ELSŐ FEJEZET

# Az alapok

Az algoritmusok világába induló utazás előkészületekkel és háttér-információkkal kezdődik. Tudnunk kell néhány dolgot, mielőtt a könyvben található algoritmusokkal és adatstruktúrákkal megismerkedhetnénk. Bizonyára ég bennünk a vágy, hogy mielőbb belemerüljünk, de ennek a fejezetnek az elolvasása a könyv többi részét teszi érthetőbbé, mert olyan fontos alapfogalmakat tartalmaz, amelyek elengedhetetlenek a kódok és az algoritmusok elemzésének megértéséhez.

A fejezetben a következő témaköröket tárgyaljuk:

- mi az algoritmus,
- az algoritmusok szerepe a szoftverekben és mindennapi életünkben,
- mit jelent az algoritmus bonyolultsága,
- az algoritmusbonyolultság széles osztályai, amelyek segítségével gyorsan megkülönböztethetjük ugyanannak a problémának a különböző megoldásait,
- a „nagy O” jelölés,
- mi az egységtesztelés, és miért fontos,
- hogyan kell a JUnit segítségével egységteszteket írni.

## Az algoritmusok definíciója

Az talán már tudjuk, hogy az algoritmusok a számítástechnika fontos részét képezik, de egészen pontosan mik is azok? Mire használhatók? Kell egyáltalán törődnünk velük?

Az algoritmusok valójában nem korlátozódnak a számítástechnika világára; mindennapi életünkben is alkalmazunk algoritmusokat. Egyszerű meghatározással az *algoritmus* valamilyen feladat végrehajtásához szükséges, jól meghatározott lépések sora. Ha tortát sütünk, és a recept utasításait követjük, tulajdonképpen egy algoritmust használunk.

Az algoritmusok segítségével egy rendszert lehetőség szerint közbenső, átmeneti állapotok sorozatán keresztül adott állapotból egy másikba vihetünk. Az életből vett másik példa az egyszerű egész szorzás művelete. Noha általános iskolában mindannyian bemagoltuk a szorzótáblákat, a szorzási folyamatot összeadások sorozatának is

tekinthetjük. Például az 5 × 2 kifejezés a 2 + 2 + 2 + 2 + 2 (illetve az 5 + 5) kifejezés gyorsírásos változata. Vagyis bármely két egész szám, például *A* és *B* esetén elmondhatjuk, hogy az *A* × *B* annyit jelent, hogy *B*-t *A*-szor önmagához adjuk. Ezt az alábbi lépések soraként fejezhetjük ki:

1. Inicializáljunk egy harmadik egész számot, *C*-t 0-ra.
2. Ha *A* nulla, akkor készen vagyunk, és az eredményt *C* tartalmazza. Ha nem ez a helyzet, haladjunk tovább a 3. lépéshez.
3. Adjuk *B* értékét *C*-hez.
4. Csökkentsük *A* értékét.
5. Lépjünk a 2. lépéshez.

Vegyük észre, hogy a torta receptjével ellentétben az összeadással szorzó algoritmus az 5. lépésben visszakanyarodik a 2. lépéshez. A legtöbb algoritmusban felfedezhetünk valamilyen ciklikusságot, amelynek a segítségével számításokat vagy egyéb műveleteket ismétlünk. Az *iterációt* és a *rekurziót* – a ciklusok két fő típusát – a következő fejezet részletesen ismerteti.

Az algoritmusokat gyakran pszeudokódként emlegetjük, amely még nem programozó személyek számára is könnyen érthető, kitalált programozási nyelv. Az alábbi kód a `Multiply` függvényt mutatja be, amely két egész szám – A és B – szorzatát, A × B-t adja vissza, és csak összeadást használ. A pszeudokód a két egész szám összeadással való szorzásának műveletét mutatja be:

```
Function Multiply(Integer A, Integer B)
  Integer C = 0

  While A is greater than 0
    C = C + B
    A = A - 1
  End

  Return C
End
```

A szorzás az algoritmusok nagyon egyszerű példája. A legtöbb alkalmazásban ennél jóval bonyolultabb algoritmusokkal találkozhatunk. A bonyolult algoritmusok megértése nehezebb, és ezért azok nagyobb valószínűséggel tartalmaznak hibákat. (A számítógép-tudomány nagy része tulajdonképpen azt próbálja igazolni, hogy bizonyos algoritmusok helyesen működnek.)

Nem minden helyzetben alkalmazhatunk algoritmusokat. Előfordulhat, hogy adott problémát több algoritmussal is megoldhatunk. Néhány megoldás egyszerű, mások bonyolultabbak, és egyik megoldás hatékonyabb lehet a többinél. Nem min-

dig a legegyszerűbb megoldás a legnyilvánvalóbb. Bár a szigorú, tudományos elemzés mindig jó kiindulópont, gyakran találjuk magunkat az *elemzési paralízis* helyzetében. Néha a jól bevált régimódi kreativitásra van szükség. Próbáljunk ki több megközelítést, és járjunk utána megérzéseinknek. Vizsgáljuk meg, miért működnek bizonyos esetekben az aktuális megoldási kísérletek, más esetekben pedig nem. Nem véletlen, hogy a számítógép-tudomány és a szoftvermérnökség egyik alapművének címe *A számítógép-programozás művészete* (írta Donald E. Knuth). A könyvben ismertetett algoritmusok nagy része *determinisztikus* – azaz az algoritmus eredménye a bemenetek alapján pontosan meghatározható. Előfordul azonban, hogy a probléma olyan bonyolult, hogy az idő és az erőforrások tekintetében a pontos megoldás megkeresése túlságosan nagy ráfordítást igényel. Ilyen esetekben a *heurisztikus* megközelítés lehet a hasznosabb. A tökéletes megoldás keresése helyett a heurisztikus megközelítés a probléma jól ismert tulajdonságai alapján állít elő egy közelítő megoldást. A heurisztikák segítségével kiválogathatjuk az adatokat, eltávolíthatunk vagy figyelmen kívül hagyhatunk lényegtelen értékeket, hogy az algoritmus számítási szempontból költségesebb részeinek kisebb adathalmazon kelljen működniük.

