2.

Rendezés

...egész nap a barlangban szorgoskodtam, ahol a vert pénzt kellett kétszersültes zsákokba töltenem... azt hiszem, sohasem szórakoztam jobban, mint amikor ezeket rendezgettem. Angol, francia, spanyol, portugál aranyak, György és Lajos tallérok, dublónok, kettős guineák, monédorok és zecchinók. ROBERT LOUIS STEVENSON
(Jim emlékei a Kincses Szigetről)

Ebben a fejezetben az egyik klasszikusnak számító, alapvető számítási feladattal, a rendezéssel foglalkozunk. Ennek értelmezéséhez nézzük először a rendezési reláció fogalmát. Legyen U egy halmaz, és $<$ egy kétváltozós reláció U-n. Ha $a, b \in U$ és $a < b$, akkor azt mondjuk, hogy a kisebb, mint b. A $<$ reláció egy *rendezés*, ha teljesülnek a következők:

1. $a \not< a$ minden $a \in U$ elemre ($<$ irreflexív);
2. Ha $a, b, c \in U$, $a < b$, és $b < c$, akkor $a < c$ ($<$ tranzitív);
3. Tetszőleges $a \neq b \in U$ elemekre vagy $a < b$, vagy $b < a$ fennáll ($<$ teljes).

Ha $<$ egy rendezés U-n, akkor az $(U, <)$ párt *rendezett halmaznak* nevezzük. Ha az U halmaz egy számítógépes adattípus is egyben, akkor az $(U, <)$ párt *rendezett típusnak* is nevezik. Különösen ez utóbbiak érdekesek a számunkra. Az U elemeit ekkor adatoknak tekinthetjük, és feltehetjük, hogy a $<$ reláció hatékonyan a rendelkezésünkre áll: ha adottak az $a, b \in U$ elemek, akkor az $a < b$ viszony ellenőrzése, más szóval a és b *összehasonlítása* hatékonyan megtehető. Ezt az összehasonlító eljárást gyakran a rendszer biztosítja, olykor viszont magunknak kell megírnunk.

Példák:

1. $\mathbb{Z}$ az egész számok halmaza. A $<$ rendezés a nagyság szerinti rendezés. $(\mathbb{Z}, <)$ tekinthető rendezett típusnak is.

2. Az *abc* betűinek Σ halmaza; a $<$ rendezést az *abc-sorrend* adja. Az x betű kisebb, mint az y betű, ha x előbb szerepel az *abc*-sorrendben, mint y.
3. A Σ betűiből alkotott szavak Σ^* halmaza a szótárszerű vagy *lexikografikus* rendezéssel. Ezt a következő recepttel értelmezhetjük: legyen $X = x_1x_2\cdots x_k$ és $Y = y_1y_2\cdots y_l$ ($x_i, y_j \in \Sigma$) két szó (X az x_i, Y pedig az y_j betűk sorozata). Az X kisebb mint Y, ha vagy $l > k$ és $x_i = y_i$ ha $i = 1, 2, \ldots, k$; vagy pedig $x_j < y_j$ teljesül a legkisebb olyan j indexre, melyre $x_j \neq y_j$. Tehát például $kar < karika$ és $bor < bot$.

A rendezés feladata általánosan így fogalmazható: *adott az* $(U, <)$ *rendezett halmaz elemeinek egy* $u_1, u_2, \ldots, u_n$ *sorozata; rendezzük ezt át egy nem csökkenő* $v_1, v_2, \ldots, v_n$ *sorrendbe.*

A nem csökkenő sorrend azt jelenti, hogy $v_1 \leq v_2 \leq \ldots \leq v_n$ teljesül, ahol $\leq$ a kisebb vagy egyenlő reláció jelölése. Az egyenlőséget akkor kell megengednünk, ha az u_i sorozatban ismétlődések vannak, ami az alkalmazásoknál gyakran előfordul.

