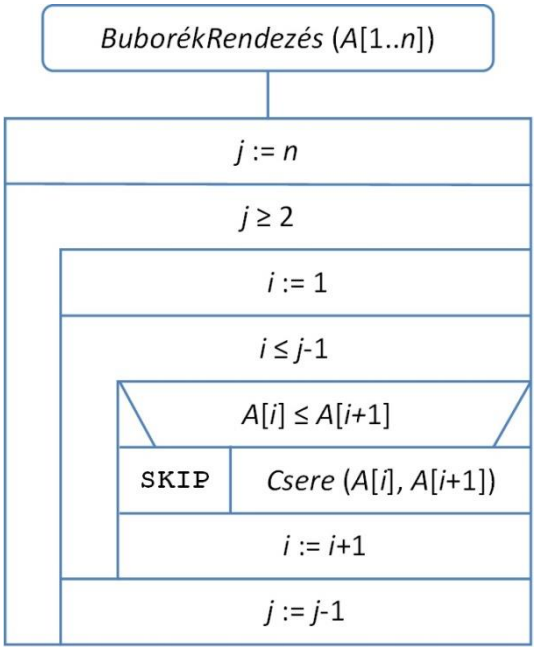
1. ALGORITMUSOK MŰVELETIGÉNYE

Az ismertetésre kerülő adatszerkezeteket és algoritmusokat mindig jellemezzük majd a *hatékonyság* szempontjából. Az adatszerkezetek egyes ábrázolásairól megállapítjuk a *helyfoglalásukat*, az algoritmusoknál pedig a *műveletigényt* becsüljük, mindkettőt az input adatok méretének függvényében. Általában megelégszünk mindkét adat nagyságrendben közelítő értékével. A nagyságrendi értékek közelítő ereje annál nagyobb, minél nagyobb méretű adatokra értelmezzük azokat. Amint látjuk majd, egy sajátos matematikai határérték-fogalmat vezetünk be és alkalmazunk a hatékonyságra irányuló számításainkban.

A műveletigény számításakor eleve azzal a közelítéssel élünk, hogy csak az algoritmus meghatározó műveleteit vesszük számításba. Egy műveletet *meghatározónak* (dominánsnak) mondunk, ha a többihez képest jelentős a végrehajtási ideje, valamint a végrehajtásainak száma. Általában kijelölhető egyetlen meghatározó művelet, amelyre a számítást elvégezzük. A műveletigényt a kiszemelt művelet végrehajtásainak számával adjuk meg, mivel az egyes műveletek végrehajtási ideje gépről-gépre változhat. A *lépésszám* közelítéssel kiszámolt *nagyságrendje* – gyakorlati tapasztalatok szerint is – jól jellemzi az algoritmus tényleges futási idejét.

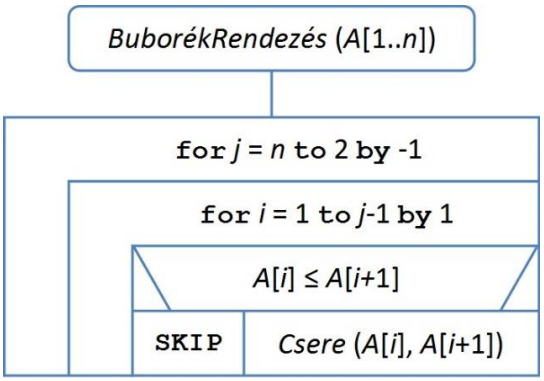
1.1. A buborékrendezés műveletigénye

A rendezés általános feladata jól ismert. A tömbökre megfogalmazott változat egyik legkorábbi (kevéssé hatékony) megoldása a *buborékrendezés*. Az eljárás működésének alapja a szomszédos elemek cseréje, amennyiben az elől álló nagyobb, mint az utána következő. Az első menetben a szomszédos elemek cseréjével „felbuborékoltatjuk” a legnagyobb elemet a tömb végére. A következő iterációs lépésben ugyanezt tesszük az „eggyel rövidebb” tömbben, és így tovább. Utoljára még a tömb első két elemét is megcseréljük, ha szükséges. Az (n–1)-edik iteráció végére elérjük, hogy az elemek nagyság szerint növekvő sorban követik egymást. A rendezés algoritmusa az 1.1. ábrán látható. (Az egyszerűség kedvéért a *Csere* műveletét nem bontottuk három értékadásra.)



1.1. ábra. A buborékrendezés algoritmusa

Az algoritmus műveleteit két kategóriába sorolhatjuk. Az egyikbe tartoznak a ciklusváltozókra vonatkozó műveletek, a másikba pedig az $A[1..n]$ tömb szomszédos elemeinek összehasonlítása és cseréje. Nyilvánvaló, hogy az utóbbiak a meghatározó műveletek. Egyrészt végrehajtásuk lényegesen nagyobb időt vesz igénybe, mint a ciklust adminisztráló utasításoké (különösen, ha a tömb elemeinek mérete jelentős), másrészt mindkét utasítás a belső ciklusban van (még ha a csere feltételhez kötött is), így végrehajtásukra minden iterációs lépésben sor kerülhet. Ezért az összehasonlítások, illetve a cserék számát fogjuk számolni. Ha ezt a döntést meghoztuk, érdemes az algoritmusban szereplő ciklusokat tömörebb, áttekinthetőbb formában, **`for`**-ciklusként újra felírni. Ez a változat látható az 1.2. ábrán.