A heurisztika egyik hétköznapi példája az utca egyik oldaláról a másikra való átkelés a világ különböző országaiban. Észak-Amerikában és Európa nagy részén a járművek az út jobboldalán haladnak. Ha életünk nagy részét eddig Észak-Amerikában töltöttük, akkor az úttesten való átkelés előtt kétségtelenül előbb balra, majd jobbra nézünk. Ha Ausztráliában balra néznénk, azt látnánk, hogy szabad az út, majd lelépnénk a járdáról, és nagy meglepetés érhetne bennünket, mert Ausztráliában az Egyesült Királysághoz, Japánhoz, illetve több más országhoz hasonlóan az út bal oldalán haladnak a járművek.

A járművek menetirányát országtól függetlenül igen könnyen megállapíthatjuk, ha egy pillantást vetünk a parkoló járművekre, és megfigyeljük, merre néznek. Ha az autók balról jobbra sorakoznak egymás után, akkor nagy valószínűséggel az úton való átkelés előtt előbb balra, majd jobbra kell figyelnünk. Ha ellenben a parkoló autók jobbról balra sorakoznak, akkor először jobbra, aztán balra kell néznünk az átkelés előtt. Ez az egyszerű heurisztika az *esetek túlnyomó részében* beválik. Sajnálatos módon azonban vannak helyzetek, amikor a heurisztika kudarcot vall: nem látunk parkoló autót, az autók összevissza parkolnak (ez Londonban elég gyakran megesik), vagy az autók az út bármelyik oldalán haladhatnak, mint Bangalore-ban.

Tehát a heurisztika használatának nagy hátulütője, hogy nem tudjuk minden helyzetben meghatározni, hogyan viselkedik – mint azt az előző példában láttuk. Ez az algoritmus bizonytalansági szintjéhez vezet, amely az alkalmazástól függően vagy elviselhető, vagy nem.

Végeredményben bármilyen problémát próbálunk megoldani, valamilyen algoritmusra kétségtelenül szükségünk lesz; minél egyszerűbb, pontosabb és érthetőbb az algoritmus, annál könnyebben meghatározhatjuk, hogy megfelelően működik-e, és a teljesítménye is elfogadható-e.

# Az algoritmusok bonyolultsága

Miután megalkottuk, hogyan tudjuk meghatározni egy új, korszakalkotó algoritmus hatékonyságát? Nyilvánvaló elvárás, hogy szeretnénk kódunkat a lehető leghatékonyabbnak tudni, tehát be kell bizonyítanunk, hogy a hozzá fűzött reményeknek megfelelően működik. De pontosan mit értünk hatékonyság alatt? Processzoridőt, memóriafelhasználást, lemez bemenet-kimenetet? És hogyan mérhetjük az algoritmusok hatékonyságát?

Az algoritmusok hatékonyságának vizsgálata során a leggyakrabban elkövetett hibák egyike, hogy a *teljesítményt* (a processzoridő/memória/lemezterület-foglalás mennyiségét) összekeverik a *bonyolultsággal* (az algoritmus mérhetőségével). A tény, hogy az algoritmus 30 milliszekundum alatt 1000 rekordot dolgoz fel, nem az algoritmus hatékonyságának fokmérője. Noha igaz, hogy végeredményben az erőforrásfogyasztás is fontos, az olyan tényezőket, mint a processzoridő a kódon kívül a mögöttes hardver – amelyen a kód fut – hatékonysága és teljesítménye, valamint a gépi kód generálásához használt fordító is erősen befolyásolja. Sokkal fontosabb annak megállapítása, hogyan viselkedik az adott algoritmus a probléma méretének növekedésével. Ha például a feldolgozni kívánt rekordok száma megkétszereződik, annak milyen hatása van a feldolgozási időre? Eredeti példánkhoz visszatérve, ha egy algoritmus 1000 rekordot 30 milliszekundum alatt dolgoz fel, míg egy másik algoritmus 40 milliszekundum alatt, akkor az első algoritmust tekinthetjük „jobbnak". Ha azonban az első algoritmus 300 milliszekundum alatt 10 000 rekordot (tízszer annyit) dolgoz fel, de a második algoritmus 80 milliszekundum alatt ugyanennyit, akkor választásunkat felül kell vizsgálni.

Általánosságban elmondhatjuk, hogy a bonyolultság az adott funkció végrehajtásához szükséges meghatározott erőforrás-mennyiség fokmérője. Lehetséges – és gyakran hasznos – a bonyolultságot lemez bemenet-kimenet, memóriafelhasználás tekintetében mérni, de a könyvben a bonyolultság processzoridőre gyakorolt hatását vizsgáljuk. A bonyolultság fogalmát tovább finomítjuk az adott funkció végrehajtásához szükséges számítások vagy műveletek számának mértékére.

Érdekes módon a műveletek pontos számát rendszerint nem szükséges mérnünk. Sokkal fontosabb az, hogyan változik a végrehajtott műveletek száma a probléma méretével. Mint az előző példában: ha a probléma mérete egy nagyságrenddel nő, ez hogyan befolyásolja az egyszerű funkció végrehajtásához szükséges műveletek számát? Ugyanannyi műveletre lesz szükség? Vagy kétszer annyira? A szám a probléma méretével lineárisan nő? Vagy exponenciálisan? Ezt kell az algoritmusbonyolultság alatt értenünk. Az algoritmus bonyolultságának mérésével a teljesítményét próbáljuk megjósolni: a bonyolultság kihat a teljesítményre, de ez fordítva nem igaz.

