Algoritmernas effektivitet

Ett problem och en algoritm

Olika problem kan lösas med datorns hjälp. Det går att hantera olika uppgifter och uppnå önskade resultat. Ett program skrivet i ett lämpligt programmeringsspråk kan köras på en dator, och på så sätt kan något problem lösas.

För att kunna lösa ett problem, måste först problemet analyseras och ett lämpligt sätt att lösa det hittas. Vägen till lösningen specificeras som en serie operationer i samband med aktuella data. Man utgår ifrån givna uppgifter och utför alla nödvändiga operationer i en viss ordning. Olika mellanresultat skapas, sparas och används på vägen. På slutet är alla planerade operationer klara och ett visst jobb utfört. Eventuellt erhåller man ett visst resultat.

En detaljplan som skapas för att lösa ett visst problem kalas för algoritm. I en algoritm specificeras vilka operationer som behöver utföras, med vilka data, och i vilken ordning. En algoritm har sin input, som omfattar alla de uppgifter som tillförs till algoritmen utifrån. En algoritm startar med vissa uppgifter, och hanterar dessa uppgifter enligt en viss logik. På så sätt löses ett givet problem.

För att kunna lösa ett visst problem, behöver man känna till det aktuella problemområdet. Lösningen baseras alltid på en viss kunskap. Sedan behöver lösningen yttras genom en serie operationer. Man behöver känna till olika tekniker för att kunna designera en lämplig algoritm. För att kunna köra en algoritm på en dator, så behöver algoritmen kodas i ett lämpligt programmeringsspråk. Alla de operationer som är specificerade i algoritmen behöver yttras genom motsvarande operationer i programmeringsspråket. Vid den här översättningen kan hända att flera successiva operationer i algoritmen kodas som en enda operation, eller att en och samma operation i algoritmen avbildas till flera operationer i motsvarande program.

Exempel

Skapa en algoritm som beräknar längden av en rätvinklig triangels hypotenusa, om längder på triangelns kateter är 3 och 4.

Enligt Pythagoras sats är summan av kvadraterna på kateterna lika med kvadraten på hypotenusan. Så för att hitta hypotenusans längd, behöver kateternas kvadrater först beräknas och adderas. På så sätt får man hypotenusans kvadrat. Längden av hypotenusan erhålls sedan som roten ur denna kvadrat.

Tillämpning av den här algoritmen på den givna triangeln går så här:

$3^2 = 9$

$4^2 = 16$

$9 + 16 = 25$

$\sqrt{25} = 5$

Alltså är hypotenusans längd 5.

Man ska skilja mellan ett allmänt problem och olika instanser av detta problem. I ett allmänt problem anges inputdata som parametrar. I en instans av detta problem kan inputdata ha konkreta värden. Att hitta en rätvinklig triangels hypotenusa för givna kateter är ett allmänt problem. Problemet har två inputparametrar: längden av den ena kateten och längden av den andra kateten. Dessa parametrar kan betecknas med a och b. Motsvarande algoritm använder dessa parametrar och producerar ett resultat: längden av triangelns hypotenusa. Inuti algoritmen finns hypotenusan som en variabel. Den kan betecknas med c. På så sätt utgår motsvarande algoritm ifrån sina parametrar a och b, och skapar ett resultat som sparas i variabeln c. Det här allmänna problemet har oändlig många instanser. Att hitta längden av hypotenusan i den rätvinkliga triangeln vars kateter är 3 och 4 är exempel på en sådan instans.

Istället för att skapa en algoritm som bara löser en instans av ett problem, så är det mycket bättre att skapa en algoritm som löser motsvarande problem med parametrar. Man skapar en allmän algoritm, som sedan kan tillämpas till alla instanser av det allmänna problemet. I datavetenskap hittar man lösningar för olika allmänna problem, och specificerar motsvarande algoritmer. Man hittar olika algoritmer för sorteringen, för sökningen, för olika beräkningar, för undersökningen av olika vägar i ett nät, o s v.

Exempel

Skapa en algoritm som beräknar längden av en rätvinklig triangels hypotenusa, om längder av triangelns kateter är givna.

Det här problemet är ett allmänt problem. Dess parametrar är längder av triangelns kateter. Dessa parametrar kan betecknas med a och b. En variabel c ska skapas, och värdet på hypotenusans längd ska lagras där. På slutet ska denna variabel innehålla algoritmens resultat. I så fall kan algoritmen matematiskt specificeras så här:

$k1 = a2$

$k2 = b2$

$k3 = k1 + k2$

$c = \sqrt{k3}$

Variablerna k1, k2 och k3 används för att lagra olika mellanresultat (kateternas och hypotenusans kvadrater). Denna algoritm kan kodas som en metod i programmeringsspråket Java, så här:

```
public static double hypotenusa (double a, double b)
{
    double    k1 = a * a;
    double    k2 = b * b;
    double    k3 = k1 + k2;
    double    c = Math.sqrt (k3);
    return c;
}
```

Ett programmeringsspråk specificerar ett antal operationer, och det är dessa operationer som används när en algoritm kodas. Operationer angivna i algoritmen avbildas till motsvarande operationer i programmeringsspråket. Ibland går det att binda flera operationer och koda dem i ett enda steg. T ex kan föregående algoritm kodas även så här:

```
public static double hypotenusa (double a, double b)
{
    double    c = Math.sqrt (a * a + b * b);
    return c;
}
```

För ett och samma problem kan flera olika sätt att lösa det hittas. Det kan finnas flera algoritmer för ett och samma problem. För att kunna välja rätt algoritm i en konkret situation, behöver de olika algoritmerna kunna utvärderas på något sätt och jämföras med varandra. En algoritm kan karakteriseras på olika sätt. Först och främst ska en algoritm vara korrekt – den ska göra precis det som man förväntar sig. Sedan kan en algoritm vara enkel och naturlig, eller komplicerad och svårt att förstå. Vissa algoritmer använder avancerade kunskaper från olika vetenskapsområden, eller är konstruerade enligt avancerade designtekniker. En algoritm kan vid exekveringen kräva mer eller mindre tid, eller mer eller mindre minne. Algoritmerna kan ha olika tidskomplexitet och minneskomplexitet.

En algoritm kan analyseras på olika sätt. En hel del analyser koncentrerar sig på den tid och det minne som krävs för algoritmens exekvering. En hel teori, så kallade komplexitetsteori, har utvecklats för att analysera tids- och minnesbehov för de olika algoritmerna. Man inför och använder olika mått för att karakterisera dessa behov för olika storlekar på input.

En algoritms tidskomplexitet

En algoritm består av ett antal operationer, som utförs i samband med inputdata. När alla dessa operationer blir klara, så blir det motsvarande problemet löst. Man erhåller ett visst resultat, eller får något jobb utfört. Men exekveringen av alla dessa operationer kan ta en betydande tid. Den tid som kan accepteras beror på omständigheter, och kan variera i stora gränser. I vissa situationer kan långa exekveringstider tolereras, i andra situationer måste jobbet utföras mycket snabbt. Därför måste finnas något sätt att uppskatta exekveringstiden för de olika algoritmerna. Ett mått för algoritmernas tidskomplexitet (eng. time complexity) måste införas och användas.

Den tid som exekveringen av en algoritm tar beror på själva algoritmen. Antalet operationer och deras typer bestämmer algoritmens exekveringstid. Men denna tid beror även på värddatorn, programmeringsspråket, programmeringsstilen och kompilatorn. Inflytandet av dessa fyra faktorer är svårt att beakta i en allmän komplexitetsteori, och därför sätter man fokus på själva algoritmen. Man koncentrerar sig på algoritmens operationer, och deras komplexitet och antal. För att uppskatta

algoritmens komplexitet, väljs den operation som tidsmässigt dominerar i algoritmen. Antalet exekveringar av den här operationen bestäms som funktion av storlek på input, och denna funktion ger algoritmens komplexitet.

Tidskomplexiteten för en algoritm yttras inte i antalet tidsenheter som exekveringen av denna algoritm tar. Den yttras på ett sätt som inte beror på värddatorn, programmeringsspråket, programmeringsstilen och kompilatorn. Tidskomplexiteten för en algoritm specificeras genom en funktion, som ger antalet exekveringar av en elementär operation för olika inputstorlekar. Denna funktion heter komplexitetsfunktion (eng. complexity function). Komplexitetsteorin handlar om olika sätt att bestämma komplexitetsfunktioner för olika algoritmer, och olika sätt att analysera dessa komplexitetsfunktioner.