1.2. ábra. Buborékrendezés (**`for`**-ciklusokkal)

A számolást az egyszerűség kedvéért külön-külön végezzük. Nézzük először az összehasonlítások $\ddot{O}(n)$-nel jelölt számát. A külső ciklus magja $(n–1)$-szer hajtódik végre, a belső ciklus ennek megfelelően rendre $n–1$, $n–2$, …, 1 iterációt eredményez. Mivel a két szomszédos elem összehasonlítása a belső ciklusnak olyan utasítása, amely mindig végrehajtódik, ezért

$$\ddot{O}(n) = (n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = n^2/2 - n/2 .$$

Az 1.3. pontban precízen bevezetjük majd azokat az aszimptotikus fogalmakat és jelöléseket, amelyekkel a műveletigény nagyságrendjét jellemezzük. Most elégedjünk meg annyival, hogy általában elegendő az, ha csak a kifejezés domináns nagyságrendű tagját tartjuk meg, az együttható elhagyásával. Az összehasonlítások számára ennek megfelelően azt mondjuk majd, hogy értéke nagyságrendben n^2 és ezt így jelöljük:

$$\ddot{O}(n) = \Theta(n^2).$$

Megjegyezzük, hogy az összehasonlítások száma a bemenő adatok bármely permutációjára ugyanannyi. Az elemzett műveletek végrehajtási száma a legtöbbször azonban változó az adatok függvényében. Ilyenkor elsősorban a végrehajtások maximális és átlagos számára vagyunk kíváncsiak, de a minimális végrehajtási számot is gyakran meghatározzuk, bár ennek általában nincs jelentősége. Ha $T(n)$-nel jelöljük a műveletigényt, akkor annak *minimális*, *maximális* és *átlagos* értékét rendre $mT(n)$, $MT(n)$ és $AT(n)$ jelöli.

Vizsgáljuk meg most a cserék $Cs(n)$-nel jelölt számát. Ez a szám már nem állandó, hanem a bemenő adatok függvénye. Nevezetesen, a cserék száma megegyezik az $A[1..n]$ tömb elemei között fennálló inverziók számával:

$$Cs(n) = inv(A).$$

Valóban, minden csere pontosan egy inverziót szüntet meg a két szomszédos elem között, újat viszont nem hoz létre. A rendezett tömbben pedig nincs inverzió. Ha a tömb eleve rendezett,

akkor egyetlen cserét sem kell végrehajtani, így a cserék száma a legkedvezőbb esetben nulla, azaz

$$mCs(n) = 0.$$

A legtöbb cserét akkor kell végrehajtani, ha minden szomszédos elempár inverzióban áll, azaz akkor, ha a tömb éppen fordítva, nagyság szerint csökkenő módon rendezett. Ekkor

$$MCs(n) = n(n-1)/2 = \Theta(n^2).$$

A cserék átlagos számának meghatározásához először is feltesszük, hogy a rendezendő számok minden permutációja egyformán valószínű. (Az átlagos műveletigény számításához mindig ismerni kell a bemenő adatok valószínűségi eloszlását, vagy legalább is feltételezéssel kell élni arra nézve!) Az általánosság megszorítása nélkül vehetjük úgy, hogy az 1, 2, ..., n számokat kell rendeznünk, ha elfogadjuk azt a szokásos egyszerűsítést, hogy a rendezendő értékek mind különbözők.

A cserék számának átlagát nyilvánvalóan úgy kapjuk, hogy az 1, 2, ..., n elemek minden permutációjának inverziószámát összeadjuk és osztjuk a permutációk számával:

$$ACs(n) = \frac{1}{n!} \sum_{p \in Perm(n)} inv(p),$$

ahol $Perm(n)$ az n elem összes permutációinak halmazát jelöli. Az összeg meghatározásánál nehézségbe ütközünk, ha megpróbáljuk azt megmondani, hogy adott n-re hány olyan permutáció van, amelyben i számú inverzió található ($0 \le i \le n(n-1)/2$). Ehelyett célszerű párosítani a permutációkat úgy, hogy mindegyikkel párba állítjuk az inverzét, pl. az $p = 1423$ és a $p^R = 3241$ alkot egy ilyen párt. Egy ilyen párban az inverziók száma együtt éppen a lehetséges $\binom{n}{2}$-t teszi ki, pl. $inv(1423) + inv(3241) = 4 + 2 = 6.$ Az állítás igazolására gondoljuk meg, hogy egy permutációban két elem pontosan akkor áll inverzióban, hogy ha az inverz permutációban nincs közöttük inverzió. Írjuk le gondolatban mind az $n!$ permutációt kétszer egymás mellé egy-egy oszlopba, a mondott párosításnak megfelelően. Ekkor minden sorban a két permutáció inverzióinak száma együtt $\binom{n}{2}$, így

$$ACs(n) = \frac{\sum inv(p) + \sum inv(p^R)}{2n!} = \frac{n!\binom{n}{2}}{2n!} = \frac{\binom{n}{2}}{2} = \frac{n(n-1)}{4} = \Theta(n^2).$$

A cserék számának átlaga tehát a legnagyobb érték fele, de nagyságrendben ez így is n^2-es.