Az $u_1, u_2, \ldots, u_n$ sorozat többféle módon adható meg. Kaphatjuk tömb, láncolt lista formájában vagy egy külső táron levő állomány rekordjainak sorozataként. Legtöbbször a $v_1, v_2, \ldots, v_n$ sorozatot ugyanolyan formában követelik tőlünk, mint amilyen a bemenet volt. Ezért a rendező módszerek átalakító lépései többnyire az elemek (tömbelem, listaelem, rekord) *mozgatásai, cseréi.*

A rendezési feladat két szempontból fontos. Gyakran előfordul, hogy magában a kérdés megfogalmazásában szerepel a rendezés. Például, ha arra vagyunk kíváncsiak, hogy egy nyilvántartásban szereplő személyek közül kinek a legnagyobb a fizetése, vagy hogy kiknek a fizetése van az átlag fölött stb. A másik, talán még fontosabb alkalmazás a rendezés használata a keresés megkönnyítésére. Ezzel később behatóan foglalkozunk. Most csak egy példát említünk. A telefonkönyvhöz általában azért fordulunk, hogy (gyorsan) megtaláljunk egy telefonszámot. Nem érdekel bennünket különösebben a nevek rendezése, de a név szerinti rendezettség komoly segítséget nyújt a gyors keresésben (képzeljük el, mi lenne, ha a telefonkönyv bejegyzéseit csak úgy, össze-vissza vetnék papírra).

A fejezet elején a rendezett halmazokban való kereséssel foglalkozunk. Utána áttekintjük a legfontosabb összehasonlítás alapú rendezéseket. Az ilyen módszerek a döntéseiket kizárólag az elemek közötti $<$ relációra alapozzák. A kulcsmanipulációs rendezések ezzel szemben használhatják az elemek (kulcsok) belső szerkezetének ismeretéből eredő előnyöket. Megismerünk két ilyen szemléletű eljárást is. Ezután bemutatunk egy gyors párhuzamos rendező módszert, végül pedig külső tárakon levő állományok rendezésével foglalkozunk.

2.1. Keresés rendezett halmazban

A rendezett halmazokban való keresési eljárások jelentős szerepet játszanak a legkülönfélébb adatkezelő rendszerekben. Érdemes tehát közelebbről is megismerkedni velük. Tegyük fel, hogy adott az $(U, <)$ rendezett halmaz véges

$$S = \{s_1 < s_2 < \ldots < s_{n-1} < s_n\}$$

részhalmaza. Az S-re úgy gondolhatunk, mint az U általunk tárolt elemeinek összességére. Adott ezenkívül a bemenet részeként egy $s \in U$ elem. A feladatunk annak eldöntése, hogy $s \in S$ teljesül-e. Igenlő válasz esetén általában az s elem S-beli „helye" is érdekel bennünket.

A feladatot összehasonlítások segítségével szeretnénk megoldani. Egy ilyen lépésben vesszük valamelyik s_i-t (S-nek a rendezés szerinti i-edik elemét), és összehasonlítjuk az s elemmel. Az eredmény lehet $s = s_i$, $s_i > s$ vagy $s > s_i$. Az eredménytől függően további ilyen összehasonlításokat végzünk, egészen addig, amíg az ($s \in S$?) kérdést meg nem tudjuk válaszolni. A fontosabb módszerek a következők.

Lineáris keresés

Az s-et először az S halmaz legkisebb elemével, s_1-gyel hasonlítjuk össze. Ha $s < s_1$, akkor végeztünk, megállapítva, hogy s nincs S-ben. Ha $s = s_1$, akkor sikerrel jártunk: az ($s \in S$?) kérdésre a válasz igenlő. Ha pedig $s > s_1$, akkor továbbmegyünk, és az s_2 elemet vetjük össze s-sel. Ennek a kimeneteleit az előzőhöz hasonlóan kezeljük. Vagy választ kapunk a keresőkérdésre, vagy folytatjuk s_3-mal. És így tovább. Addig lépkedünk előre az $s_1, s_2, \ldots, s_i, \ldots$ sorozat mentén, amíg az ($s \in S$?) kérdés el nem dől.

Mi mondható a módszer – összehasonlításokban mért – költségéről? A legkedvezőtlenebb esetben, amikor is $s \geq s_n$, az S halmaz minden elemével hasonlítunk; ez $|S| = n$ összehasonlítást jelent. Ennél a feladatnál szokás *átlagos lépésszámról* is beszélni. Ekkor azzal a feltevéssel élünk, hogy a keresett s elem egyenlő eséllyel lehet a $(-\infty, s_1]$, $(s_1, s_2], \ldots, (s_{n-1}, s_n]$ és (s_n, ∞) intervallumok bármelyikében. Itt $(a, b]$ az U azon s elemeinek az összességét jelenti, melyekre $a < s \leq b$ igaz. Hasonlóan, $(-\infty, b]$ az U halmaz b-nél nem nagyobb, (a, ∞) pedig az a-nál nagyobb elemeinek halmaza.