För att uppskatta en algoritms tidskomplexitet, väljer man en tidsmässigt dominant operation i algoritmen. Denna operation kan vara en enkel operation, t ex en jämförelse eller en tilldelning som upprepas. I andra situationer kan den valda operationen bestå av flera mindre operationer. Man väljer en grupp operationer, en sammansatt operation, och betraktar den som en elementär operation. Tidskomplexiteten yttras sedan genom antalet exekveringar av denna operation för olika inputstorlekar. Det går även att välja olika elementära operationer och på så sätt erhålla olika komplexitetsfunktioner för en och samma algoritm. På så sätt kan man få en bättre uppfattning av algoritmens tidskomplexitet. Det kan underlätta jämförelser av olika algoritmer för ett och samma problem.

Exempel

Bestäm tidskomplexiteten för sorteringsalgoritmen ”utbytessortering” (eng. exchange sort).

Sorteringsalgoritmen ”utbytessortering” tar emot en sekvens med ett antal jämförbara dataenheter, och sorterar dessa dataenheter. Sekvensen med dataenheterna representerar algoritmens input. Antalet dessa dataenheter representerar storlek på denna input.

Man bestämmer först det element i sekvensen som ska ligga på första positionen i sekvensen. Man går genom sekvensen och jämför varje element med det element som finns på första positionen. Så snart ett element som är mindre än det aktuella första elementet påträffas, byter man deras platser. Efter ett fullständigt pass genom sekvensen, ligger minsta element på första positionen. På samma sätt placeras rätt element på andra positionen i sekvensen, därefter placeras tredje element, o s v. Man går alltid genom de element som finns efter den aktuella positionen, och utför utbyten om så behövs.

Nedan följer en möjlig implementering för utbytesalgoritmen. Algoritmen implementeras för heltal, men på ett liknande sätt kan den implementeras även för dataenheter av andra typer, så länge dessa dataenheter kan jämföras med varandra.

```
public static void sort (int[] numbers)
{
    int    number = 0;
    for (int i = 0; i < numbers.length - 1; i++)
        for (int j = i + 1; j < numbers.length; j++)
            if (numbers[j] < numbers[i])
            {
                number = numbers[i];
                numbers[i] = numbers[j];
                numbers[j] = number;
             }

}
```

För att bestämma tidskomplexiteten för denna algoritm, behöver man först skilja en tidsmässigt dominant operation, och räkna sedan hur många gånger den här operationen utförs. Två operationer i algoritmen upprepas. Den ena är elementjämförelse, och den utförs i varje pass genom inre loopen. Den andra operationen som upprepas är positionsutbyte för två element. Denna operation kan utföras eller inte utföras, beroende på resultatet av jämförelsen. Man kan välja vilken som helst av dessa två operationer, och yttra algoritmens komplexitet genom antalet exekveringar av den valda operationen. En bra uppskattning erhålls genom att välja en elementjämförelse som en elementär operation. Elementutbyten kan ändå inte ändra storleksordning av algoritmens komplexitetsfunktion. När man uppskattar en algoritms tidskomplexitet, är man normalt ute efter komplexitetsfunktionens storleksordning. I fall att denna algoritm ska jämföras med en algoritm som har en liknande tidskomplexitet, kan även antalet positionsutbyten bestämmas.

Antalet dataenheter i sekvensen kan betecknas med n. I så fall behöver man bestämma hur många gånger jämförelseoperationen utförs för olika n. Vid första passet genom sekvensen utförs $n - 1$ jämförelser, eftersom element på varje position jämförs med det element som finns på första platsen. I nästa pass jämförs alla de element som ligger efter andra platsen med det element som ligger på andra platsen. Det finns $n - 2$ sådana jämförelser. Antalet jämförelser minskas med 1 i varje ny pass genom inre loopen. Först är det $n - 1$ jämförelser, sedan $n - 2$, sedan $n - 3$, o s v. I sista passet blir det bara 1 jämförelse. Det totala antalet jämförelser ges genom följande summa:

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

Denna summa är $n(n - 1)/2$, och den representerar algoritmens komplexitetsfunktion. Om denna komplexitetsfunktion betecknas med T, så kan man skriva så här:

$$T(n) = n(n - 1)/2$$

Algoritmens tidskomplexitet yttras genom den här komplexitetsfunktionen. Det återstår att analysera denna komplexitetsfunktion och tolka resultatet. Analysen avslöjar att utbytessortering inte är tillräckligt effektiv för stora sekvenser. Det finns snabbare sorteringsalgoritmer. Men denna algoritm är mycket enkel, och kan räcka i vissa situationer.

För att få någon tidsuppfattning, kan man utgå ifrån att den valda jämförelseoperationen kräver någon bestämd tid, t ex en nanosekund. Exekveringstiden kan i så fall uppskattas för olika n genom att multiplicera värdet på komplexitetsfunktionen med den valda tidsenheten. T ex om en nanosekund väljs som tiden och 1000000 som storlek på sekvensen, tar exekveringen av utbytessortering omkring 8 minuter.

Ofta är det lätt att bestämma storlek på input för en algoritm. I en sorteringsalgoritm är det antalet dataenheter som sorteras. Men det kan hända att algoritmen tar flera datasamlingar och bearbetar dem på något sätt. I ett sådant fall yttras algoritmens tidskomplexitet genom en funktion av flera variabler.

Komplexitetsfunktionen har storlek på algoritmens input, och inte själva input, som sin variabel. Men i vissa situationer kan det vara acceptabelt att använda själva input som variabel för motsvarande funktion. En algoritm som bestämmer roten av ett heltal, har detta heltal som sin input, och storlek av detta heltal som storlek på input. Storleken på heltalet kan yttras genom antalet bitar som krävs för att representera detta heltal. Om heltalet är n, så behövs $\log_2 n + 1$ bitar (logaritmen avrundas neråt). Detta är algoritmens storlek på input, och det ska vara komplexitetsfunktionens variabel. Men tillräkliga insikter om algoritmens komplexitet kan erhållas även om själva heltalet n används som variabel. Det går t ex att jämföra två olika algoritmer som beräknar roten genom att använda en sådan funktion som mått.

Komplexitetsteorin gäller i mångt och mycket uppskattningar. När man väljer en elementär operation och räknar antalet exekveringar av den här operationen, försummar man de operationer som utförs bara en gång (eng. overhead operations). Man försummar även de operationer som kontrollerar att den valda elementära operationen exekveras så länge detta behövs (eng. control operations). För tillräckligt stora inputstorlekar tar vanligtvis dessa andra operationer en försumbar tid i jämförelse med den tid som krävs för exekveringen av den valda elementära operationen. Men för mindre inputstorlekar kan relation mellan dessa tider ändras. I så fall kan man ha olika uppskattningar för olika räckvid av inputstorlek. På så sätt kan en algoritm vara bättre en någon annan algoritm för vissa inputstorlekar, och sämre för andra inputstorlekar.

Sämsta, genomsnittliga och bästa fall

För att kunna bestämma en algoritms tidskomplexitet, särskiljs först den operation i algoritmen vars upprepade exekvering tar mest tid. Man bestämmer sedan hur många gånger den här operationen utförs för olika storlek på input. Om detta antal betecknas med T och inputstorlek med n, blir algoritmens komplexitetsfunktion $T(n)$.

I vissa algoritmer beror antalet exekveringar av den valda, elementära operationen inte bara på inputstorlek, utan även på själva input. För en uppsättning inputdata kan denna operation utföras bara några få gånger, men för en annan input kan exekveringen upprepas många gånger. I så fall går det inte att yttra algoritmens tidskomplexitet genom en komplexitetsfunktion som gäller alla möjliga input. Det går inte att bestämma tidskomplexiteten för alla möjliga fall (eng. every-case time complexity). För att kunna beskriva tidskomplexiteten för sådana algoritmer, måste andra mått införas.

När antalet exekveringar av en elementär operation beror även på input, och inte bara på inputstorlek, kan algoritmens tidskomplexitet beskrivas på flera olika sätt. En möjlighet är att bestämma ett maximalt möjligt antal exekveringar av denna operation för en given inputstorlek n. Om detta antal betecknas med W, så blir komplexitetsfunktionen $W(n)$. Denna komplexitetsfunktion beskriver inte algoritmens tidskomplexitet i alla möjliga fall, utan bara i värsta möjliga fall (eng. worst-case time complexity). Komplexitetsfunktionen ger övre gräns på antalet exekveringar. I vissa, kritiska applikationer kan det här måttet vara speciellt användbart.

Ibland beskrivs en algoritms tidskomplexitet med minsta möjliga antalet exekveringar av en elementär operation för en given inputstorlek n. Om detta antal betecknas med B, så blir komplexitetsfunktionen $B(n)$. Denna komplexitetsfunktion beskriver algoritmens tidskomplexitet i bästa fall (eng. best-case time complexity). Den ger nedre gräns på antalet exekveringar. Det är sällan att den här komplexitetsfunktionen används.