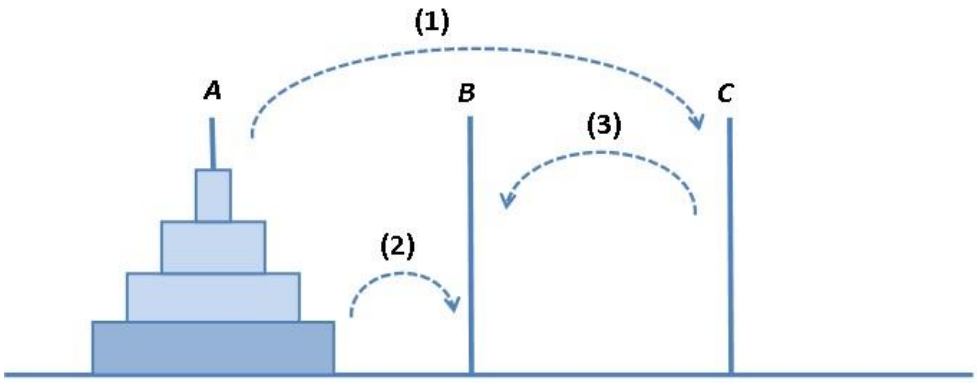
Az elemzés eredménye az, hogy a buborékrendezés a legrosszabb és az átlagos esetben is – mindkét meghatározó művelete szerint – nagyságrendben n^2-es algoritmus.

1.2. A Hanoi tornyai probléma megoldásának műveletigénye

A következő algoritmus, amelyet elemzünk, a Hanoi tornyai probléma megoldása. A probléma régről ismert. Adott három rúd és az elsőn n darab, felfelé egyre csökkenő méretű korong, ahogyan az 1.3. ábrán látható $n = 4$ esetére.

A feladat az, hogy a korongokat át kell helyezni az A rúdról a B-re, a C rúd felhasználásával, oly módon, hogy egyszerre csak egy korongot szabad mozgatni és csak nála nagyobb korongra, vagy üres rúdra lehet áthelyezni.

A feladatnak több különböző megoldása van, köztük olyan (iteratív) heurisztikus algoritmusok, amelyek a mesterséges intelligencia területére tartoznak. Itt a szokásos rekurzív algoritmust ismertetjük.



1.3. ábra. Hanoi tornyai probléma

Számítógépes programok, algoritmusok esetén akkor beszélünk rekurzióról, hogyha az adott eljárás a működése során „meghívja önmagát”, vagyis a számítás egyik lépéseként önmagát hajtja végre, más bemeneti adatokkal, paraméterekkel.

Sok hasznos algoritmus rekurzív szerkezetű, és többnyire az ún. „oszd meg és uralkodj” elv alapján működnek. Ennek a lényege az, hogy a feladatot több részfeladatra bontjuk, amelyek egyenként az eredetihez nagyon hasonlóak, de kisebb méretűek, így könnyebben megoldhatók. A definiált működésnek az lesz a hatása, hogy a részfeladatokat hasonlóan, további egyre kisebb és kisebb részekre osztja az eljárás, amíg el nem éri az elemei feladatok megadott méretét. A már kellően kicsi részfeladatokat közvetlen módon megoldjuk. Ez az elemi lépés gyakran egészen egyszerű. Ezután a rekurzív hívások rendszerében „visszafelé” haladva minden lépésben összegezzük, összevonjuk a részfeladatok megoldását. Az utolsó lépésben, amikor a működés visszaér a kiinduló szintre, megkapjuk az eredeti feladat megoldását.

A módszer általános elnevezése onnan ered, hogy a rekurzió minden szintjén három alapvető lépést hajt végre:

- felosztja a feladatot több részfeladatra,
- „uralkodik” a részfeladatokon, rekurzív módon való megoldásukkal; amennyiben a feladat mérete kellően kicsiny, közvetlenül megoldja,
- összevonja a részfeladatok megoldását az eredeti feladat megoldásává.