Ha $s \in (-\infty, s_1]$, akkor 1 összehasonlítást végzünk, ha pedig $s \in (s_i, s_{i+1}]$ akkor $i + 1$-et ($1 \leq i < n$). Az utolsó intervallum $(s_n, \infty]$ elemeire a költség n összehasonlítás. A kapott $n + 1$ szám átlaga

$$\frac{1 + 2 + \cdots + n - 1 + n + n}{n + 1} = \frac{n(n+1)}{2(n+1)} + \frac{n}{n+1} = \frac{n}{2} + \frac{n}{n+1}.$$

A módszer átlagos költsége tehát $(n/2) + 1$ összehasonlítás körül van.

Bináris keresés

Az eljárás az *oszd meg és uralkodj* elven alapul. A feladat megoldását sokkal kisebb hasonló feladatok megoldására egyszerűsítjük. Először az S halmaz középső elemével hasonlítjuk össze s-et. Ennek az s_i elemnek az i sorszáma lehet k vagy $k+1$, ha $n = 2k$. Ha pedig $n = 2k+1$, akkor $i = k+1$. Mit ad az s és s_i összevetése? Ha $s = s_i$, akkor egyetlen lépésben végeztünk. Ha $s < s_i$, akkor az s elemet elég már csak az $\{s_1, \ldots, s_{i-1}\}$ részhalmazban keresni. Ugyanígy, ha $s > s_i$, akkor az $\{s_{i+1}, \ldots, s_n\}$ részhalmazra szorítkozhatunk. Egyetlen jól irányzott kérdéssel tehát vagy megtaláljuk az s elemet, vagy pedig visszavezetjük a kérdést egy sokkal kisebb S_1 rendezett halmazban való keresésre. Az S_1 mérete legfeljebb az S méretének a fele: $|S_1| \leq |S|/2 = n/2$. Az eljárást ismételjük, amíg ez a felezés lehetséges. Az egyre zsugorodó célhalmazok legyenek

$$S_0 = S, S_1, \ldots, S_j.$$

Az utolsó S_j halmazról feltehetjük, hogy nem üres, más szóval $1 \leq |S_j|$, továbbá, hogy az S_j középső elemével való összehasonlítás már megválaszolja az ($s \in S$?) keresőkérdést. Eszerint a felhasznált összehasonlítások száma $j+1$. Az $|S_{i+1}| \leq |S_i|/2$ egyenlőtlenségek összefűzésével kapjuk, hogy $|S| = n$, $|S_1| \leq n/2$, $|S_2| \leq n/4, \ldots, |S_j| \leq n/2^j$. Az utolsó szerint $1 \leq n/2^j$, amiből $j \leq \log_2 n$.

Összesen tehát $j+1 \leq \log_2 n + 1$ összehasonlító lépés elég. Picivel pontosabb a $j+1 \leq \lceil \log_2(n+1) \rceil$ korlát. Ez $j \leq \lfloor \log_2 n \rfloor < \log_2(n+1)$ miatt érvényes.

Feladat: Mutassuk meg, hogy a bináris keresés optimális: kevesebb összehasonlítással nem oldható meg a feladat. (Alkalmazzuk az ellenség-módszert. Az ellenség úgy válaszol az algoritmus kérdéseire, hogy utána s keresését a nagyobbik részhalmazban kell folytatni.)

A két módszert, a lineáris keresést és a bináris keresést egybevetve megállapíthatjuk, hogy az utóbbi sokkal kevesebb összehasonlítást igényel. Mégsem mondhatjuk, hogy a lineáris keresés rossz módszer. Ha ugyanis az S halmazt listaként vagy külső táras állomány részeként kapjuk, akkor a bináris keresés többnyire nem alkalmazható. A gond ott van, hogy ekkor nem tudunk elég gyorsan a halmaz közepébe ugrani. A lineáris lépkedés pedig a legszerényebb szervezési módok mellett is megy. A tömbök fontos előnye, hogy alkalmasak a bináris keresésre. Általában a programozási környezet biztosítja, hogy az $A[1:n]$ tömb bármelyik $A[i]$ eleme gyorsan elérhető.