Ofta yttrar man en algoritms tidskomplexitet genom ett genomsnittligt antal exekveringar av en elementär operation. Om inputstorlek betecknas med n, och det genomsnittliga antalet med A, så blir komplexitetsfunktionen $A(n)$. Denna komplexitetsfunktion ger algoritmens tidskomplexitet i genomsnitt, när algoritmen exekveras många gånger för alla möjliga input av en given inputstorlek (eng. average-case time complexity). Detta mått kan vara passande i många applikationer, så länge man tillåter att vissa exekveringar kan ta kritiskt långa tider.

För att kunna bestämma det genomsnittliga antalet exekveringar av en elementär operation, behöver man känna sannolikheten att en viss input händer. Vid beräkningen multipliceras antalet exekveringar för just denna input med motsvarande sannolikhet. Komplexitetsfunktionen $A(n)$ blir i så fall summan av dessa produkter för alla möjliga input.

Exempel

Bestäm tidskomplexiteten för sökningsalgoritmen ”sekventiell sökning”.

Sökningsalgoritmen ”sekventiell sökning” tar emot en sekvens med ett antal jämförbara dataenheter, och en ytterligare dataenhet av samma typ. Algoritmen bestämmer om givna dataenheten finns i sekvensen, och i så fall på vilken position. Sekvensen med dataenheterna och den sökta dataenheten representerar algoritmens input. Antalet dataenheter i sekvensen representerar storlek på denna input. Det kan betecknas med n.

Vid sökningen jämförs den sökta dataenheten i tur och ordning (sekventiellt) med de dataenheter som finns i sekvensen. Så snart en likadan dataenhet påträffas, blir dess position lösningen till sökningsproblemet. Om ingen av dataenheter i sekvensen är likadant med den sökta dataenheten, så kan lösningen representeras med -1.

En möjlig implementering av algoritmen är given nedan. Algoritmen implementeras för heltal, men på ett liknande sätt kan den implementeras även för dataenheter av andra typer, så länge dessa dataenheter kan jämföras med varandra.

```
public static int search (int[] elements, int element)
{
    int    lastIndex = elements.length - 1;
    int    indexOfElement = -1;
    for (int index = 0; index <= lastIndex; index++)
    {
        if (element == elements[index])
        {
            indexOfElement = index;
            break;
        }
    }

    return indexOfElement;

}
```

Jämförelseoperationen upprepas ett antal gånger. Antalet dessa upprepningar bestämmer algoritmens tidskomplexitet för stora inputstorlek. Men antalet upprepningar beror inte bara på antalet dataenheter i sekvensen, utan även på det vilken dataenhet som söks och vilka dataenheter som finns i sekvensen. Hur många gånger jämförelseoperationen ska utföras vid en exekvering av algoritmen beror även på input. Därför kan inte komplexitetsfunktionen $T(n)$, som skulle gälla för alla möjliga input, bestämmas. Istället kan någon av komplexitetsfunktionerna $W(n)$, $A(n)$ eller $B(n)$ bestämmas.

Algoritmens värsta fall händer om den sökta dataenheten finns på sista positionen i sekvensen. För att hitta dess position i sekvensen, så måste den jämföras med alla dataenheter i sekvensen. Desamma händer även när den sökta dataenheten inte finns i sekvensen. Det betyder att algoritmens tidskomplexitet i värsta fall är:

$$W(n) = n$$

Bästa fall vid sekventiell sökning händer när den sökta dataenheten ligger på första positionen i sekvensen. I så fall utförs en enda jämförelse. Antalet exekveringar av den elementära operationen beror inte på inputs storlek, och därför

$$B(n) = 1$$

För att hitta algoritmens tidskomplexitet i ett genomsnittligt fall, kan flera antaganden göras. Anta att alla dataenheter är olika, att alla positioner vid sökningen är lika sannolika, och att sannolikheten för att dataenheten finns i sekvensen är p.

Under givna antaganden är sannolikheten att den sökta dataenheten finns på någon bestämd position k i sekvensen p/n. I så fall utförs jämförelseoperationen k gånger. Det betyder att jämförelseoperationen utförs 1 gång med sannolikheten p/n, 2

gånger med samma sannolikhet, 3 gånger med samma sannolikhet, o s v. Sannolikheten att den sökta dataenheten inte finns i sekvensen är 1 – p. I så fall utförs n jämförelser. Det totala antalet jämförelser blir i genomsnitt:

$$A(n) = 1p/n + 2p/n + 3p/n + \ldots + np/n + n(1 - p)$$

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

$$= (n(n + 1)/2)p/n + n(1 - p)$$

$$A(n) = n(1 - p/2) + p/2$$

Om sannolikheten att den sökta dataenheten finns i sekvensen är $p = 1/2$, så är komplexitetsfunktionen:

$$A(n) = 3n/4 + 1/4$$

Alltså, om $p = 1/2$ så genomsöks i genomsnitt omkring tre fjärdedelar av sekvensen.

En algoritms minneskomplexitet

För att uppskatta en algoritms minnesbehov (eng. memory complexity, extra space usage), kan motsvarande komplexitetsfunktion bestämmas. Denna komplexitetsfunktion ger antalet minnesceller som behövs vid exekveringen. Det minne som används för att lagra inputdata och outputdata räknas inte här. Det är bara de minnesceller som behövs för lagring av olika mellanresultat som räknas.

En komplexitetsfunktion som beskriver en algoritms tidsbehov har inputstorlek som sin variabel. Detsamma gäller de komplexitetsfunktioner som ger en algoritms minnesbehov. Om en sådan komplexitetsfunktion betecknas med M, och storlek på input med n, så blir $M(n)$ komplexitetsfunktionen. Värdet på M för en given n ger antalet nödvändiga minnesceller.

Vissa algoritmer kräver bara ett konstant antal minnesceller för sin exekvering. Behovet av andra algoritmer kan bero på storlek på input. T ex en sorteringsalgoritm kan behöva en vektor som är lika stor som den vektor som sorteras. Den här extravektorn används vid sorteringen. Det är uppenbart att en sådan algoritm har stora minnesbehov när ett stort antal dataenheter sorteras. Komplexitetsfunktionen är i så fall

$$M(n) = n$$

För en given algoritm kan olika komplexitetsfunktioner bestämmas för att uppskatta algoritmens tids- och minnesbehov. Men alla dessa komplexitetsfunktioner kan analyseras på ett gemensamt sätt. Genom denna analys vinner man olika insikter om den motsvarande algoritmens kvalitet. Det går att jämföra olika algoritmer för ett och samma problem, och välja dem som bäst passar.

Analys av komplexitetsfunktioner

Komplexitetsfunktioner

En algoritms tidskomplexitet kan yttras genom en matematisk funktion, som ger antalet exekveringar av en vald elementär operation för olika storlekar av input. Det kan vara en linjär funktion, en kvadratisk funktion, en logaritmisk funktion, o s v. Storleken på input kan ofta beskrivas med en enda parameter. Denna parameter representerar antalet dataenheter som hanteras av algoritmen, och därför är parametern ett positivt heltal. Antalet exekveringar av en elementär operation är också ett positivt heltal, men om ett genomsnitt beräknas kan decimaler erhållas också. Alltså avbildar komplexitetsfunktionen positiva heltal till reella positiva tal.

Värdet på komplexitetsfunktionen för en given storlek av input ger antalet exekveringar av en vald elementär operation. På så sätt får man ett mått för algoritmens tidseffektivitet, som är oberoende av värddatorn, implementeringsspråket, kompilatorn och implementeringsdetaljer. Då går det att jämföra olika algoritmer för ett och samma problem, och att välja den algoritm som bäst passar i en given situation. Komplexitetsfunktionen ger inte exekveringstiden i antalet sekunder, utan indirekt, genom antalet exekveringar av en elementär operation. För att uppskatta exekveringstiden kan en viss tid tilldelas till den elementära operationen. Man kan säga: låt exekveringen av den elementära operationen ta en nanosekund, eller en mikrosekund, o s v. Man bortser olika detaljer i algoritmen, och ger en uppskattning.

Antalet dataenheter som hanteras av algoritmen kan betecknas med n. Komplexitetsfunktionen kan betecknas med T, W, A eller B, beroende på det vad de representerar (eng. every-case time complexity, worst-case time complexity, average-case time complexity, best-case time complexity). På så sätt har man komplexitetsfunktionen $T(n)$, $W(n)$, $A(n)$ eller $B(n)$. För att kunna analysera dessa funktioner på ett enhetligt sätt, kan mer allmänna funktionsbeteckningar användas. Komplexitetsfunktionerna kan betecknas med f, g, h, o s v, oavsett vilken typ av komplexiteten de representerar. Det kan gälla även minneskomplexiteten. På så sätt kan en allmän teori av komplexitetsfunktioner byggas.