A könyvben az algoritmusok és adatstruktúrák bemutatása során a bonyolultságukat is elemezni fogjuk. Az elemzések megértéséhez nem lesz szükség matematikai doktorátusra. Az egyszerű elméleti bonyolultságelemzést minden esetben könnyen követhető empirikus eredmények követik tesztesetek formájában, amelyeket mi magunk is kipróbálhatunk, és kísérletezhetünk a bemenet módosításával, hogy kitapasztalhassuk a szóban forgó algoritmus hatékonyságát. A legtöbb esetben az átlagos bonyolultság adott – a kód elvárt tipikus esetbeli futási sebessége. Számos esetben a legrosszabb esetbeli és a legjobb esethez tartozó idő is adott. Az, hogy a legjobb, a legrosszabb és az átlagos esetek közül melyik a meghatározó, részben az algoritmustól függ, de a legtöbbször attól az adattípustól, amelyet az algoritmus használ.. Mindenesetre fontos megjegyeznünk, hogy a bonyolultság nem az elvárt teljesítmény pontos mértékét biztosítja, hanem az elérhető teljesítményt szorítja bizonyos határok vagy korlátok közé.

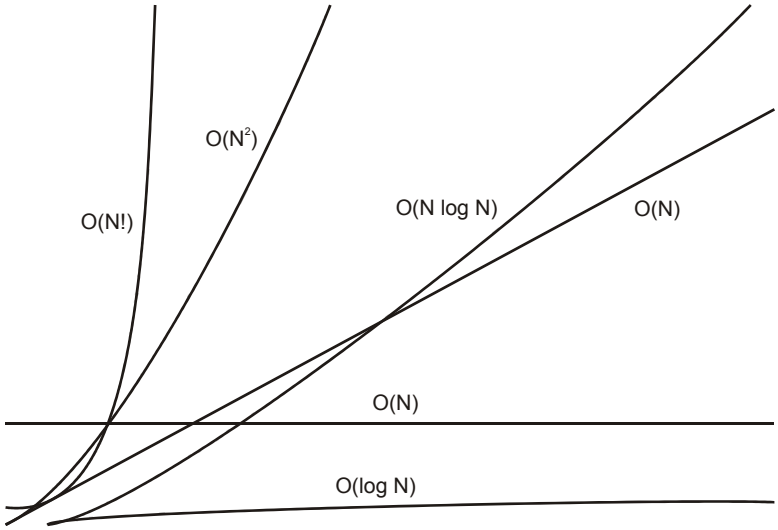
# A nagy O jelölés

Mint már korábban említettük, a műveletek pontos száma valójában nem fontos. Az algoritmus bonyolultságát a funkció végrehajtásához szükséges műveletek számának *nagyságrendjével* definiálhatjuk a nagy `O` jelöléssel – *order of* (nagyságrend) – innen a nagy O. Az O mögötti kifejezés a probléma méretét jelölő `N`-hez képest a relatív növekedést jelenti. Az alábbi lista néhány gyakran alkalmazott nagyságrendet mutat be, a későbbiekben mindegyikre részletesen visszatérünk.

- `O(1)`: az „ordó 1” konstans futási idejű függvényt jelent
- `O(N)`: az „ordó N” lineáris futási idejű függvényt jelent
- `O(N²)`: az „ordó N négyzet” kvadratikus futási idejű függvényt jelent
- `O(log N)`: az „ordó logaritmus N” logaritmikus futási idejű függvényt jelent
- `O(N log N)`: az „ordó N logaritmus N” a probléma méretével és a logaritmikus idővel arányos futási idejű függvényt jelent
- `O(N!)`: az „ordó N faktoriális” faktoriális futási idejű függvényt jelent

Természetesen a fenti lista elemein kívül is van még néhány hasznos bonyolultság, de ezek elegendőek lesznek a könyvben bemutatott algoritmusok bonyolultságának leírására.

Az 1.1. ábrán látjuk, hogy a különböző bonyolultság-nagyságrendek hogyan viszonyulnak egymáshoz. A vízszintes tengely a probléma méretét jelenti – például a keresési algoritmussal feldolgozandó rekordok számát. A függőleges tengely az egyes osztályok algoritmusainak számításigényét jelenti. Az ábra nem jelzi a futásidőt vagy a szükséges processzorciklusokat; pusztán annyit mutat, hogy a számítógépes erőforrásigény a megoldani kívánt probléma méretével együtt nő.




**1.1. ábra.** *A bonyolultság különböző nagyságrendjeinek összehasonlítása*

Az előző listában talán feltűnt, hogy a nagyságrendek egyike sem tartalmaz konstanst. Azaz, ha az algoritmus várt futásidejű teljesítménye az `N`, `2×N`, `3×N` vagy akár `100×N` értékekkel arányban áll, a bonyolultság minden esetben `O(N)`. Első pillantásra kicsit furcsának tűnhet – természetes, hogy a `2×N` jobb, mint a `100×N` – , de mint már korábban említettük, nem az a célunk, hogy megállapítsuk a műveletek pontos számát, hanem hogy a különböző algoritmusok relatív hatékonyságát összehasonlítsuk. Más szóval az `O(N)` idő alatt befejezett algoritmus túlszárnyal egy másik, `O(N²)` ideig futó algoritmust. Továbbá, ha `N` nagy értékeivel akad dolgunk, a konstansok nem sokat változtatnak a helyzeten: a teljes méret arányát tekintve az 1 000 000 000 és a 20 000 000 000 közötti különbség majdnem elhanyagolható, még akkor is, ha az egyik a másiknak a hússzorosa.

Természetesen szeretnénk összehasonlítani a különböző algoritmusok tényleges teljesítményét, különösen akkor, ha az egyik 20 perc alatt befejeződik, míg a másik csak 3 óra alatt, és mindkét algoritmus nagyságrendje `O(N)`. Azt kell megjegyezünk, hogy sokkal könnyebb megfelezni egy `O(N)` bonyolultságú algoritmus idejét, mint módosítani egy olyan algoritmust, amely az `O(N)` nagyságrendhez képest `O(N²)` nagyságrenddel bír.