Az itt tárgyalt keresési feladatnak több értelmes változata van. Például az S lehet egy rendezett sorozat. A különbség annyi, hogy ekkor az s_i elemek között

ismétlődések is előfordulhatnak. A módszerek költségéről mondottak ekkor is érvényben maradnak, legalábbis ami a legrosszabb eseteket illeti.

Egy másik érdekes változat, amikor a keresett s elem összes S-beli előfordulását meg kell találnunk. A költség ekkor függ az ismétlődések számától is.

Példa: Tegyük fel, hogy az $A[1:n]$ tömbben

név	*cím*	*telefonszám*

alakú bejegyzéseket tárolunk, a *név* szerint rendezve. A rendezettség annyit tesz, hogy ha $i < j$, akkor az $A[i]$-ben levő *név* mező értéke nem nagyobb, mint az $A[j]$-ben levőé. Ha mondjuk Rezeda Kázmér telefonszámára vagyunk kíváncsiak, akkor bináris keresést használhatunk. Az A tömb legfeljebb $\lceil \log_2(n+1) \rceil$ elemének a megnézésével kiderül, hogy van-e információnk ilyen nevű személyről. Ha igen, akkor mindjárt kapunk is egy Rezeda Kázmérról szóló bejegyzést. Ha az összes Rezeda Kázmérra kíváncsiak vagyunk, akkor ettől a találati helytől jobbra, illetve balra lépkedve megtaláljuk valamennyit. Ez még körülbelül annyi tömbelem elérését jelenti, mint ahányszor ismétlődik a név.

Megemlítünk még egy rendezett halmazban való keresési módszert. Ez az ún. intelligens kereső rendszerekben fordul elő, használata elég bonyolult szervezést igényel. A neve *interpolációs keresés*. Csak a módszer alapgondolatát szeretnénk itt érzékeltetni. Amikor a telefonkönyvben például Fátyol Szilvia számát keressük, akkor nem fogunk az elejétől kezdve lineárisan lépkedni. Nem próbálkozunk a telefonkönyv közepével sem, a bináris keresésnek megfelelően. A kötetet inkább valahol az első negyede táján nyitjuk ki, mert úgy érezzük, hogy az F-fel kezdődő nevek arrafelé vannak. Az F-betűs neveknél pedig nagy léptekkel igyekszünk megtalálni az elsőket, hiszen az á az abc elejéről való.

Arról van itt szó, hogy információval rendelkezünk a tárolt halmaz elemeinek az eloszlásáról, aminek a segítségével jól-rosszul becsülni tudjuk a keresett elem lehetséges helyét. Az interpolációs keresők ilyen természetű ismeretek felhasználásával próbálják gyorsítani a keresést. Az információ minőségétől függően az összehasonlítások száma akár $\log \log n$-re is levihető.

2.2. Összehasonlítás alapú rendező módszerek

Itt az olyan rendező eljárások legfontosabbjait érintjük, amelyek nem tekinthetnek a rendezendő elemek (másik szokásos szóhasználattal: *kulcsok*) belsejébe. Az elemekkel kapcsolatos döntéseiket csak és kizárólag a $<$ relációra alapozhatják. Ezeket összehasonlítás alapú módszereknek nevezzük. Az eljárásokat tömbként

megadott bemenetek esetére ismertetjük. Az olvasóra bízzuk annak meggondolását, hogy az elmondottak miként és mennyire maradnak érvényben más megadási módokra. A feladat ezek után a következő:

Adott az $(U, <)$ rendezett halmazból (típusból) való elemekből álló $A[1 : n]$ tömb. Rendezzük át az A elemeit úgy, hogy azok a $<$ szerint nem csökkenő sorrendben legyenek.

A követelmény szerint az eljárás végén az A-nak ugyanazokat az elemeket kell tartalmaznia és ugyanakkora multiplicitással, mint kezdetben. A végállapotban érvényesek az $A[1] \leq A[2] \leq \ldots \leq A[n]$ egyenlőtlenségek. A módszerek költségének számításakor általában két tényezőt veszünk figyelembe: az összehasonlításokat és az elemek mozgatásait[1].