Ennek az általános elvnek megfelelően a Hanoi tornyai probléma rekurzív megoldási elve a következő:

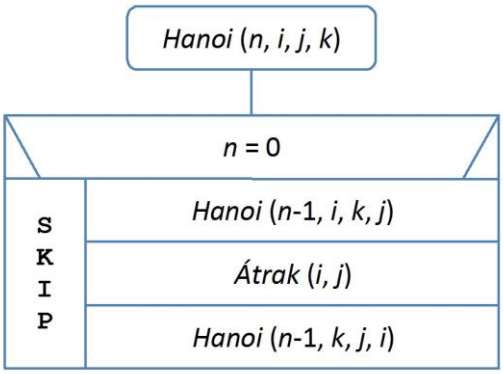
1. A felső $n-1$ korongot helyezzük át az *A* rúdról a *C*-re a megengedett lépésekkel úgy, hogy a *B* rudat vesszük segítségül. (Ez az eredetihez hasonló, de kisebb méretű feladat.)
2. Az egyedül maradt alsó korongot tegyük át az *A* rúdról az üres *B*-re. (Ez az elemi méretű feladat közvetlen megoldása.)
3. Vigyük át a *C* rúdon található $n-1$ korongot a *B*-re az *A* rúd igénybe vételével, hasonlóan ahhoz, ahogyan az 1. pontban eljártunk.

A rekurzív eljárás működése 1. és 3. lépésben szereplő $n-1$ korongot hasonló szemlélettel $n-2$ korong kétszeri mozgatására, valamint egy korong áthelyezésére bontja, és így tovább.

A rekurzióból való "kijárat" megfogalmazása természetes módon egyetlen korongra adódna: ha $n = 1$, akkor egyszerűen tegyük át a korongot a rendeltetési helyére. Egyszerűbb formájú algoritmust kapunk, amelyet elemezni is könnyebb, ha "még lejjebb" adjuk meg a kijáratot a rekurzióból: ha $n = 0$, akkor nem kell semmit sem tennünk. Erre az ad lehetőséget, hogy így az $n = 1$ eset is kezelhető a fenti három lépéssel: az 1. és a 3. lépés üressé válik, a 2. lépés pedig tartalmazza az elemi áthelyezés műveletét.

Általános szabályként jegyezzük meg azt, hogy a rekurzióból való kijáratot „engedjük minél lejjebb", azaz a szóban forgó változó minél kisebb értékére próbáljuk azt megadni (pl. a kínálkozó $n = 1$ helyett az $n = 0$ esetre). Arra törekszünk tehát, hogy minél kisebb legyen az a részfeladat, amelyet már közvetlenül oldunk meg.

Írjuk meg ezután az eljárást! A *Hanoi* (*n*, *i*, *j*, *k*) rekurzív eljárás *n* számú korongot átvisz az *i*-edik rúdról a *j*-edikre, a *k*-adik rúd felhasználásával. Az eljárást "kívülről" a konkrét feladatnak megfelelő paraméterekkel kell meghívni, pl. az ábrán látható esetben így: Hanoi (4, 1, 2, 3). Az eljárás struktogramja a 1.4. ábrán látható. A rekurzív eljárás önmagát hívja egészen addig, amíg *n* nagyobb nullánál. Amikor $n = 1$, akkor is ez történik, de a rekurzív hívások végrehajtása az $n = 0$ ágon már egy-egy üres utasítást eredményez. Az *Átrak* (*i*, *j*) utasítás jelöli azt az elemi műveletet, amellyel egyetlen korongot átteszünk az *i*-edik rúdról a *j*-edikre.



1.4.ábra. A Hanoi tornyai probléma megoldása

Meghatározzuk az *n* korong átpakolásához szükséges lépésszámot. Az algoritmus rekurzív megfogalmazása szinte kínálja az átrakások számára vonatkozó rekurzív egyenletet:

$$T(n) = \begin{cases} 2T(n-1)+1 & , \text{ ha } n \geq 1, \\ 0 & , \text{ ha } n = 0. \end{cases}$$

Ha kiszámítjuk $T(n)$ értékét néhány *n*-re, akkor azt sejthetjük, hogy $T(n) = 2^n - 1$. Ezt teljes indukcióval könnyen bebizonyíthatjuk. Azt kaptuk, hogy a Hanoi tornyai egy exponenciális műveletigényű probléma: $T(n) = \Theta(2^n)$.

(A legenda szerint Indiában, egy ősi város templomában a szerzetesek ezt a feladatot 64 korongra kezdték el valamikor megoldani azzal, hogy amikor a rakosgatás végére érnek, elkövetkezik a világ vége. Ha minden korongot 1 mp alatt helyeznek át, akkor a 64 korong teljes átpakolása nagyságrendben 600 milliárd évet venne igénybe. Ha 10^{-6} mp-cel számolunk, ami már a számítógépek sebessége, akkor is nem egészen 600 ezer évre lenne szükség a korongok átrendezéséhez. A nagyságrendek érzékeltetésére: az univerzumról úgy tudjuk, hogy kevesebb, mint 14 milliárd éves, a másodiknak kiszámolt időtartam pedig a homo sapiens megjelenése óta eltelt idővel egyezik meg nagyságrendben.)

1.3. Függvények aszimptotikus viselkedése*

(Azt javasoljuk, hogy az olvasó ne tekintse befejezettnek alapszintű tanulmányait az algoritmusok témakörében, amíg ezzel az alfejezettel meg nem ismerkedett! Célszerű most átolvasni ezt a *-gal jelölt 1.3. részt, majd később – talán többször is – visszatérni ide a mélyebb megértés céljával.)