## Konstans idő: O(1)

Megbocsátható az a feltételezés, hogy az `O(1)` bonyolultság azt jelenti, hogy az algoritmus egyetlen művelet segítségével végrehajtja a funkciót. Noha ez valóban lehetséges, az `O(1)` tulajdonképpen valójában annyit jelent, hogy az algoritmus konstans ideig fut; vagyis a teljesítményt nem befolyásolja a probléma mérete. Valószínűleg nem tévedünk, ha úgy véljük, ez túl szép ahhoz, hogy igaz legyen.

Az egyszerű funkciók futása garantáltan `O(1)` ideig tart. A konstans időbeli teljesítmény legegyszerűbb példája a számítógép operatív memóriáját címezi, és kiterjesztésként tömbbeli keresést hajt végre. A tömb egy elemének keresése a mérettől függetlenül általában ugyanannyi ideig tart.

Bonyolultabb problémák esetén azonban nagyon nehéz konstans ideig futó algoritmust találni: a „Listák" című fejezet (3.) és a „Hasítás" című fejezet (11.) bevezeti az `O(1)` időbeli bonyolultsággal rendelkező adatstruktúrákat és algoritmusokat.

A konstans időbeli bonyolultsággal kapcsolatban még azt kell megjegyeznünk, hogy ez még mindig nem garantálja az algoritmus gyorsaságát, csak azt, hogy a végrehajtásához szükséges idő mindig ugyanannyi lesz: az algoritmus, amely egy hónapig fut, még mindig `O(1)` algoritmus, még akkor is, ha ez a futásidő teljességgel elfogadhatatlan.

## Lineáris idő: O(N)

Az algoritmus akkor fut `O(N)` nagyságrenddel, ha a funkció végrehajtásához szükséges műveletek száma egyenesen arányos a feldolgozni kívánt elemek számával. Az 1.1. ábrára pillantva láthatjuk, hogy az `O(N)` vonala felfelé folytatódik, a meredeksége változatlan.

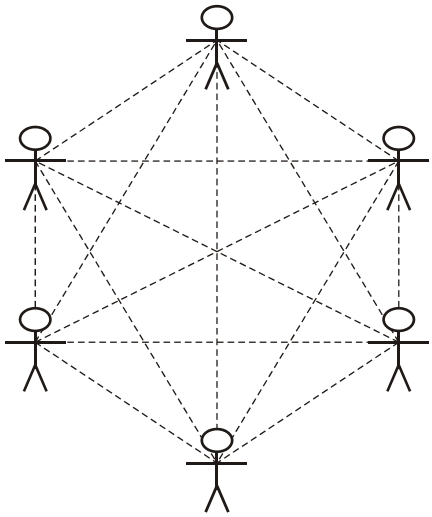
Ilyen algoritmus például az áruházi pénztárnál való várakozás. A vásárlókat átlagosan ugyanannyi idő alatt lehet kiszolgálni: ha egy vásárló kosarát két perc alatt fel lehet dolgozni, körülbelül `2×10 = 20` perc kell tíz vásárló kiszolgálásához, és `2×40 = 80` perc 40 vásárlóhoz. A lényeg, hogy nem fontos, hány vásárló áll a sorban, az egyes vásárlók kiszolgálásához szükséges idő nagyjából ugyanannyi marad. Elmondhatjuk, hogy a kiszolgálás ideje egyenesen arányos a vásárlók számával, tehát az idő `O(N)`.

Érdekes módon, ha bármikor megkétszerezzük vagy akár megháromszorozzuk a műveletben a regiszterek számát, a feldolgozási idő továbbra is `O(N)` marad. Ne feledjük, hogy a nagy O jelölés mindig minden konstanst figyelmen kívül hagy.

Az `O(N)` futási idejű algoritmusok rendszerint elfogadhatóak. Legalább olyan hatékonynak tekinthetők, mint az `O(1)` futásidejű algoritmusok, de ahogy már említettük, igen nehéz konstans idejű algoritmust találni. Ha sikerül lineáris idővel futó algoritmust találnunk, mint azt a „Sztringkeresés" című fejezetben (16.) látni fogjuk, kis elemzéssel – és zseniális ötletekkel – még hatékonyabbá tehetjük.

## Kvadratikus idő: O($N^2$)

Képzeljünk el egy csoportot, amelynek tagjai most találkoznak egymással először, és az illemszabályoknak megfelelően mindannyian kézfogással üdvözlik a csoport összes többi tagját. Ha a csoportban hatan vannak, akkor ez az 1.2. ábrán látható módon összesen `5+4+3+2+1 = 15` kézfogást jelent.



**1.2. ábra.** *A csoport minden tagja üdvözli a csoport összes többi tagját*

Mi történne, ha a csoport hét főből állna? Az üdvözlés összesen `6+5+4+3+2+1 = 21` kézfogásba kerülne. És ha nyolcból? Ez `7+6+...+2+1 = 28` kézfogást jelentene. És, ha kilencen lennének a csoportban? Már nagyjából láthatjuk a lényeget: Ahányszor a csoport mérete egy fővel növekszik, egy további embernek kell kezet ráznia az összes többivel.