2.2.1. Buborék-rendezés

A rendezésére egy kézenfekvő ötlet, hogy a rosszul álló szomszédos elempárokat megcseréljük: ha valamely i-re $A[i] > A[i+1]$, akkor a két cella tartalmát kicseréljük. Ha már nincs ilyen értelemben rosszul álló pár, akkor A rendezett állapotban van. Az ötlet egy kulturált kivitelezése a *buborék-rendezés*. A tömb elejéről indulva a rosszul álló szomszédos elemeket cserélgetve eljutunk a tömb végéig. Ekkor a legnagyobb elem(ek egyike) $A[n]$-ben van. Utána ismételjük ezt az $A[1 : n-1]$ tömbre, majd az $A[1 : n-2]$ tömbre, stb. Végül A rendezett lesz.

procedure buborék
(az $A[1 : n]$ tömböt nem csökkenően rendezi *)*
for $(j = n-1, j > 0, j := j-1)$ **do**
  for $(i = 1, i \leq j, i := i+1)$ **do**
    { ha $A[i+1] < A[i]$, akkor cseréljük ki őket.}

A külső ciklus j változója vezérli az éppen aktuális $A[1 : j+1]$ tömb méretét. A belső ciklusban pedig végigmegyünk ezen a résztömbön. Az elvégzett összehasonlítások száma ennek megfelelően $n-1, n-2, \ldots, 2, 1$. Ez összesen $\frac{n(n-1)}{2}$ összehasonlítást jelent minden bemenetre. A legrosszabb esetben ugyanennyi az elemcserék száma is. Ez akkor fordulhat elő, ha kezdetben a tömb fordítottan rendezett: $A[1] \geq A[2] \geq \ldots \geq A[n]$.

[1] Feltevésünkkel elhanyagolunk néhány további költségtényezőt. Ilyen például a ciklusváltozók léptetésének az ideje. Ezek azonban nem befolyásolják az összköltség *nagyságrendjét*.

A módszer nevének magyarázatához képzeljük el, hogy a tömb függőleges helyzetben áll, $A[n]$ van legfelül és $A[1]$ legalul. Képzeljük el még azt is, hogy a nagyobb elemek könnyebbek a kisebbeknél. A módszer alkalmazásakor a könnyebb elemek felfelé mozognak, mint a buborékok valami folyadékban.

Az eljárás, mint később látni fogjuk, a versenytársainál sokkal több összehasonlítást használ, ha n nagy. A kódja viszont rövid, egyszerű, és könnyen átírható tömb helyett listával megadott bemenetre. Eme tulajdonságai miatt kis sorozatok (maximum 10-20 eleműek) rendezésére ajánlható. A módszer *stabil* (másik szokásos szóval: *konzervatív*) abban az értelemben, hogy az egyforma elemek egymás közti sorrendjén nem változtat.

2.2.2. Beszúrásos rendezés

A módszer alapgondolata igen egyszerű. Tegyük fel, hogy az $A[1 : k]$ résztömb már rendezett. Ezt használva rendezzük az $A[1 : k+1]$ résztömböt. Kezdetben, amikor is $k = 1$, a feltétel nyilván teljesül. Ezután sorra léptetve k-t rendezzük az $A[1 : 2]$, $A[1 : 3]$, ..., $A[1 : n-1]$, $A[1 : n]$ tömböket. Ezzel végül megoldjuk a feladatot.

Hogyan jutunk k-ról $k+1$-re? Meg kell találnunk az $A[k+1]$ elem helyét az $A[1 : k]$ rendezett résztömb elemei között. Ezt a *beszúrásnak* nevezett lépést egy példán keresztül mutatjuk be. Tegyük fel, hogy $k = 4$ és az $A[1 : 5]$ résztömb az alábbi:

1	5	7	9	6

Az $A[1 : 4]$ résztömb már rendezett. Ebben megkeressük az utolsó elemnek – ami esetünkben 6 – a rendezett sorrend szerinti helyét. Ezt tehetjük lineáris vagy bináris kereséssel. A keresés eredményeként kiderül, hogy a 6 helye az $A[2]$ és az $A[3]$ elemek között van. A beszúrást úgy végezhetjük, hogy az $A[3 : 4]$ résztömb elemeit eggyel jobbra toljuk, és a hatost az így szabaddá tett $A[3]$-ba mozgatjuk. Ezzel az $A[1 : 5]$ résztömböt rendeztük:

1	5	6	7	9

Nézzük meg a módszer költségét! Ez bizonyos mértékig függ a beszúró lépésben alkalmazott keresési eljárástól.