Kiszámítottuk két algoritmus műveletigényét a bemenő adatok méretének függvényében. Szeretnénk pontosabb matematikai fogalmakra támaszkodva beszélni az algoritmusok hatékonyságáról. Ebben a pontban ennek megalapozása történik. Először is alkalmasan megválasztjuk az algoritmusok műveletigényét, illetve lépésszámát jelentő függvények értelmezési tartományát és értékkészletét.

A továbbiakban legyen f olyan függvény, amelyet a természetes számok halmazán értelmezünk és nem-negatív valós értékeket vesz fel:

$$f : \mathbf{N} \to \mathbf{R}_0^+,$$

ahol $\mathbf{N} = \{0,1,2,\ldots\}$.

Definiáljuk egy adott g függvény esetén azon függvények osztályait, amelyek nagyságrendben rendre nem nagyobbak, kisebbek, nem kisebbek, nagyobbak, mint g, illetve g-vel azonos nagyságrendűek. Az osztályok jelölései és nevei: O „nagy ordó” vagy „ordó”, o ”kis ordó”, Ω „nagy omega”, ω „kis omega” és Θ theta.

1.3.1. Definíciók

1. Egy adott g függvény esetén $O(g)$-vel jelöljük függvényeknek azt a halmazát, amelyre

   $$O(g) = \{f : \text{létezik } c > 0 \text{ és } n_0 \geq 0 \text{ úgy, hogy minden } n \geq n_0 \text{ esetén } f(n) \leq cg(n)\}$$

   teljesül. Ha $f \in O(g)$, akkor azt mondjuk, hogy g aszimptotikus felső korlátja f-nek. Ebben az esetben szokásos módon inkább az $f = O(g)$ jelölést alkalmazzuk. Valamint, használjuk a következő függvényhalmazt is:

   $$o(g) = \{f : \text{minden } c > 0 \text{ esetén létezik } n_0 \geq 0, \text{hogy minden } n \geq n_0 \text{ esetén } f(n) < cg(n)\}$$

2. Hasonlóan, jelölje $\Omega(g)$ azon függvények halmazát, amelyekre

   $$\Omega(g) = \{f : \text{létezik } c > 0 \text{ és } n_0 \geq 0 \text{ úgy, hogy minden } n \geq n_0 \text{ esetén } f(n) \geq cg(n)\}$$

   áll fenn. Ha $f \in \Omega(g)$ – szokásos jelöléssel $f = \Omega(g)$ –, akkor azt mondjuk, hogy g aszimptotikus alsó korlátja f-nek. Az itt megjelenő másik függvényhalmaz pedig:

   $$\omega(g) = \{f : \text{minden } c > 0 \text{ esetén létezik } n_0 \geq 0, \text{hogy minden } n \geq n_0 \text{ esetén } f(n) > cg(n)\}$$

3. Végül, $\Theta(g)$ jelöli azt a függvényosztályt, amelyet a

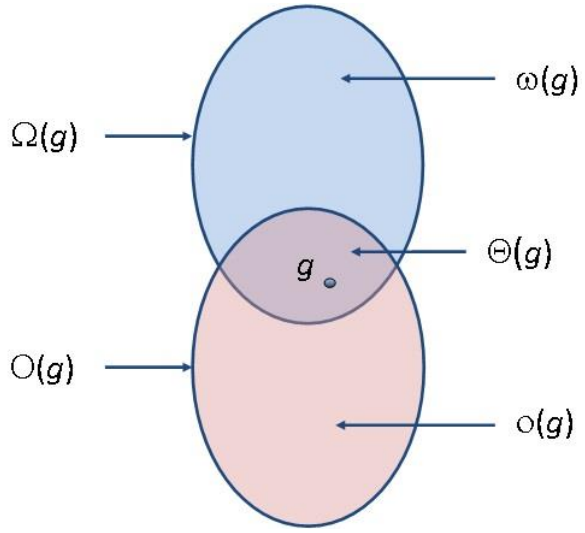
   $$\Theta(g) = \{f : \text{létezik } c_1, c_2 > 0 \text{ és } n_0 \geq 0, \text{hogy minden } n \geq n_0 \text{ esetén } c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

   összefüggés ír le. Ha $f \in \Theta(g)$ – szokásos jelöléssel $f = \Theta(g)$ –, akkor azt mondjuk, hogy g aszimptotikusan éles korlátja f-nek.

   A definíciók alapján teljesül a következő egyenlőség:

   $$\Theta(g) = \mathrm{O}(g) \cap \Omega(g).$$

Ha egy tetszőleges, rögzített g függvényt tekintünk, akkor a többi függvény általában besorolható a most definiált osztályok némelyikébe. (Megjegyezzük, hogy vannak nem összehasonlítható függvények is, pl. $g(n)=1$ és $f(n)=n(1+\sin n)$, ill. $g(n)=n$ és $f(n)=n^{1+\sin n}$ minden n-re. Könnyen belátható ezekre, hogy sem $f=O(g)$, sem pedig $f=\Omega(g)$ nem teljesül.) A nagyságrendi viszonyokról az 1.5. ábra ad szemléletes képet.