Az `N` méretű csoportban a kézfogások száma `(N²–N)/2` lesz. Mivel a nagy O minden konstanst figyelmen kívül hagy – ebben az esetben a `2`-t –, a kézfogások száma `N²–N`. Mint azt az 1.1. táblázatban látjuk, ahogy `N` egyre nő, `N` kivonása `N²`-ből a végeredményre egyre elhanyagolhatóbb hatással van, tehát bátran elhagyhatjuk a kivonást, ezáltal az algoritmus bonyolultsága `O(N²)` lesz.

| N | $N^2$ | $N^2$—N | Különbség |
|---|---|---|---|
| 1 | 1 | 0 | 100,00% |
| 10 | 100 | 90 | 10,00% |
| 100 | 10 000 | 9 900 | 1,00% |

| N | $N^2$ | $N^2$—N | Különbség |
|---|---|---|---|
| 1 000 | 1 000 000 | 999 000 | 0,10% |
| 10 000 | 100 000 000 | 99 990 000 | 0,01% |

**1.1. táblázat.** *Az N kivonása $N^2$-ből az N növekedése mellett*

A kvadratikus idővel futó algoritmusok a programozók legvadabb rémálmaiban jelennek meg; bármely `O(N2)` bonyolultságú algoritmus kizárólag a legjelentéktelenebb problémák megoldására alkalmas. A keresést tárgyaló 6. és 7. fejezetben további érdekes példákat találunk.

## Logaritmikus idő: O(log N) és O(N log N)

Az 1.1. ábrán látható, hogy az `O(log N)` jobb, mint az `O(N)`, de nem olyan jó, mint az `O(1)`.

A logaritmikus algoritmus futásideje a probléma méretének – rendszerint 2-es alapú – logaritmusával együtt növekszik. Ez annyit jelent, hogy ha a beviteli adathalmaz mérete milliós szorzóval növekszik, a futásidő csak `log(1000000) = 20` szorzóval növekszik. Az egész számok 2-es alapú logaritmusát egyszerűen kiszámíthatjuk, ha megkeressük, hogy a szám tárolásához hány bináris számjegy szükséges. Például, a 300 2-es alapú logaritmusa 9, mivel a decimális 300 megjelenítéséhez 9 bináris számjegy szükséges (a bináris ábrázolás 100101100).

A logaritmikus futásidők megvalósításához az algoritmusnak a beviteli adathalmaz nagy részét rendszerint figyelmen kívül kell hagynia. Ennek eredményeként a legtöbb algoritmus – amely így viselkedik – valamilyen keresést foglal magában. A „Bináris keresés és beszúrás" című fejezet (9.), és a „Bináris keresőfák" című fejezet (10.) `O(log N)` algoritmusokat mutat be.

Ha ismét megtekintjük az 1.1. ábrát, láthatjuk, hogy az `O(N log N)` jobb, mint az `O(N²)`, de nem olyan jó, mint az `O(N)`. A 6. és a 7. fejezetben `O(N log N)` algoritmusokkal találkozhatunk.

## Faktoriális idő: O(N!)

Lehet, hogy nem gondolnánk, de néhány algoritmus még az `O(N²)`-nél is rosszabbul teljesít – az 1.1. ábrán hasonlítsuk össze az `O(N²)` és `O(N!)` bonyolultságokat. (Valójában vannak ennél sokkal rosszabb algoritmusok is, de ezeket a könyvben nem tárgyaljuk.)

Elég ritkán találkozhatunk ilyen funkciókkal, különösen, ha olyan példákat keresünk, amelyek nem a kódolásról szólnak. Ha tehát elfelejtettük volna, hogy mi a faktoriális – vagy még nem is találkoztunk vele soha –, íme egy gyors ismétlés:

A faktoriális az egész szám és az azt megelőző természetes számok szorzata.

Például a `6!` („6 faktoriális”) `= 6×5×4×3×2×1 = 720` és a `10! = 10×9×8×7×6×5×4×3×2×1= 3628800`.

Az 1.2. táblázat az $N^2$ és az `N!` összehasonlítását tartalmazza 1 és 10 közötti egész számokra.

| N | $N^2$ | N! |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 4 | 2 |
| 3 | 9 | 6 |
| 4 | 16 | 24 |
| 5 | 25 | 120 |
| 6 | 36 | 720 |
| 7 | 49 | 5 040 |
| 8 | 64 | 40 320 |
| 9 | 81 | 362 880 |
| 10 | 100 | 3 628 800 |

**1.2. táblázat.** *Az $N^2$ és az N! összehasonlítása kis egész számok esetén*

Mint látjuk, ha az `N` értéke `N=2` vagy annál kisebb, a faktoriális bonyolultság jobb, mint a kvadratikus, de ezen a ponton a faktoriális bonyolultság nekilendül, és katasztrófával fenyeget. Következésképpen az `O(N²)` bonyolultsághoz képest még inkább reménykednünk kell abban, hogy algoritmusunk bonyolultsága ne `O(N!)` legyen.

# Egységtesztelés

Mielőtt folytatnánk utazásunkat az algoritmusok világában, kicsit elkalandozva beszélnünk kell egy szívünknek igen kedves témáról: az egységtesztelésről. Az elmúlt években az egységtesztelés nagyon népszerűvé vált azoknak a fejlesztőknek a körében, akiknek fontos az általuk készített rendszerek minősége. Közülük sokan kellemetlenül érzik magukat, ha a szoftver készítése közben nem építenek egy automatikus teszteket

magában foglaló csomagot is, amely bizonyítja, hogy az elkészült szoftver az elvárásoknak megfelelően működik. Mi is ezt a hozzáállást képviseljük. Ezért minden ismertetett algoritmus esetén bemutatjuk, hogyan működik az adott algoritmus, és egységtesztek révén azt is, hogy mit csinál. Mindenkinek ajánljuk, hogy fejlesztési munkája során váljon ez a szokásává, mivel nagyban segíti a túlórázás elkerülését.

A következő néhány részben gyors áttekintést kapunk az egységtesztelésről, és betekintést nyerünk a JUnit-keretrendszerbe Java-program egységteszteléséhez. A könyvben a JUnit-rendszert alkalmazzuk, tehát érdemes megismerkednünk ezzel a programmal, hogy könnyebben megértsük a könyv példáit. Ha már kemény, teszteléssel fertőzött fejlesztőnek érezzük magunkat, a könyv e részét nyugodtan átlapozhatjuk. Örüljünk neki!