Beszúrásos rendezés lineáris kereséssel

Ami az összehasonlítások számát illeti, a legkedvezőtlenebb esetet akkor kapjuk, amikor az A tömb már a bemeneti állapotában rendezett. A $k+1$-edik

elem beszúrásakor ebben az esetben k összehasonlítást használunk. Ezeket $k = 1, 2, \ldots, n-1$-re összeadva $\frac{n(n-1)}{2}$ összehasonlítás adódik. Könnyű meggondolni, hogy a cserék szempontjából a fordítottan rendezett tömb adja a legrosszabb esetet. A $k+1$-edik elem beillesztéséhez $A[1 : k+1]$ mindegyik elemét mozgatni kell. Összesen ez $\frac{(n+2)(n-1)}{2}$ mozgatást jelent.

Vizsgálhatjuk a módszer átlagos viselkedését is. Hogy ezt megtehessük, meg kell mondanunk, hogy milyen lehetséges bemenetekre számoljuk az átlagot. Nos, tegyük fel, hogy az átlagot a tömbnek az $1, 2, \ldots n-1, n$ számokkal való összes lehetséges kitöltésére vesszük. Itt természetesen nincs jelentősége annak, hogy mik a tömbelemek, hiszen a módszer csak a $<$ relációt használja. Mindössze az számít, hogy n különböző elem, és ezáltal $n!$ lehetséges input sorrend van.

Feladat: Tegyük fel, hogy $A[1 : k]$ már rendezett, és az $A[k+1]$ elemmel még nem foglalkoztunk. Mutassuk meg, hogy a beszúrás során $A[k+1]$ ugyanannyiszor eshet az első k elem által meghatározott $k+1$ intervallum bármelyikébe. (Legyen H az $\{1, 2, \ldots, n\}$ egy $k+1$-elemű részhalmaza, és nézzük azokat az inputokat, amelyeknél az első $k+1$ elem pont a H-ból van. Ezek között mindegyik $i \in H$ ugyanannyiszor lesz a $k+1$. helyen.)

A feladat szerint teljesül a feltétel, amit a lineáris keresés átlagos viselkedésének elemzésekor megköveteltünk. Az összehasonlítások átlagos száma tehát a k-ról $k+1$-re lépésnél legalább $\frac{k}{2} + \frac{k}{k+1} > \frac{k}{2}$. Ezt összegezve ($k = 1, 2, \ldots, n-1$) kiderül, hogy az eljárás átlagosan legalább $\frac{n(n-1)}{4}$ összehasonlítást végez. Lényegében ugyanezen érvelés mutatja, hogy a mozgatások átlagos száma is $n^2/4$ körül van.

Beszúrásos rendezés bináris kereséssel

A már rendezett k elem közé a beszúrás $\lceil \log_2(k+1) \rceil$ összehasonlítást igényel. Ez a $k = 1, \ldots, n-1$ értékekre összesen

$$(*) \qquad K := \lceil \log_2 2 \rceil + \cdots + \lceil \log_2 n \rceil$$

összehasonlítást jelent. Innen tetszőleges n-re $K \leq n\lceil \log_2 n \rceil$. A K költséget a $(*)$ kifejezés alapján másként, valamivel pontosabban is becsülhetjük. Világos, hogy $\lceil \log_2 k \rceil < 1 + \log_2 k$, amiből

$$K < n - 1 + \log_2 2 + \cdots + \log_2 n = n - 1 + \log_2(n!).$$

Most segítségül hívjuk a Stirling-formulát, ami a faktoriális-függvény növekedési rendjét fejezi ki elemi függvényekkel. A nevezetes formula következő alakját