En mängd av liknande komplexitetsfunktioner

Det kan hända att tidskomplexiteten av en algoritm yttras genom följande komplexitetsfunktion:

$$f(n) = 2n^2 + 20n + 100$$

För olika värden på variabeln n bidrar de olika termerna olika till funktionens värde. För små värden på n dominerar termen 100, för något större värden på n dominerar termen $20n$, och för stora n dominerar termen $2n^2$. Den medföljande tabellen visar värden på de olika termerna och funktionens värde för olika n.

n	2n2	20n	100	f
1	2	20	100	122
2	8	40	100	148
5	50	100	100	250
7	98	140	100	338
10	200	200	100	500
20	800	400	100	1300
50	5000	1000	100	6100
100	20000	2000	100	22100
1000	2000000	20000	100	2020100
10000	200000000	200000	100	200200100

Den kvadratiska termen dominerar för stora n

Termerna 100 och $20n$ bestämmer algoritmens komplexitet för små storlek på input. När algoritmen hanterar ett stort antal dataenheter blir dessa termer försumbara, och termen $2n^2$ dominerar. I en polynomisk komplexitetsfunktion dominerar

termen med högsta exponent helt för stora värden på inputstorlek, och de andra termerna kan försummas. Det underlättar komplexitetsanalys för en algoritm betydligt. Man behåller helt enkelt bara termen med högsta exponent, och tar bort alla andra termer. Komplexiteten undersöks normalt för stora värden på inputstorlek, då en betydlig tid kan krävas för exekveringen. I så fall bestämmer den dominerande termen värdet på komplexitetsfunktionen. I fall att exekveringstiden undersöks för en mindre storlek av input, ska även de andra termerna betraktas.

Det finns en oändlig mängd kvardatiska komplexitetsfunktioner. Förutom funktionen f, kan många andra kvadratiska komplexitetsfunktioner anges. Exempel på sådana komplexitetsfunktioner är:

$$g(n) = 0.1\ n^2 + 100n$$

$$h(n) = 10\ n^2 + 20$$

$$f_1(n) = n^2 + 5n + 10$$

$$f_2(n) = 10\ n^2 + 2n + 5$$

Alla dessa komplexitetsfunktioner har en gemensam egenskap: termen som innehåller n^2 dominerar för stora n, och de andra termerna kan försummas. Det gäller oavsett storlek på koefficient som står framför n^2. Vid en asymptotisk analys av motsvarande algoritmer (analys för tillräckligt stora inputstorlekar), ser man n^2 som en bestämmande del i alla dessa funktioner.

De komplexitetsfunktioner där termen med n^2 dominerar för tillräckligt stora n, bildar en mängd som har ett speciellt namn. Denna mängd heter $\Theta(n^2)$ (Θ är en stor bokstav i grekiska - theta). Om en komplexitetsfunktion tillhör mängden $\Theta(n^2)$, så sägs det också att denna komplexitetsfunktion är av **storleksordning** n^2 (eng. order of n^2). Genom att veta en komplexitetsfunktions storleksordning, får man en uppfattning om komplexiteten för den motsvarande algoritmen för tillräckligt stora inputstorlekar. Om komplexitetsfunktionen för en algoritm är av storleksordning n^2, så sägs det att denna algoritm är en $\Theta(n^2)$ algoritm, eller att algoritmen är $\Theta(n^2)$. Det är en algoritm med kvadratisk tid (eng. quadratic-time algorithm).

För komplexitetsfunktionen f gäller det:

$$f(n) \in \Theta(n^2)$$

$f(n)$ är av storleksordning n^2

Detsamma gäller även för de andra angivna komplexitetsfunktionerna, och många andra komplexitetsfunktioner. Alla dessa komplexitetsfunktioner tillhör till en och samma mängd - mängden $\Theta(n^2)$, och alla dessa komplexitetsfunktioner är av storleksordnig n^2. Motsvarande algoritmer liknar varandra enligt deras komplexitet för tillräckligt stora n.

Mängder med polynomiska komplexitetsfunktioner

De komplexitetsfunktioner vars dominanta term för stora n är n^2 (multiplicerat med någon positiv konstant), tillhör till mängden $\Theta(n^2)$. Men det finns många komplexitetsfunktioner som inte tillhör till denna mängd. De tillhör till andra typiska mängder. Var och en av dessa mängder heter i grunden Θ, men den term som anges mellan parantes skiljer sig. Denna term representerar den asymptotiskt dominanta termen i alla de komplexitetsfunktioner som tillhör till den motsvarande mängden. På så sätt finns mängderna $\Theta(1)$, $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^3)$, $\Theta(n^4)$, o s v.

Det finns algoritmer vars komplexitet inte beror på storlek av input. Exempel av en sådan algoritm är den algoritm som bestämmer det minsta elementet i en redan sorterad sekvens. Det minsta elementet finns alltid på första platsen, oavsett hur många element som finns i sekvensen. Exekveringstiden av en sådan algoritm är konstant relativt inputstorlek, och motsvarande komplexitetsfunktion är också konstant. De komplexitetsfunktioner som är oberoende av storlek på input, tillhör till mängden $\Theta(1)$. Komplexitetsfunktionen $f_0(n) = 100$ tillhör till den här mängden.

$$f_0(n) = 100$$
$$f_0(n) \in \Theta(1)$$

$f_0(n)$ är av storleksordning 1

De komplexitetsfunktioner som asymptotisk har n (multiplicerat med någon positiv konstant) som dominerande term tillhör mängden $\Theta(n)$. De är av storleksordning n. Alla linjära komplexitetsfunktioner tillhör uppenbart till denna mängd (men inte bara dessa funktioner). Komplexitetsfunktionen $f_1(n) = 20n + 100$ tillhör till den här mängden. Det är termen $20n$ som dominerar för tillräckligt stora n, och termen 100 kan försummas i sådana fall.

$$f_1(n) = 20n + 100$$
$$f_1(n) \in \Theta\ (n)$$

$f_1(n)$ är av storleksordning n

De komplexitetsfunktioner som asymptotisk har n^2 (multiplicerat med någon positiv konstant) som dominerande term tillhör mängden $\Theta(n^2)$. De är av storleksordning n^2. Alla kvadratiska komplexitetsfunktioner tillhör uppenbart till denna mängd (men inte bara dessa funktioner). Komplexitetsfunktionen $f_2(n) = 2n2 + 20n + 100$ tillhör till den här mängden. Det är termen $2n^2$ som dominerar för stora n, och termerna $20n$ och 100 kan försummas i sådana fall.

$$f_2(n) = 2n^2 + 20n + 100$$
$$f_2(n) \in \Theta(n^2)$$
$f_2(n)$ är av storleksordning n^2

En polynomisk komplexitetsfunktion har följande form:

$$f_k(n) = a_0 n^k + a_1 n^{k-1} + a_2 n^{k-2} + \ldots + a_{k-1} n + a_k, \quad k \text{ ett heltal}, k \geq 0$$

För tillräcklig stora värden på n dominerar den första termen som har största exponent. Alla andra termer blir så småningom försumbara relativt den här termen. I så fall blir det n^k som bestämmer värde på komplexitetsfunktionen. Komplexitetsfunktionen är av storleksordning n^k, och den tillhör mängden $\Theta(n^k)$, för något icke-negativt heltal k.

$$f_k(n) \in \Theta(n^k)$$
$f_k(n)$ är av storleksordning n^k

Det är uppenbart att de algoritmer vars komplexitetsfunktioner tillhör mängden $\Theta(1)$ är snabbare än de algoritmer vars komplexitetsfunktioner tillhör mängden $\Theta(n)$ för tillräckligt stora n. Å andra sidan är de algoritmer vars komplexitetsfunktioner tillhör mängden $\Theta(n)$ asymptotiskt snabbare än de algoritmer vars komplexitetsfunktioner tillhör mängden $\Theta(n^2)$, de algoritmer vars komplexitetsfunktioner tillhör mängden $\Theta(n^2)$ är asymptotiskt snabbare än de algoritmer vars komplexitetsfunktioner tillhör mängden $\Theta(n^3)$, o s v. Dessa mängder gör det möjligt att de olika algoritmerna klassificeras när det gäller deras effektivitet för tillräckligt stora inputstorlekar. Tillhörighet till någon av dessa mängder sätter en effektivitetsstämpel på motsvarande algoritm.

Två komplexitetsfunktioner som tillhör till en och samma mängd, kan betydligt skilja sig när det gäller koefficienten framför den dominerande termen. Om dessa koefficienter är 0.1 och 10, så är den ena komplexitetsfunktionen ungefär 100 gånger mindre än den andra för tillräckligt stora n. Det betyder att motsvarande algoritm är 100 gånger snabbare. Men ändå tillhör komplexitetsfunktionerna till en och samma mängd. De liknar varandra när de jämförs med komplexitetsfunktioner från andra mängder. När en av flera algoritmer vars komplexitetsfunktioner är av samma storleksordning väljs, så beaktas även motsvarande koefficienter. När algoritmerna används i samband med mindre inputstorlekar, beaktas även de andra termerna i en komplexitetsfunktion, och inte bara den term som dominerar för stora inputstorlekar.