## Mi az egységtesztelés?

Az egységteszt olyan program, amely egy másikat tesztel. Java-környezetben ez egy Java-osztályt jelent, amelynek a célja a többi osztály tesztelése. Nagyjából ennyi. Mint az életben a legtöbb dolog, ez is könnyen megtanulható, de sokat kell gyakorolni. Az egységtesztelés művészet és egyben tudomány is; rengeteg irodalmat találunk a tesztelésről, tehát itt nem merülünk el a tesztelés részleteiben. Az A függelék több könyvet is ajánl a témával kapcsolatban.

Az egységteszt alapvető működése az alábbiakat foglalja magában.

1. A teszt támogatásához szükséges objektumok, mint a példaadatok előkészítése. Ezek az úgynevezett *tartozékok*.
2. Az objektumok használatával futtassuk a tesztet, és győződjünk meg róla, hogy valóban az történt, amire számítottunk. Ezek a folyamatok a *vizsgálatok*.
3. Végül szabaduljunk meg minden felesleges dologtól. Ez a *lebontás* folyamata.

A könyvben az egységtesztek elnevezésénél minden esetben úgy járunk el, hogy az osztálynévvel létrehozott tesztosztály nevéhez hozzáfűzzük a `Teszt` szót. Ha például a `Mütyür` névre hallgató osztályt készülünk tesztelni, az osztály egységtesztjeként létrehozott osztály neve `MütyürTeszt` lesz. Erre rengeteg példát találunk majd a könyvben. A forrásfájlok elrendezése is egységes szabály szerint történik. Az egységteszteket a fő forrásfájlok csomagstruktúrájával megegyező párhuzamos forrásfán helyezzük el. Ha például a Mütyür a `com.wrox.algorithms` osztályon belül létezik, a forrásfájlok elrendezése az 1.3. ábrán látható elrendezéshez lesz hasonlatos.

```
src
  | - main
  |      | - com
  |            | - wrox
  |                   | - algorithms
  |                          | - Mütyür
  |
  |
  -- test
       | - com
            | - wrox
                  | - algorithms
                         | - MütyürTeszt
```

**1.3. ábra.** *Az egységteszt forrásfájljai párhuzamos csomagstruktúrába rendeződnek*

Ez annyit jelent, hogy a fájlok tetején a Java-csomag utasítás pontosan ugyanaz, de maguk a fájlok a fájlrendszer különböző könyvtáraiban helyezkednek el. A modell a könnyebb csomagolhatóság és a fő kód terjesztésének érdekében az éles kódot elkülöníti a tesztelés kódjától, és így biztosítja, hogy az éles kód az elkészítése folyamán nem a tesztkódon alapul, mivel a két könyvtár fordítása közben az osztályútvonal kissé eltérő. Tetszetős az a tény is, hogy lehetővé teszi a tesztek számára a csomagelvű metódusok hozzáférését, ezt is érdemes megfontolni.

Az egységtesztek részletezésének befejezése előtt még meg kell jegyeznünk, hogy a tesztelés más közös típusaival is találkozhatunk majd munkánk során. A továbbiakban megismerkedünk néhány definícióval, amelyek segítségével megfelelő háttérinformációkhoz jutunk, és elkerülhetjük a szükségtelen félreértéseket. A könyv csak alkalmazza az egységtesztelést, tehát a tesztelés további típusaival kapcsolatban érdemes megtekinteni a referenciaanyagokat. A szakirodalomban az alábbi szakkifejezésekkel találkozhatunk:

- **Fekete dobozos tesztelés:** képzeljük el, hogy DVD-lejátszónkat teszteljük. Csak az előlap gombjaihoz és a hátlap csatlakozóihoz férünk hozzá (hacsak nem akarunk búcsút inteni a garanciának). A DVD-lejátszó összetevőit nem tudjuk tesztelni, mivel nem férünk hozzájuk. Kizárólag a fekete doboz külső oldalán látható vezérlőelemekre hagyatkozhatunk. Számítástechnikai szempontból ez a telepített alkalmazás felhasználói felületének hozzáféréséhez hasonlít. Rengeteg összetevővel állunk szemben, de azokhoz nem férünk hozzá.
- **Funkcionális tesztelés:** a kifejezést a fekete dobozos tesztelés szinonimájaként használhatjuk.
- **Fehér dobozos tesztelés:** olyan tesztelésre utal, amely kisebb vagy nagyobbmértékben hozzáférhet a rendszer átívelő komponensszervezéséhez, és képes egyedi komponenseket is tesztelni, rendszerint a felhasználói felület igénybevétele nélkül.

- **Integrációs teszt:** nagy, elosztott rendszer külön összetevőjének tesztelését jelenti. Az ilyen tesztek célja, hogy egymástól független fejlesztésük folyamán biztosítsák a rendszerek előzetes megállapodásoknak megfelelő működését.

Az egységtesztelés a tesztelési módszerek közül a legaprólékosabb, mivel az ilyen tesztek során bármely más osztálytól függetlenül egyetlen osztályt tesztelünk. Ez annyit jelent, hogy az egységteszteket gyorsan futtathatjuk, és egymástól viszonylag függetlenek.

## Miért fontos az egységtesztelés?

Ha nem értjük, hogy miért olvashatunk egy alapvetően algoritmusokról szóló könyvben ilyen sokat az egységtesztelésről, gondoljunk csak a Java-fordítóra, amely lehetővé teszi Java programjaink futtatását. Működőképesnek vélhetünk egy megírt kódot anélkül, hogy lefordítanánk? Valószínűleg nem! Gondoljunk úgy a fordítóra, mint a program egyfajta tesztelésére – biztosítja, hogy a program helyes szintaxissal készült. Nagyjából ennyi. Nem ad visszacsatolást arról, hogy a program praktikus vagy hasznos feladatot hajt végre: ez az a pont, ahol az egységtesztek bekerülnek a képbe. Ha jobban érdekel bennünket az, hogy programjaink valóban valami hasznos dolgot hajtanak-e végre, mint az, hogy helyesen írtuk-e be a Java-kódot, akkor az egységtesztek segítségével komoly gátat emelhetünk mindenféle programhiba elé.

