ;D.

Mogli bi se potruditi i sve zadatke + malo teorije ukomponirati u zbirku (koja bi se stalno punila novim zadatcima, i uvijek bi bila na raspolaganju za skinuti (s najnovijim zadatcima)).
I, mogli bi zamoliti Botičkog i Čupića za recenziju...
Nakon toga bi bilo dovoljno dobro organiziranog materijala za učenje ASP-a.

Sretno svima na 1. MI!

Ivan