használjuk: $n! \sim (n/e)^n\sqrt{2\pi n}$. Logaritmusra térve

$$\log_2 n! \sim n(\log_2 n - \log_2 e) + \frac{1}{2}\log_2 n + \log_2\sqrt{2\pi} \leq n(\log_2 n - 1,442)$$

adódik, feltéve, hogy n elég nagy. Azt is használtuk itt, hogy $\log_2 e = 1,442695\ldots$, és $\log_2\sqrt{2\pi} = 1,32\ldots$. Kapjuk, hogy $K \leq n(\log_2 n - 0,442)$ elég nagy n-re. Az összehasonlítások száma sokkal kedvezőbb, mint a lineáris keresést használó változaté vagy a buborék-rendezésé. Rövidesen látni fogjuk, hogy ebből a szempontból a bináris keresést használó módszer közel optimális.

Ha viszont a mozgatásokat is számításba vesszük, akkor ez a módszer sem bizonyul túl jónak. A mozgatások számát illetően a lineáris változatról mondottak itt is érvényesek, hiszen a mozgatások száma nem függ a keresési módszertől. A legrosszabb esetben $\frac{(n+2)(n-1)}{2}$, átlagosan pedig mintegy $\frac{n^2}{4}$ mozgatást végzünk.

2.2.3. Egy alsó becslés

Itt egy egyszerű, általános alsó becslést adunk rendező módszerek összehasonlításainak számára. A korlát érvényes lesz minden összehasonlítás alapú eljárásra.

Tegyük fel, hogy van egy rendező algoritmusunk, ami helyesen rendezi az $(U, <)$ rendezett halmaz rögzített $u_1 < u_2 < \ldots < u_n$ elemének minden input permutációját. Ez azt jelenti, hogy az u_i elemek mind az $n!$ lehetséges bemeneti sorrendjét cserékkel átalakítja az $u_1, u_2, \ldots, u_n$ sorrendre. Tegyük még fel, hogy a módszer kizárólag két kimenetelű döntéseken alapul. Ezen azt értjük, hogy felírható, mint

$$\textbf{if } \textit{feltétel} \textbf{ then } \textit{akció}_1 \textbf{ else } \textit{akció}_2 \ ;$$

alakú utasítások egymásutánja. Itt $akció_i$ két dolgot jelenthet: egy másik utasításra való ugrást vagy pedig egy átrendezést. Az átrendezés az elemek akármilyen bonyolult csereberéje lehet.

Tegyük fel, hogy a módszer legfeljebb k döntéssel megoldja a feladatot az $u_1, u_2, \ldots, u_n$ elemek $n!$ lehetséges bemeneti sorrendjének bármelyikére. Megmutatjuk, hogy ekkor $k \geq \log_2(n!)$.

Évégből nézzük a módszer működését a $v_1, v_2, \ldots, v_n$ input sorozaton, és jegyezzük fel sorban a kapott döntések eredményeit. Ezáltal egy legfeljebb k hosszúságú, *igen*, *nem* értékekből álló sorozatot, *kódszót* kapunk.

Azt állítjuk, hogy így különböző bemeneti sorozatokhoz különböző kódszavakat rendelünk. Legyen ugyanis $v_1, v_2, \ldots, v_n$ és $w_1, w_2, \ldots, w_n$ két bemenet, amelyekhez ugyanaz a válasz-sorozat tartozik. Ekkor ugyanaz lesz a végrehajtásra kerülő átrendezések $P_1, P_2, \ldots, P_s$ sorozata is. Végül mindkét esetben a rendezett $u_1, u_2, \ldots, u_n$ sorrendet kapjuk. A P_i akciók *invertálhatók*, hiszen a

leképezés, amit megvalósítanak, kölcsönösen egyértelmű. Léteznek tehát a P_i^{-1} „visszarendezések". Ezért ha a $v_1, v_2, \ldots, v_n$ bemeneten a $P_1, P_2, \ldots, P_s$ átrendezések az $u_1, u_2, \ldots, u_n$ sorozathoz vezettek, akkor az utóbbira alkalmazva a $P_s^{-1}, P_{s-1}^{-1}, \ldots, P_1^{-1}$ átrendezéseket visszakapjuk a $v_1, v_2, \ldots, v_n$ sorozatot. Ugyanez elmondható a $w_1, w_2, \ldots, w_n$ bemenetre is. A két bemenet tehát megegyezik.