Az egységtesztek másik előnye, hogy megbízható dokumentációt biztosítanak a tesztelés alatt álló osztály viselkedéséről. Ha majd látunk néhány egységtesztet működés közben, észre fogjuk venni, hogy a tesztek vizsgálatával mennyivel könnyebb rájönni, mit csinál az osztály, mint ha magát a kódot nézegetnénk. (A kódhoz akkor kell fordulni, ha azt szeretnénk kideríteni, hogy a program *hogyan* csinálja azt, amit csinál, de ez már teljesen más téma.)

## JUnit-bevezető

Az első hely, ahová el kell látogatnunk, a JUnit-webhelye a `www.junit.org/` címen. Itt nemcsak a letölthető szoftvert találjuk meg, hanem a JUnit integrált fejlesztői környezetünkben való használatával kapcsolatos információkat is, valamint a JUnit kibővítéseit és fejlesztéseit, amelyek a speciális igények kielégítésére készültek.

A szoftver letöltése után csak a `junit.jar` fájlt kell hozzáadnunk a classpath környezeti változóhoz, és már készen is állunk első egységtesztünk létrehozására. Az egységteszt létrehozásához készítsünk egy Java-osztályt, amely kibővíti a `junit.framework.TestCase` alaposztályt. Az alábbi kód a JUnit segítségével készített egységteszt alapvető felépítését mutatja:

```
package com.wrox.algorithms.queues;

import com.wrox.algorithms.lists.LinkedList;
import com.wrox.algorithms.lists.List;
import junit.framework.TestCase;

public class RandomListQueueTest extends TestCase {
  private static final String VALUE_A = "A";
  private static final String VALUE_B = "B";
  private static final String VALUE_C = "C";

  private Queue _queue;
  ...
}
```

Ne foglalkozzunk azzal, valójában mit tesztel ez az egységteszt; erre a könyv későbbi részében a várakozási sorok ismertetése során még visszatérünk, és ráérünk akkor megérteni. A lényeg, hogy az egységteszt szokványos osztály, amely a JUnit-keretrendszer által biztosított alaposztállyal rendelkezik. A kód meghatározza az osztályt, kibővíti az alaposztályt, majd deklarál néhány statikus tagot és egy példány tagot, amely a tesztelni kívánt várakozási sort tárolja.

A következő lépés a `setUp` metódus felülbírálása és az objektumok teszteléséhez szükséges kód hozzáadása. Ebben az esetben ez annyit jelent, hogy a felülbírált `setUp` metódust a szuperosztályban kell meghívni, és a várakozásisor-objektumot teszteléshez példányosítani:

> *Figyeljük meg a* `setUp` *metódus írásmódját! Vegyük észre középen a nagy U betűt! A Java egyik gyenge pontja, hogy a metódusokat pusztán egybeeséssel bírálhatjuk felül, nem elegendő a szándék. Ha elgépeljük a metódus nevét, a kód nem az elvárásoknak megfelelően működik majd.*

```
protected void setUp() throws Exception {
  super.setUp();

  _queue = new RandomListQueue();
}
```

A JUnit-keretrendszer által biztosított dolgok egyike a garancia, hogy a tesztmetódus futtatása során (ezt is mindjárt látni fogjuk), a rendszer minden egyes tesztfuttatás előtt meghívja a `setUp` metódust. Hasonlóképpen minden tesztmetódus futtatása után a `tearDown` metódus biztosítja számunkra a lehetőséget, hogy a következő példában bemutatott módon kitakarítsunk magunk után:

```
protected void tearDown() throws Exception {
  super.tearDown();

  _queue = null;
}
```

Meglepődhetünk azon, hogy miért kell a példánytag mezőt `null` értékre állítani. Noha nem feltétlenül szükséges, az egységtesztek túlnyomó többségében, ha ezt a lépést figyelmen kívül hagyjuk, az egységtesztek a szükségesnél jóval több memóriát foglalnak; tehát érdemes hozzászokni a lépés végrehajtásához.

A tényleges egységteszt kódjának következő metódusa a várakozási sor viselkedését teszteli, ha a sor üres, és valaki megpróbál egy elemet kivenni belőle; ezt az objektum tervezője nem tette lehetővé. Ez egy roppant érdekes eset, mivel bemutatja azt a technikát, amelynek a segítségével bebizonyíthatjuk, hogy a helytelen használat során osztályaink az elvárt módon kudarcot vallanak. Íme a kód:

```
public void testAccessAnEmptyQueue() {
  assertEquals(0, _queue.size());
  assertTrue(_queue.isEmpty());

  try {
    _queue.dequeue();
    fail();
  } catch (EmptyQueueException e) {
    // ezt várjuk
  }
}
```

A kóddal kapcsolatban vegyük észre a következőket:

- A metódus neve `test`-tel kezdődik. Ez a JUnit keretrendszer követelménye, amelynek a segítségével a tesztmetódust meg tudja különböztetni a támogató metódustól.
- A metódus első sora az `assertequals()` metódust alkalmazza annak ellenőrzésére, hogy a várakozási sor hossza valóban nulla. A metódus szintaxisa `assertEquals(ezt várjuk, tényleges)`. A metódusnak túlterhelt verziói is léteznek az összes Java-alaptípus számára, tehát a metódussal a könyvben részletesen megismerkedhetünk. Az egységtesztek világában valószínűleg ez a legáltalánosabb érvényességvizsgálat: annak ellenőrzése, hogy egy adott objektum az elvárt értékkel rendelkezik. Ha valamilyen okból kifolyólag az elvárttól eltérő értéket talál, a JUnit-keretrendszer megszakítja a teszt végrehajtását, és hibát jelez. Ezáltal az egységteszteket tömörré és olvashatóvá tehetjük.