1.5. ábra. A függvényosztályok egymáshoz való viszonya

A függvények aszimptotikus viszonyainak jelen vizsgálatához az szolgál alapul, hogy kellően nagyméretű bemenet esetén egy algoritmus futási idejének (ill. az azt leíró függvénynek) csak a nagyságrendje lényeges. Viszonylag kisméretű bemeneteket leszámítva tehát az aszimptotikusan leghatékonyabb algoritmus lesz a ténylegesen leggyorsabb!

1.3.2. Visszavezetés határértékekre

Valamely adott f és g függvény aszimptotikus viszonyát a $\lim\limits_{n\to\infty}\frac{f(n)}{g(n)}$ határérték – amennyiben létezik – a következőképpen határozza meg:

1. Ha $\lim\limits_{n\to\infty}\frac{f(n)}{g(n)}=\begin{cases}0, vagy\\ c>0\end{cases}$, akkor $f=O(g)$. $\quad\lim\limits_{n\to\infty}\frac{f(n)}{g(n)}=0$ esetén $f=o(g)$.
2. Ha $\lim\limits_{n\to\infty}\frac{f(n)}{g(n)}=\begin{cases}\infty, vagy\\ c>0\end{cases}$, akkor $f=\Omega(g)$. $\quad\lim\limits_{n\to\infty}\frac{f(n)}{g(n)}=\infty$ esetén $f=\omega(g)$.
3. Ha $\lim\limits_{n\to\infty}\frac{f(n)}{g(n)}=c>0$, akkor $f=\Theta(g)$.

Megjegyzés: A $\lim\limits_{n\to\infty}\frac{f(n)}{g(n)}$ határérték általában létezik, de pl. ha

$$f(n)=\begin{cases}1, ha\ n\equiv 1(\mathrm{mod}\,2)\\ 2, ha\ n\equiv 0(\mathrm{mod}\,2)\end{cases}$$

és $g(n)=1$ minden n-re, akkor nem létezik a határérték, de a két függvény aszimptotikus viszonya mégis megállapítható, ebben az esetben $f=\Theta(g)$.

1.3.3. L'Hospital szabály alkalmazása

Ha az adott f és g függvényekre $\lim_{n\to\infty} f(n) = \infty$ és $\lim_{n\to\infty} g(n) = \infty$, és mindkét függvény valós kiterjesztése differenciálható, akkor gyakran alkalmazzuk a L'Hospital szabályt a $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ határérték meghatározására:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

1.3.4. Tulajdonságok

A $\mathrm{O}, o, \Theta, \omega, \Omega$ jelöléseket függvények közötti bináris relációként is felfoghatjuk (pl. $\mathrm{O} = \{(f,g) : f = \mathrm{O}(g)\}$). Így a relációkra vonatkozó ismert definíciók értelmezhetők rájuk, és beláthatók a következő állítások:

1. $\mathrm{O}, o, \Theta, \omega, \Omega$ mind tranzitív: *f*, *g*, *h* függvényekre pl.: $f = \omega(g) \wedge g = \omega(h) \Rightarrow f = \omega(h)$.
2. $\mathrm{O}, \Theta, \Omega$ mindegyike reflexív: pl. $f = \Omega(f)$.
3. Θ szimmetrikus: $f = \Theta(g) \Leftrightarrow g = \Theta(f)$.
4. O és Ω, valamint o és ω „felcserélten szimmetrikusak: pl. $f = \mathrm{O}(g) \Leftrightarrow g = \Omega(f)$.
5. Rögzített *h* függvény mellett $\mathrm{O}(h), o(h), \Theta(h), \omega(h), \Omega(h)$ halmazok zártak az összeadásra és a (pozitív) számmal való szorzásra nézve: pl. $f = \omega(h) \wedge g = \omega(h) \Rightarrow f + g = \omega(h)$; és $f = \omega(h) \wedge c > 0 \Rightarrow cf = \omega(h)$.
6. Összegben a nagyobb függvény határozza meg az aszimptotikát: $\max\{f,g\} = \Theta(f+g)$. (Itt a *max* jelölés az aszimptotikusan nagyobb függvényt jelenti.) Ha ezt a szokásos alakot átírjuk az $f + g = \Theta(\max\{f,g\})$ formába, akkor könnyen kiolvasható belőle az, hogy egy szekvencia műveletigényének nagyságrendjét a nagyobb műveletigényű tag határozza meg.