Typiska mängder med komplexitetsfunktioner

Komplexitetsfunktioner för vissa algoritmer är polynomiska, men det finns många algoritmer vars komplexitetsfunktioner inte är polynomiska. En komplexitetsfunktion kan även vara logaritmisk eller exponentiell, eller kan vara av någon annan typ. Den kan också vara en kombination av olika typer funktioner. Det följer flera exempel på komplexitetsfunktioner.

$$f_0(n) = 1$$
$$f_1(n) = 0.5n + 0.5$$
$$f_2(n) = 0.5n^2 - 0.5n$$
$$f_3(n) = n^3$$
$$f_4(n) = n^{2.81}$$
$$f_5(n) = 6n^{2.81} - 6n^2$$
$$f_6(n) = \log_2 n + 1$$
$$f_7(n) = n \log_2 n - n + 1$$
$$f_8(n) = 2^{n+1} - 1$$

Logaritmiska termer $\log_2 n$ och $n\log_2 n$ förekommer ofta när effektiviteten för olika algoritmer analyseras. Därför är det viktigt att veta hur snabbt dessa funktioner växer, och hur dessa funktioner relateras till polynomiska funktioner. Det visas i den medföljande tabellen.

n	$\log_2 n$	$n \log_2 n$	n^2
8	3	24	64
16	4	64	256

32	5	160	1024
64	6	384	4096
128	7	896	16384
256	8	2048	65536
512	9	4608	262144
1024	10	10240	1048576
2048	11	22528	4194304
4096	12	49152	16777216

Logaritmer och potenser

Det är uppenbart att den logaritmiska funktionen $\log_2 n$ växer mycket långsamt. För varje fördubbling av n ökas bara funktionen med 1. Den är även långsammare än funktionen $f(n) = n$. Därför är algoritmer med logaritmisk komplexitetsfunktion snabba även för stora inputstorlekar. Vissa sökningsalgoritmer är av den här komplexiteten.

De komplexitetsfunktioner där termen $\log_2 n$ (multiplicerad med någon positiv konstant) dominerar för stora n, bildar en särskild mängd. Den här mängden heter $\Theta(\log_2 n)$. Komplexitetsfunktionen $f(n) = 2\log_2 n + 10$ tillhör klart till denna mängd. Den är av storleksordning $\log_2 n$.

$$f(n) = 2\log_2 n + 10$$

$$f(n) \in \Theta(\log_2 n)$$

$f(n)$ är av storleksordning $\log_2 n$

Termen $n\log_2 n$ växer för stora n mycket snabbare än termen n, och mycket långsammare än termen n^2. De komplexitetsfunktioner där termen $n\log_2 n$ (multiplicerad med någon positiv konstant) dominerar för stora n, tillhör till mängden $\Theta(n\log_2 n)$. Komplexitetsfunktionen $g(n) = n\log_2 n - n + 1$ tillhör klart till denna mängd. Den är av storleksordning $n\log_2 n$.

$$g(n) = n\log_2 n - n + 1$$
$$g(n) \in \Theta(n\log_2 n)$$

$g(n)$ är av storleksordning $n\log_2 n$

Vissa sorteringsalgoritmer har komplexiteten $\Theta(n\log_2 n)$. Om både kvadratiska termer och termer av formen $n\log_2 n$ förekommer i någon komplexitetsfunktion, tillhör den här komplexitetsfunktionen till mängden $\Theta(n^2)$. Den kvadratiska termen dominerar i så fall för stora n.

De exponentiella komplexitetsfunktioner växer mycket snabbt när storlek på input ökas. Termen 2^n fördubblas när n ökas med 1. Exponentiella termer dominerar för tillräckligt stora n över termerna n, n^2, n^3, o s v. Dessutom växer inte två exponentiella funktioner med olika baser lika snabbt. Funktionen 4^n växer mycket snabbare än funktionen 3^n, som i sin tur växer mycket snabbare än funktionen 2^n, o s v. Mängder med de komplexitetsfunktioner där motsvarande exponentiella termer dominerar är $\Theta(2^n)$, $\Theta(3^n)$, $\Theta(4^n)$, o s v. Algoritmer som har exponentiell komplexitet kan vara mycket långsamma för stora n. Exekveringen kan ta även många år. Men dessa algoritmer kan ändå vara acceptabla för mindre storlekar på input.

Ännu långsammare är algoritmer vars komplexitet yttras via fakultetsfunktioner. Mängden $\Theta(n!)$ innehåller komplexitetsfunktioner som kan vara mycket långsamma även för mindre storlekar på input.

Följande sekvens anger de typiska mängderna, ordnade enligt storleksordning av motsvarande komplexitetsfunktioner.

$$\Theta(1),\ \Theta(\log_2 n),\ \Theta(n),\ \Theta(n\log_2 n),\ \Theta(n^2),\ \Theta(n^3),\ \Theta(2^n),\ \Theta(3^n),\ \Theta(n!)$$

Alla dessa mängder är disjunkta. Om en komplexitetsfunktion tillhör till en av dessa mängder, kan den inte tillhöra även till någon annan av dessa mängder. Indelningen av komplexitetsfunktioner i olika mängder gör möjligt att man får en bra uppfattning om komplexiteten för en viss algoritm. Man kan jämföra olika algoritmer för ett och samma problem, liksom algoritmer för olika problem. Tillhörighet till någon av dessa mängder sätter en effektivitetsstämpel på motsvarande algoritm. I vissa situationer är det svårt att exakt bestämma en algoritms komplexitetsfunktion. I sådana fall kan det räcka med att bestämma komplexitetsfunktionens storleksordning. Komplexitetsteorin gäller i mångt och mycket uppskattningar.

Definition av en komplexitetsfunktions storleksordning

Mängderna $\Theta(\log_2 n)$, $\Theta(n)$, $\Theta(n\log_2 n)$, $\Theta(n^2)$, och många andra mängder med komplexitetsfunktioner, har formen $\Theta(f(n))$. Komplexitetsfunktionen $f(n)$ är i så fall en typisk representant av alla de funktioner som finns i den motsvarande mängden. Hela mängden är igenkänd tack vare denna funktion, och därför väljs enkla funktioner som representanter för mängderna. T ex istället för funktionen n^2 kunde någon annan komplexitetsfunktion från mängden $\Theta(n^2)$ väljas som dess representant. Om funktionen $2n^2 + 10n + 20$ väljs, så anges den motsvarande mängden som $\Theta(2n^2 + 10n + 20)$ istället för bara $\Theta(n^2)$.

Komplexitetsfunktionerna 2^n och 3^n tillhör till mängderna $\Theta(2^n)$ och $\Theta(3^n)$. Man undrar varför dessa komplexitetsfunktioner tillhör till olika mängder. Till vilken mängd tillhör komplexitetsfunktionen $f(n) = 2.1^n$? Tillhör komplexitetsfunktionerna $\log_2 n$ och $\log_{10} n$ till samma mängd?

För att kunna svara på dessa frågor behöver man exakt veta vad mängden $\Theta(f(n))$ är. Följande definition preciserar vilka komplexitetsfunktioner ingår i denna mängd.

Definition

För en given komplexitetsfunktion $f(n)$, så är $\Theta(f(n))$ mängden av de komplexitetsfunktioner $g(n)$ som uppfyller följande villkor.

Det finns två positiva reella konstanter c och d och ett icke negativt heltal N, så att för alla $n \geq N$ gäller:

$$cf(n) \leq g(n) \leq df(n)$$

Alltså tillhör komplexitetsfunktionen $g(n)$ till mängden $\Theta(f(n))$, om och bara om den kan begränsas med komplexitetsfunktionen $cf(n)$ nerifrån (eng. asymptotic lower bound), och med komplexitetsfunktionen $df(n)$ uppifrån (eng. asymptotic upper bound) för tillräckligt stora n. Komplexitetsfunktionerna $g(n)$ och $f(n)$ är av samma storleksordning, och de motsvarande algoritmerna är ungefär lika bra för tillräckligt stora n.

För att bevisa att komplexitetsfunktionen $g(n)$ tillhör mängden $\Theta(f(n))$, räcker det att hitta en uppsättning konstanter c och d, och motsvarande N, så att de givna villkoren gäller. Men samma villkor kan gälla även för många andra uppsättningar av c, d och N.

Exempel

Bevisa att komplexitetsfunktionen $g(n) = 0.5n^2 - 0.5n$ tillhör till mängden $\Theta(n^2)$.

Enligt definitionen behöver man bevisa att det finns positiva reella konstanter c och d och ett icke negativt heltal N, så att för alla $n \geq N$ gäller $cn^2 \leq 0.5n^2 - 0.5n \leq dn^2$.