- A második sor egy másik nagyon általános érvényességvizsgálatot alkalmaz, az `asserttrue()`metódust, amellyel ellenőrizhető, hogy a tesztfuttatás során a logikai értékek a várt állapotban vannak. Ebben az esetben arról győződhetünk meg, hogy a várakozási sor helyesen az üres állapotot jelzi.
- A `try/catch` blokk veszi körül a várakozásisor-objektumon a metódushívást, amely kivételt jelez, ha a várakozási sor üres. Ez a felépítés csak nagyon kevéssé tér el a Java normális kivételkezelésétől, tehát vizsgáljuk meg jó alaposan. Jelen esetben az a *jó*, hogy kód kivételt jelez, és az a *rossz*, ha nem. Ebből kifolyólag a kód a `catch` blokkban semmit sem csinál, hanem a `try` blokkban közvetlenül a tesztelni kívánt metódus meghívása után meghívja a JUnit-keretrendszer `fail()` metódusát. Ha a `fail()` metódus megszakítja a tesztet, és hibát jelez, azaz a metódus várt kivételt jelez, a végrehajtás keresztülhalad a metódus végén, és a teszt sikeres lesz. Ha a rendszer nem jelez kivételt, a teszt azonnal megbukik. Ha ez egy kissé zavarosnak tűnik, olvassuk át újra a példát!

Íme egy újabb egységteszt-példametódus ugyanabban az osztályban.

```
public void testClear() {
  _queue.enqueue(VALUE_A);
  _queue.enqueue(VALUE_B);
  _queue.enqueue(VALUE_C);

  assertEquals(3, _queue.size());
  assertFalse(_queue.isEmpty());

  _queue.clear();

  assertEquals(0, _queue.size());
  assertTrue(_queue.isEmpty());
}
```

A metódus neve ismét `test`-tel kezdődik, hogy a JUnit-reflexió segítségével megtalálhassa. A teszt néhány új elemet ad a várakozási sorhoz, megvizsgálja, hogy a `size()` és az `isEmpty` metódusok megfelelően működnek-e, majd kiüríti a várakozási sort, és ismét ellenőrzi a metódusok működését.

Az egységteszt megírása után a következő lépés a teszt futtatása. Vegyük észre, hogy egységtesztünk nem tartalmaz `main` metódust, tehát közvetlenül nem tudjuk futtatni. A JUnit több tesztfuttató környezetet is biztosít különböző interfészekkel – egészen az egyszerű szövegalapú konzolfelülettől kezdve a gazdag grafikus felületig.

A legtöbb Java-fejlesztői környezet, mint például az Eclipse vagy az IntelliJ IDEA közvetlen támogatást biztosít a JUnit-tesztek futtatásához, de ha csak parancssor áll rendelkezésünkre, az előző tesztet az alábbi parancs segítségével is futtathatjuk (a classpath környezeti változónak természetesen tartalmaznia kell a `junit.jar` fájlt):

```
java junit.textui.TestRunner com.wrox.algorithms.queues.Random-
ListQueueTest
```

A grafikus változat futtatása ugyanilyen egyszerű:

```
java junit.swingui.TestRunner com.wrox.algorithms.queues.Random-
ListQueueTest
```

A JUnit rengeteg, a szoftver építésére alkalmazott eszközben – például Antben vagy Mavenben – használható. Fejlesztői életünket nagyban megkönnyíthetjük a szoftver minden egyes változatának jó egységtesztcsomagban való futtatásával, és a szoftver sokkal robusztusabb is lesz, tehát további részletekért érdemes ellátogatni a JUnit webhelyére.

## Tesztelésen alapuló programozás

A könyv összes algoritmusa és adatstruktúrája egységteszteket is tartalmaz, amelyek biztosítják, hogy a kód az elvárásoknak megfelelően működjön. Az egységtesztek valójában még a tesztelni kívánt kód megírása *előtt* megszülettek! Ez kissé furcsának tűnhet, de ha egységtesztekkel lesz dolgunk, tudnunk kell arról az egyre nagyobb népszerűségnek örvendő technikáról, amelyet azok a fejlesztők alkalmaznak, akiknek fontos a megírt kód minősége. Ez a technika a *tesztelésen alapuló programozás*.

A *tesztelésen alapuló programozás* fogalma Kent Becktől, az eXtreme Programming atyjától származik, aki több könyvet is írt az eXtreme Programming témakörében, valamint a tesztelésen alapuló programozásról. Az alapötlet szerint a fejlesztési próbálkozások felvesznek egy bizonyos ritmust, amely a tesztkód írása, az éles kód írása és a kód a célnak való megfelelés érdekében végrehajtott tisztogatása (átszervezése) között váltakozik. A ritmus a szoftver készítése során a folyamatos előrehaladás érzését kelti, miközben olyan szilárd egységtesztcsomagot építünk, amely megvéd a kód változtatásai által okozott programhibáktól.

Ha a könyv olvasása közben úgy határozunk, hogy az egységtesztelést szeretnénk saját kódjainkba is beépíteni, rengeteg könyv áll rendelkezésünkre ebben a témakörben. Ajánlásaink az A függelékben találhatók.

# Összefoglalás

A fejezetben a következő témakörökkel foglalkoztunk.

- Az algoritmusok mindennapi életünkben is jelen vannak.
- Az algoritmusok a legtöbb számítógéprendszer központi részét alkotják.
- Mit jelent az algoritmusbonyolultság?
- Az algoritmusokat bonyolultságuk tekintetében összehasonlíthatjuk.
- A nagy O jelölést széles körben alkalmazhatjuk az algoritmusok bonyolultság alapján való osztályozására.
- Mi az egységtesztelés, és miért fontos?
- Hogyan lehet egységteszteket írni a JUnit segítségével?