1.3.5. Függvényosztályok

A fenti 1., 2. és 3. tulajdonságok miatt Θ – mint bináris reláció – ekvivalenciareláció a függvények halmazán, tehát osztályokra bontja azt. Az egyes osztályok reprezentálhatók a legegyszerűbb függvényükkel, pl.: $\Theta(1), \Theta(n), \Theta(n^2), \Theta(\sqrt{n}), \ldots$

Megjegyzés: A gyakorlatban – amennyiben nem okoz félreértést – az egyszerűség kedvéért a g, $\mathrm{O}(g)$, $o(g)$, $\Theta(g)$, $\omega(g)$, $\Omega(g)$ jelölések helyett gyakran a $g(n)$, $\mathrm{O}(g(n))$, $o(g(n))$, $\Theta(g(n))$, $\omega(g(n))$, $\Omega(g(n))$ jelöléseket használjuk. Így pl. a $3n^2 + 7n = \Theta(n^2)$ jelentése: $f = \Theta(h)$, ahol $f(n) = 3n^2 + 7n$ és $h(n) = n^2$ minden n-re.

Néhány további könnyen belátható tulajdonság:

- Polinom esetén a legnagyobb fokú tag a meghatározó (lásd: fenti 6. tulajdonság):

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = \Theta(n^k).$$

- Bármely két (1-nél nagyobb) alapszámú logaritmikus függvény aszimptotikusan egyenértékű: $\log_a n = \Theta(\log_b n)$ (a>1 és b>1). Ezért az alap feltüntetése nem szükséges: $\log_a n = \Theta(\log n)$

- Hatványfüggvények esetén különböző kitevők különböző függvényosztályokat jelölnek ki: $a \geq 0$ és $\varepsilon > 0$ esetén $n^a = \mathrm{O}(n^{a+\varepsilon})$ és $n^a \neq \Theta(n^{a+\varepsilon})$, vagyis $n^a = o(n^{a+\varepsilon})$.

1.3.6. Az aszimptotikus jelölések használata

Az aszimptotikus jelölések használatában elterjedt néhány gyakori hiba. Ezek általában nem zavarók, mégis fel kell hívni a figyelmet rájuk. A kritikával azonban már csak azért is óvatosan kell bánni, mert a szerzők egy része kizárólag az O-t használja.

1. Leggyakoribb „hiba” az, hogy algoritmusok műveletigényének jellemzésekor O-t használnak, de Θ-t értenek alatta. Az így leírt egyenlőség általában igaz, de mást (kevesebbet) fejez ki, mint amit általában közölni szeretnének általa. Pl. a buborékrendezésre igaz, hogy $MT(n) = \mathrm{O}(n^2)$, de többet mond az, hogy $MT(n) = \Theta(n^2)$.

2. A $T(n)$ jelölés használata is hibát rejthet. $T(n)$ az összes lehetséges futási időt magában foglalja, amelyek nagyságrendben különbözőek lehetnek. Pl. ahogy korábban láttuk, a buborékrendezésre igaz (*Cs* helyett *T*-vel), hogy: $MT(n) = \Theta(n^2)$ és $AT(n) = \Theta(n^2)$. A legkedvezőbb esetben azonban nincs csere, vagyis $mT(n) = 0$. Ekkor a $T(n) = \Theta(n^2)$ (felületes) kijelentés nem igaz, helyette a $T(n) = \mathrm{O}(n^2)$ írásmódot kellene használni, mely az összes esetre helyes lesz. Ez azonban kevesebb információt tartalmaz, mint a fenti két egyenlőség, ezért ajánlatos $MT(n)$-t és $AT(n)$-t használni.

3. Néhány tipikus használat:

   a) $2n^2 + 3n + 1 = \Theta(n^2)$, $\qquad 3n + 1 = \mathrm{O}(n^2)$, de $3n + 1 \neq \Theta(n^2)$

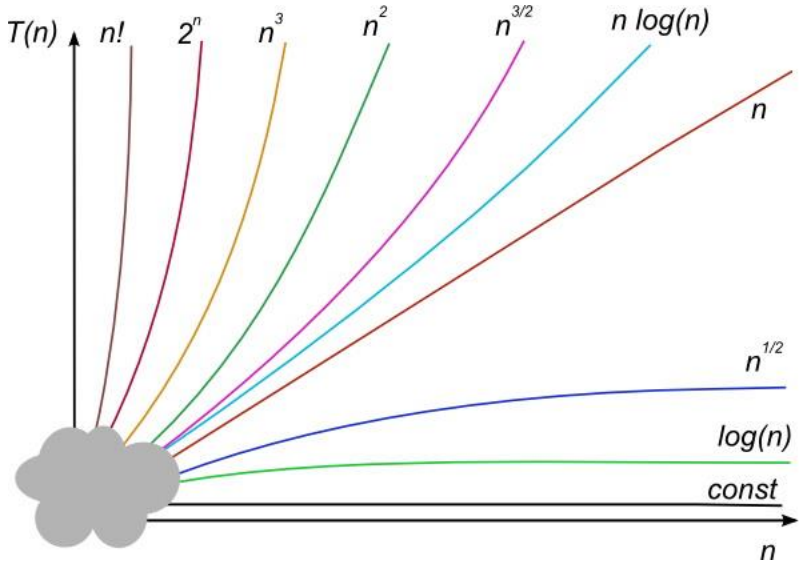
   b) $2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$

   c) A logaritmikus keresés futási idejére fennáll a következő (közelítő) rekurzív egyenlet: $T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$.