Det är lättare att bestämma konstanten d. Om man väljer $d = 1$, så gäller det $0.5n^2 - 0.5n \leq 1n^2$ för $n \geq 0$. Konstanten c kan bestämmas ifrån olikheten $cn^2 \leq 0.5n^2 - 0.5n$. Man gissar c så att olikheten ska gälla för tillräckligt stora n. Om 0.25 väljs för c, så blir olikheten $0.25n^2 \leq 0.5n^2 - 0.5n$. Efter bearbetning erhålls olikheten $0.5n \leq 0.25n^2$, som efter förkortning med n och 0.25 ger $2 \leq n$.

Alltså gäller den första olikheten i definitionen för $c = 0.25$ för alla $n \geq 2$. Den andra olikheten gäller för $d = 1$ för alla $n \geq 0$. Det betyder att båda olikheterna gäller för $c = 0.25$ och $d = 1$ för alla $n \geq 2$ ($N = 2$). Enligt definitionen så tillhör komplexitetsfunktionen $g(n) = 0.5n^2 - 0.5n$ till mängden $\Theta(n^2)$.

Komplexitetsfunktionen $g(n) = 0.5n^2 - 0.5n$ kan för $n \geq 2$ begränsas nerifrån med komplexitetsfunktionen $0.25n^2$, och uppifrån med komplexitetsfunktionen $1n^2$ (alltså n^2). Det betyder att komplexitetsfunktionen $g(n)$ har samma storleksordning som komplexitetsfunktionen n^2 för $n \geq 2$.

En annan möjlig uppsättning värden på konstanterna är $c = 0.1$, $d = 10$ och $N = 100$.

Exempel

Bevisa att komplexitetsfunktionen $g(n) = n^2$ tillhör till mängden $\Theta(0.5n^2 - 0.5n)$.

Enligt definitionen behöver man bevisa att det finns positiva reella konstanter c och d och ett icke negativt heltal N, så att för alla $n \geq N$ gäller $c(0.5n^2 - 0.5n) \leq n^2 \leq d(0.5n^2 - 0.5n)$.

Om man väljer $c = 1$ och $d = 4$, så gäller då båda två olikheterna för $n \geq 2$ ($N = 2$). Alltså tillhör komplexitetsfunktionen $g(n) = n^2$ till mängden $\Theta(0.5n^2 - 0.5n)$.

Komplexitetsfunktionen n^2 tillhör till mängden $\Theta(0.5n^2 - 0.5n)$, och komplexitetsfunktionen $0.5n^2 - 0.5n$ tillhör till mängden $\Theta(n^2)$. Komplexitetsfunktionerna n^2 och $0.5n^2 - 0.5n$ är av samma storleksordning för tillräckligt stora n. De båda komplexitetsfunktionerna tillhör till en och samma mängd. Denna mängd kan anges både som $\Theta(n^2)$ och som $\Theta(0.5n^2 - 0.5n)$, och på annat sätt.

Exempel

Bevisa att komplexitetsfunktionen $g(n) = n^3$ inte tillhör till mängden $\Theta(n^2)$.

Om komplexitetsfunktionen g(n) tillhör till mängden $\Theta(n^2)$, så finns två positiva reella konstanter c och d och ett icke negativt heltal N, så att för alla $n \geq N$ gäller $cn^2 \leq n^3 \leq dn^2$. Efter förkortning med n^2 omvandlas den andra olikheten till $n \leq d$. Så olikheten kan bara gälla för de värden på n som är mindre än d. Oavsett vilket (fast, oberoende av n) värde på d väljs, så kan olikheten inte gälla för $n > d$. Eftersom olikheten inte kan gälla för tillräckligt stora n, är villkoret i definitionen inte uppfyllt. Komplexitetsfunktionen $g(n) = n^3$ tillhör inte till mängden $\Theta(n^2)$, eftersom det inte går att begränsa den för tillräckligt stora n med komplexitetsfunktionen dn^2, oavsett hur konstanten d väljs. Komplexitetsfunktionerna n^3 och n^2 är inte av samma storleksordning. Motsvarande algoritmer är inte ungefär lika bra.

Exempel

Bevisa att komplexitetsfunktionen $g(n) = n$ inte tillhör till mängden $\Theta(n^2)$.

Komplexitetsfunktionen $g(n) = n$ kan inte begränsas nerifrån med komplexitetsfunktionen cn^2 för tillräckligt stora n, oavsett hur konstanten c väljs. Det går inte att hitta konstanten c och ett gränsvärde N, så att olikheten $cn^2 \leq n$ ska gälla för alla $n \geq N$. Denna olikhet kan gälla bara för $n \leq 1/c.$, när c väl är vald.

Komplexitetsfunktionen $g(n) = n$ tillhör inte till mängden $\Theta(n^2)$. Komplexitetsfunktionerna n och n^2 är inte av samma storleksordning.

Det går att bevisa att två komplexitetsfunktioner är av samma storleksordning genom att beräkna gränsvärdet på deras kvot när n går mot oändlighet. Om gränsvärdet finns och är ett positivt tal, så är komplexitetsfunktionerna av samma storleksordning.

Sats

Om $g(n) / f(n) \rightarrow c$, $c > 0$ när $n \rightarrow \infty$, så $g(n) \in \Theta(f(n))$.

Om gränsvärdet $g(n) / f(n)$ finns och är ett positivt tal, så finns också gränsvärdet $f(n) / g(n)$ och är ett positivt heltal. Det betyder att om $g(n) \in \Theta(f(n))$, så $f(n) \in \Theta(g(n))$. Komplexitetsfunktionerna är av samma storleksordning, och de tillhör till en och samma mängd. De kan begränsa varandra både uppifrån och nerifrån för tillräckligt stora n, om de multipliceras med rätt valda konstanter.

Exempel

Bevisa att komplexitetsfunktionerna $\log_2 n$ och $\log_{10} n$ är av samma storleksordning.

Man kan beräkna gränsvärdet för motsvarande funktioner med reell variabel x, istället för heltalsvariabeln n, alltså för $\log_2 x$ och $\log_{10} x$.

Enligt L'Hospital's regel är gränsvärdet för kvot av två funktioner som går mot oändlighet, lika med gränsvärdet av kvoten av funktionernas derivator (om dessa derivator finns). Derivatorna för funktionerna $\log_2 x$ och $\log_{10} x$ är $1/(x \ln 2)$ och $1/(x \ln 10)$, och deras kvot är $\ln 10 / \ln 2$, alltså oberoende av x. Så $\ln 10 / \ln 2$ blir kvotens gränsvärde, både för funktioner med reell variabel och för komplexitetsfunktioner med heltalsvariabel. Eftersom gränsvärdet är positivt, så är komplexitetsfunktionerna $\log_2 n$ och $\log_{10} n$ av samma storleksordning. Det gäller allmänt alla logaritmiska komplexitetsfunktioner vars bas är större än 1.

Så $\log_{10} n \in \Theta(\log_2 n)$ och $\log_2 n \in \Theta(\log_{10} n)$. Komplexitetsfunktionerna $\log_2 n$ och $\log_{10} n$ tillhör till en och samma mängd.

Övre och nedre asymptotiska gränser

För att uppskatta en algoritms effektivitet bestämmer man dess komplexitetsfunktion. Den här komplexitetsfunktionen ger antalet exekveringar av en vald elementär operation för olika inputstorlekar. Värden på komplexitetsfunktionen kan vara speciellt intressanta för stora inputstorlekar, eftersom då kan algoritmen kräva mycket tid för exekveringen. I så fall analyserar man komplexitetsfunktionen, och bestämmer dess storleksordning för stora värden på dess variabel. Komplexitetsfunktionen läggs i någon av typiska mängder med komplexitetsfunktioner, och på så sätt sätts en effektivitetsstämpel på algoritmen. Det går att klassificera och jämföra olika algoritmer när det gäller deras effektivitet.

I vissa situationer kan det vara svårt att bestämma en komplexitetsfunktions storleksordning. Motsvarande uttryck kan vara alltför komplicerat, eller till och med kan det vara svårt att erhålla ett exakt uttryck för komplexitetsfunktionen. Men även i sådana fall behöver man någon uppskattning av algoritmens effektivitet. Det kan t ex vara av intresse att hitta någon, så bra som möjligt, övre gräns för antalet exekveringar av en elementär operation. På så sätt sätter man en övre gräns på komplexitetsfunktionens storleksordning. Ibland kan det vara upplysande att bestämma en nedre gräns på komplexitetsfunktionen. Speciellt kan det vara viktigt att hitta dessa gränser för stora inputstorlekar. Istället för att hitta exakt storleksordning på komplexitetsfunktionen, kan man nöja sig med att erhålla lämpliga asymptotiska gränser (gränser för stora inputstorlekar).