1.4. Algoritmusok műveletigényének tipikus nagyságrendjei

Néhány jellegzetes, és a gyakorlatban gyakran előforduló függvény menetét és egymáshoz viszonyított aszimptotikus nagyságrendjét szemlélteti az 1.6. ábra (nem mérethűen, főleg nem az origó közelében). Ezek a függvények aszimptotikusan mind különböző nagyságrendűek.

A hatványfüggvények – az ábrán is látható – egyik fontos tulajdonsága, hogy bármely $\alpha > 1$ kitevő esetén n^α az $n \cdot \log n$ és az a^n ($a > 1$) közé, míg ha $0 < \alpha < 1$, akkor n^α a $\log n$ és az n közé esik aszimptotikusan.



1.6. ábra. A gyakran előforduló nagyságrendek grafikonja

Megadunk néhány példát az algoritmusok aszimptotikus futási idejére.

- Verem vagy sor bármely művelete $\Theta(1)$
- Logaritmikus (=bináris) keresés $\Theta(\log n)$
- Prímszámteszt ($n^{1/2}$-ig) $\Theta\ (n^{1/2})$
- Lineáris keresés $\Theta(n)$
- Kupacrendezés $\Theta(n\ \log n)$
- Shell rendezés $\Theta(n^{3/2})$
- Buborékrendezés $\Theta(n^2)$
- Mátrixszorzás $\Theta(n^3)$
- Hanoi tornyai $\Theta(2^n)$
- Utazó ügynök probléma $\Theta(n!)$

A műveletigények gyakran előforduló nagyságrendjeinek értékeit az 1.7. ábrán látható tartalmazza néhány jellegzetes n értékre, mindössze 1024-ig. Az n=12 nevezetes érték az $n!$ függvény esetében. A 12! még egészként ábrázolható, a 13! már csak lebegőpontos számként, és egy $n!$ műveletigényű probléma egzakt teljes körű megoldása általában az n=12 értékre még kivárható egy erős asztali gépen, de nagyobb értékekre már nem. Ezt a 10^8 nagyságrendű határt a 2^n függvény 29-nél éri el, az n^3-ös műveletigényű algoritmusok pedig 500 felett nem sokkal. Figyelemre méltó még az is, hogy az n^2 és az $n\log(n)$ értékek n=1024 esetén (ami a gyakorlatban csekély adatmennyiség) már két nagyságrendben térnek el. (Néhány olyan értéket, amely a Matlab programban túlcsordulást okozott, nem helyettesítettük kiszámolt értékekkel, hiszen ezeknek a nagyságrendeknek nincs valós értelmük; nem csak az algoritmusok világában, de a földi tér-idő mennyiségek szempontjából sem.)

Látszik, miért mondják az informatikusok, hogy legfeljebb az n^3-ös algoritmusoknak vesszük hasznát a gyakorlatban. Az is nyilvánvaló, hogy nagyobb n-ekre miért érdemes $n\log(n)$ műveletigényű rendezéseket használni az n^2-es eljárásokkal szemben, illetve miért érdemes a gráfos algoritmusokat gyorsítani – például minimumválasztó kupac adatszerkezet alkalmazásával –, hiszen ott is ez a két utóbbi nagyságrend állítható egymással szembe.

n	log(n)	n^(1/2)	n	n log(n)	n^(3/2)	n^2	n^3	2^n	n!
...									
10	3,32	3,16	10	33,22	31,62	100	1000	1024	3628800
11	3,46	3,32	11	38,05	36,48	121	1331	2048	39916800
12	3,58	3,46	12	43,02	41,57	144	1728	4096	479001600
13	3,70	3,61	13	48,11	46,87	169	2197	8192	6,227E+09
...									
29	4,86	5,39	29	140,88	156,17	841	24389	536870912	8,842E+30
30	4,91	5,48	30	147,21	164,32	900	27000	1,074E+09	2,653E+32
31	4,95	5,57	31	153,58	172,60	961	29791	2,147E+09	8,223E+33
32	5	5,66	32	160	181,02	1024	32768	4,295E+09	2,631E+35
...									
64	6	8	64	384	512	4096	262144	1,845E+19	1,269E+89
...									
128	7	11,31	128	896	1448,15	16384	2097152	3,403E+38	3,86E+215
...									
256	8	16	256	2048	4096	65536	16777216	1,158E+77	****
...									
512	9	22,63	512	4608	11585,24	262144	134217728	1,34E+154	****
...									
1024	10	32	1024	10240	32768	1048576	1,074E+09	****	****

1.7. ábra. A gyakran előforduló nagyságrendek értékeinek táblázata

A táblázat valós értékei jól érzékeltetik azt, hogy mennyire fontos az algoritmusok műveletigényével tisztában lenni, és azt a lehetőség szerint minél alacsonyabbra választani.