Storleksordning av en komplexitetsfunktion yttras genom dess tillhörighet till någon av de typiska mängderna med komplexitetsfunktioner. Komplexitetsfunktionen kan t ex tillhöra mängden $\Theta(n^2)$, och på så sätt fastställs att dess storleksordning är n^2. Det betyder att komplexitetsfunktionen har cn^2 som sin övre asymptotisk gräns, och dn^2 som sin nedre asymptotisk gräns, där c och d är två positiva reella konstanter. Komplexitetsfunktionen är asymptotisk lika bra som komplexitetsfunktionen n^2.

I fall att bara en övre asymptotisk gräns för en komplexitetsfunktion ska anges, används tillhörighet till någon av de typiska mängderna $O(1)$, $O(\log_2 n)$, $O(n)$, $O(n\log_2 n)$, $O(n^2)$, o s v. Man använder stora O notationen för att yttra en övre asymptotisk gräns för komplexitetsfunktionen. På så sätt tillhör t ex komplexitetsfunktionen $g(n) = 0.5n2 - 0.5n$ till mängden $O(n^2)$. Man säger att $f(n)$ är stora O av n^2.

$$g(n) = 0.5n^2 - 0.5n$$

$$g(n) \in O(n^2)$$

$g(n)$ är stora O av n^2

Mängden $O(f(n))$ innehåller alla de komplexitetsfunktioner som för tillräckligt stora n kan begränsas uppifrån med funktionen $cf(n)$, där c är en positiv konstant. Det preciseras i följande definition.

> **Definition**
>
> För en given komplexitetsfunktion $f(n)$, så är $O(f(n))$ mängden av de komplexitetsfunktioner $g(n)$ som uppfyller följande villkor.
>
> Det finns en positiv reell konstant c och ett icke negativt heltal N, så att för alla $n \geq N$ gäller:
>
> $$g(n) \leq cf(n)$$

Exempel

Bevisa att komplexitetsfunktionen $g(n) = 0.5n^2 - 0.5n$ tillhör till mängden $O(n^2)$.

Enligt definitionen behöver man bevisa att det finns en positiv reell konstant c och ett icke negativt heltal N, så att för alla $n \geq N$ gäller $0.5n^2 - 0.5n \leq cn^2$.

Om man väljer $c = 1$, så gäller det $0.5n^2 - 0.5n \leq 1n^2$ för alla $n \geq 0$. Villkoret i definitionen är uppfyllt med konstanten $c = 1$ och $N = 0$. Så tillhör komplexitetsfunktionen $g(n)$ till mängden $O(n^2)$.

Komplexitetsfunktionen $g(n) = 0.5n^2 - 0.5n$ kan för $n \geq 0$ begränsas uppifrån med komplexitetsfunktionen $1n^2$ (alltså n^2). Det betyder att komplexitetsfunktionen $g(n)$ är inte sämre (när det gäller dess storleksordning) än komplexitetsfunktionen n^2 för $n \geq 0$.

En komplexitetsfunktion kan på olika sätt begränsas uppifrån. Det finns många möjliga uppskattningar för en komplexitetsfunktion, men man är intresserad för en tillräckligt bra uppskattning. T ex komplexitetsfunktionen $g(n) = 0.5n^2 - 0.5n$ tillhör mängden $O(n^2)$, men den tillhör också till mängden $O(n^3)$. Om det går att asymptotisk begränsa $g(n)$ med n^2, så går det desto lättare att göra det med n^3, som är större än n^2. Men n^2 är en bättre uppskattning, en mindre övre gräns.

En nedre asymptotisk gräns för en komplexitetsfunktion yttras genom Ω (grekisk bokstav omega) notationen. Det finns mängderna $\Omega(1)$, $\Omega(\log_2 n)$, $\Omega(n)$, $\Omega(n\log_2 n)$, $\Omega(n^2)$, o s v. Tillhörighet till någon av dessa mängder definieras på följande sätt.

Definition

För en given komplexitetsfunktion $f(n)$, så är $\Omega(f(n))$ mängden av de komplexitetsfunktioner $g(n)$ som uppfyller följande villkor.

Det finns en positiv reell konstant c och ett icke negativt heltal N, så att för alla $n \geq N$ gäller:

$$g(n) \geq cf(n)$$

Exempel

Bevisa att komplexitetsfunktionen $g(n) = 0.5n^2 - 0.5n$ tillhör till mängden $\Omega(n^2)$.

Enligt definitionen behöver man bevisa att det finns en positiv reell konstant c och ett icke negativt heltal N, så att för alla $n \geq N$ gäller $0.5n^2 - 0.5n \geq cn^2$. Om man väljer $c = 0.25$ och $N = 2$, så gäller detta villkor. Så tillhör komplexitetsfunktionen $g(n)$ till mängden $\Omega(n^2)$.

Komplexitetsfunktionen $g(n) = 0.5n^2 - 0.5n$ kan för $n \geq 2$ begränsas nerifrån med komplexitetsfunktionen $0.25n^2$. Det betyder att komplexitetsfunktionen $g(n)$ är inte bättre än komplexitetsfunktionen $0.25n^2$ för $n \geq 2$.

En komplexitetsfunktion kan på olika sätt begränsas nerifrån. Komplexitetsfunktionen $g(n) = 0.5n^2 - 0.5n$ tillhör mängden $\Omega(n^2)$, men den tillhör också till mängden $\Omega(n)$. Om det går att asymptotisk begränsa $g(n)$ nerifrån med n^2, så går det desto lättare att göra det med n, som är ännu mindre än n^2. Men n^2 är en bättre uppskattning, en större nedre gräns.

Mängden $O(f(n))$ innehåller alla de komplexitetsfunktioner som för tillräckligt stora n kan begränsas uppifrån med komplexitetsfunktionen $df(n)$ (d positiv konstant). Mängden $\Omega(f(n))$ innehåller alla de komplexitetsfunktioner som för tillräckligt stora n kan begränsas nerifrån med komplexitetsfunktionen $cf(n)$ (c positiv konstant). Snittet av dessa två mängder innehåller de komplexitetsfunktioner som för tillräckligt stora n kan begränsas uppifrån med komplexitetsfunktionen $df(n)$ och nerifrån med komplexitetsfunktionen $cf(n)$. Detta snitt är mängden $\Theta(f(n)$. Den komplexitetsfunktion som tillhör till både $O(f(n))$ och $\Omega(f(n))$, är av storleksordning $f(n)$. Alltså

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

Så ett sätt att bevisa att någon komplexitetsfunktion är av storleksordning $f(n)$, är att bevisa att denna komplexitetsfunktion tillhör till både $O(f(n))$ och $\Omega(f(n))$.

Om en funktion $cf(n)$ begränsar funktionen $g(n)$ uppifrån, så begränsar funktionen $(1/c)g(n)$ funktionen $f(n)$ nerifrån. Det betyder att om en komplexitetsfunktion $g(n)$ tillhör mängden $O(f(n))$, så tillhör komplexitetsfunktionen $f(n)$ mängden $\Omega(g(n))$.

En oändlig familj av övre asymptotiska gränser

Förutom mängderna $\Theta(f(n))$, $O(f(n))$ och $\Omega(f(n))$, används i komplexitetsteorin även mängden $o(f(n))$. Om någon komplexitetsfunktion tillhör till denna mängd, så sägs att denna komplexitetsfunktion är lila o av $f(n)$. Mängden $o(f(n))$ definieras på följande sätt.

Definition

För en given komplexitetsfunktion $f(n)$, så är $o(f(n))$ mängden av de komplexitetsfunktioner $g(n)$ som uppfyller följande villkor.

För varje positiv reell konstant c så finns ett icke negativt heltal N, så att för alla $n \geq N$ gäller:

$$g(n) \leq cf(n)$$

Vad är skillnaden mellan mängderna $O(f(n))$ och $o(f(n))$? För tillhörighet till mängden $O(f(n))$ krävs det att det finns någon positiv konstant c, så att $cf(n)$ blir asymptotisk övre gräns. Men för tillhörighet till mängden $o(f(n))$ krävs det att för vilken som helst positiv konstant c blir $cf(n)$ en övre asymptotisk gräns. Det är uppenbart att det andra är ett hårdare krav. Man kan välja konstanten c mycket liten, men ändå ska $cf(n)$ vara en övre asymptotisk gräns. Begränsningen ska gälla för alla $n \geq N$, där N kan bero på c, och kan vara mycket stor.

Genom att variera värdet på konstanten c så får man en oändlig familj av övre asymptotiska gränser. Ju mindre värde på konstanten c väljs, desto bättre begränsning får man. Men samtidigt kan det hända att denna begränsning bara gäller för mycket stora värden på n, möjligen orealistiska när det gäller motsvarande algoritm.

Exempel

Bevisa att komplexitetsfunktionen $g(n) = 0.5n^2 - 0.5n$ inte tillhör till mängden $o(n^2)$.

Enligt definitionen behöver man bevisa att för varje positiv reell konstant c så finns ett icke negativt heltal N, så att för alla $n \geq N$ gäller $0.5n^2 - 0.5n \leq cn^2$.

Om man väljer $c = 0.25$, så ska det gälla att $0.5n^2 - 0.5n \leq 0.25n^2$ för alla $n \geq N$, för något icke negativt heltal N. Men för detta val av konstanten c omvandlas olikheten till $n \leq 2$. Oavsett hur heltalet N väljs, så kan den olikheten inte gälla för alla $n \geq N$. Så tillhör komplexitetsfunktionen $g(n)$ inte till mängden $o(n^2)$.

Exempel

Bevisa att komplexitetsfunktionen $g(n) = n$ tillhör till mängden $o(n^2)$.

Enligt definitionen behöver man bevisa att för varje positiv reell konstant c så finns ett icke negativt heltal N, så att för alla $n \geq N$ gäller $n \leq cn^2$. Efter förkortning med n omvandlas olikheten till $1 \leq cn$. Denna olikhet gäller för alla $n \geq 1/c$. Så låt N bli $1/c$, och då gäller olikheten för alla $n \geq N$.

Så oavsett hur c väljs så finns det motsvarande N, så att olikheten gäller för alla $n \geq N$. Komplexitetsfunktionen $g(n) = n$ tillhör till mängden $o(n^2)$. Denna komplexitetsfunktion är asymptotiskt mycket bättre än komplexitetsfunktionen n^2. Den kan göras hur många gånger som helst mindre än komplexitetsfunktionen n^2 för tillräckligt stora n.

Om en komplexitetsfunktion $g(n)$ tillhör till mängden $o(f(n))$, så tillhör den också till mängden $O(f(n))$. Tillhörighet till mängden $o(f(n))$ medför att $g(n)$ asymptotiskt kan begränsas uppifrån med komplexitetsfunktionen $cf(n)$ för vilken som helst värde på den positiva konstanten c. Det betyder att det finns något värde på denna konstant så att $g(n)$ asymptotiskt kan begränsas uppifrån med komplexitetsfunktionen $cf(n)$. Enligt definitionen så tillhör då komplexitetsfunktionen $g(n)$ även till mängden $O(f(n))$.

Om en komplexitetsfunktion $g(n)$ tillhör till mängden $o(f(n))$, så tillhör den inte till mängden $\Omega(f(n))$. Om komplexitetsfunktionen tillhör mängden $\Omega(f(n))$, så kan den asymptotiskt begränsas nerifrån med komplexitetsfunktionen $cf(n)$ för något värde på den positiva konstanten c. Men tillhörigheten till $o(f(n))$ medför att även för detta värde på c komplexitetsfunktionen $g(n)$ asymptotiskt kan begränsas uppifrån med komplexitetsfunktionen $0.5cf(n)$, som är mindre än $cf(n)$. Det är omöjligt att komplexitetsfunktionen $g(n)$ samtidigt begränsas nerifrån med $cf(n)$, och uppifrån med $0.5cf(n)$.

Alltså om en komplexitetsfunktion tillhör till mängden $o(f(n))$, tillhör den även till mängden $O(f(n))$, men tillhör inte till mängden $\Omega(f(n))$. Den tillhör till skillnaden mellan dessa två mängder.

$$\text{om } g(n) \in o(f(n)) \text{ så } \quad g(n) \in O(f(n)) \setminus \Omega(f(n))$$

En komplexitetsfunktion $g(n)$ tillhör till mängden $o(f(n))$ om och bara om för varje positiv reell konstant c så finns ett icke negativt heltal N, så att för alla $n \geq N$ gäller $g(n) \leq cf(n)$. Villkoret $g(n) \leq cf(n)$ betyder att kvoten $g(n)/f(n)$ kan göras mindre än vilket som helst positivt värde för tillräckligt stora n. Alltså kvoten går mot 0 när n går mot oändlighet.

$$g(n) \in o(f(n)) \text{ om och bara om } g(n) \,/\, f(n) \to 0 \text{ när } n \to \infty$$

Exempel

Bevisa att komplexitetsfunktionen $g(n) = n^{0.5}$ tillhör till mängden $o(n)$.

Det räcker att bevisa att $g(n)/n$ går mot 0 när n går mot oändlighet. Men det är sant, eftersom

$$g(n)/n = n^{0.5}/n = 1/\ n^{0.5} \to 0 \text{ när } n \to \infty$$

Så komplexitetsfunktionen $g(n) = n^{0.5}$ tillhör till mängden $o(n)$.

Sammanfattning

En algoritm hanterar på något sätt ett antal dataenheter, och på så sätt löser ett problem. Dessa dataenheter utgör algoritmens input, och antalet dessa dataenheter representerar storlek på denna input.

Som en del i analys av en algoritm, bestämmer man dess tidskomplexitet. Man särskiljer en tidsmässigt dominant operation som upprepas, och bestämmer hur många gånger denna operation utförs för olika storlekar på input. Den funktion som erhålls på så sätt kallas för komplexitetsfunktion. Via denna komplexitetsfunktion yttras algoritmens tidskomplexitet.

I vissa algoritmer beror antalet exekveringar av en elementär operation inte bara på storlek på input, utan även på själva input. För att uppskatta tidskomplexiteten i dessa fall, bestämmer man största möjliga antal exekveringar, eller genomsnittliga antal exekveringar eller minsta möjliga antal exekveringar. Man identifierar värsta fall, genomsnittliga fall och bästa fall, och bestämmer motsvarande komplexitetsfunktioner.

Förutom tidskomplexiteten, kan även minneskomplexiteten bestämmas för en algoritm. Den specificerar nödvändiga minnesresurser för att problemet ska kunna lösas. Motsvarande komplexitetsfunktion ger antalet nödvändiga minnesceller för olika inputstorlekar. De minnesresurser som uppskattas på det här sättet omfattar inte det minne som tas upp av inputdata och outputdata.

För att kunna tolka olika komplexitetsfunktioner, delas de i olika disjunkta mängder. Mängden $\Theta(f(n))$ omfattar alla de komplexitetsfunktioner som asymptotiskt (för tillräckligt stora inputstorlekar) kan begränsas uppifrån med funktionen $cf(n)$, och nerifrån med funktionen $df(n)$, där c och d är positiva konstanter och n storlek på input. Typiska sådana mängder är $\Theta(\log_2 n)$, $\Theta(n)$, $\Theta(n\log_2 n)$, $\Theta(n^2)$, o s v. Om en komplexitetsfunktion tillhör mängden $\Theta(f(n))$, så sägs det att komplexitetsfunktionen är av storleksordning $f(n)$.

Det finns olika metoder för bestämmandet av en komplexitetsfunktions storleksordning. Man kan t ex behålla bara den term i komplexitetsfunktionen som dominerar för stora inputstorlekar och försumma de andra termerna. För att bevisa att en komplexitetsfunktion $g(n)$ tillhör mängden $\Theta(f(n))$, kan man utgå ifrån definitionen för mängden $\Theta(f(n))$, eller beräkna gränsvärdet för kvoten $g(n)/f(n)$ när n går mot oändligheten. Om detta gränsvärde är ett positivt tal, så är det $g(n)$ av storleksordning $f(n)$.

I vissa situationer är det svårt att bestämma en komplexitetsfunktions storleksordning, eller t o m kan det hända att det är svårt att bestämma själva komplexitetsfunktionen för en viss algoritm. I så fall kan det vara nyttigt att bestämma en asymptotisk övre gräns eller en asymptotisk nedre gräns på komplexitetsfunktionen. På så sätt kan man säga att en komplexitetsfunktion asymptotisk kan begränsas uppifrån med funktionen $cf(n)$, för en positiv konstant c. Mängden av alla sådana komplexitetsfunktioner kallas för $O(f(n))$. Mängden av de komplexitetsfunktioner som asymptotiskt kan begränsas nerifrån med $cf(n)$, för en positiv konstant c, kallas för $\Omega(f(n))$. Genom att säga att en komplexitetsfunktion är av $O(f(n))$, eller $\Omega(f(n))$, för någon funktion $f(n)$, gör man en uppskattning av den motsvarande algoritmens tidskomplexitet. Man vet att tidskomplexiteten inte är sämre, eller bättre, än någon standardkomplexitet.

Ibland kan en oändlig familj av asymptotiska övre gränser för en komplexitetsfunktion bestämmas. I så fall kan komplexitetsfunktionen asymptotiskt begränsas uppifrån med $cf(n)$, för vilket som helst (även mycket litet) värde på c. Mängden av alla de komplexitetsfunktioner som på så sätt kan begränsas uppifrån kallas för $o(f(n))$. Tidskomplexiteten för motsvarande algoritmer är i så fall mycket bättre än tidskomplexiteten för de algoritmer vars komplexitetsfunktion är $f(n)$.