Ha egy bemenethez tartozó válaszsorozat k-nál rövidebb, akkor megfelelő számú *igen*t a végére írva egészítsük ki k hosszúságúra. Ezután is igaz marad, hogy különböző bemenetekhez különböző kódszavak tartoznak. Ez azon múlik, hogy egy (eredeti értelemben vett) kódszó nem lehet egy másik folytatása. Ugyanis amíg a válaszok egyeznek, addig pontosan ugyanazokat az utasításokat hajtjuk végre mindkét esetben. Ha a két kódszó az első j válaszig egyezik, és az egyik bemenetnél a program végére értünk, akkor a másiknál is ott leszünk.

Az eddigieket összegezve arra jutottunk, hogy az $n!$ lehetséges input mindegyikéhez egy k hosszúságú kódszót rendeltünk, mégpedig különböző bemenetekhez különböző kódszavakat. A kódszavak száma tehát legalább annyi, mint a bemenetek száma. A szóbajövő kódszavak száma 2^k, amiből $2^k \geq n!$, és végül $k \geq \log_2(n!)$ adódik. Érvényes tehát a következő:

Tétel: *Tegyük fel, hogy egy pusztán két kimenetelű döntéseket használó program legfeljebb k ilyen döntés alapján rendezi n elem bármelyik bemeneti sorrendjét. Ekkor $k \geq \log_2(n!)$ teljesül.* □

Példa: Az előző gondolatmenet illusztrálásaként vegyük a buborék-rendezést az $n = 3$ esetben. A ciklusok kifejtése után a program így fest:

if $A[2] < A[1]$ **then** $A[2] \leftrightarrow A[1]$;
if $A[3] < A[2]$ **then** $A[3] \leftrightarrow A[2]$;
if $A[2] < A[1]$ **then** $A[2] \leftrightarrow A[1]$.

Itt az $A[i] \leftrightarrow A[j]$ akció az $A[i]$ és $A[j]$ tartalmának cseréjét jelenti. Tegyük fel, hogy az 1,2,3 számok permutációit rendezzük. A következő kis táblázat megadja az egyes permutációkhoz tartozó kódszavakat. Látható, hogy hat különböző kódszót kaptunk.

Bemenet	kódszó
123	*nem, nem, nem*
132	*nem, igen, nem*
213	*igen, nem, nem*
231	*nem, igen, igen*
312	*igen, igen, nem*
321	*igen, igen, igen*

Az összehasonlítás alapú rendezések eleget tesznek az alsó becslésnél használt modell kikötésének. Ha az input sorozat (tömb, lista) elemei mind különbözők, akkor a $<$ reláció tesztelésével tényleg két kimenetelű döntéseket használunk. Egyenlőség nem fog előfordulni. A ciklusokkal kapcsolatos tesztek nem érdekesek. Az előző példához hasonlóan adott n-re a ciklusok kiküszöbölhetők. Érvényes ezért az alábbi:

Következmény: *Egy összehasonlítás alapú rendező módszer n elem rendezésekor legalább* $\log_2(n!)$ *összehasonlítást használ.* □

A beszúrásos rendezés jól megközelíti ezt a korlátot. Hátulütője viszont, hogy n^2-tel arányos számú mozgatást igényel, és így az összköltsége nagy lesz. Ezután egy olyan módszert ismertetünk, ami a mozgatások számában is állja a versenyt.

2.2.4. Összefésüléses rendezés

Két már rendezett sorozat (tömb, lista, stb.) tartalmának egy sorozatba való rendezése egyszerű feladatot jelent. Tegyük fel, hogy az $A[1:k]$ valamint a $B[1:l]$ tömbök rendezettek, és tartalmukat rendezetten tárolni akarjuk a $C[1:k+l]$ tömbben. A $C[1]$-be nyilván az $A[1]$ és $B[1]$ elemek közül a nem nagyobb kerül. Ha ez mondjuk $A[1]$, akkor az $A[2]$ és $B[1]$ egyike lesz $C[2]$. Általában, ha az $A[1:i]$ és $B[1:j]$ résztömböket már feldolgoztuk, tartalmuk nem csökkenően $C[1:i+j]$-ben van, akkor nyilván

$$C[i+j+1] := \min\{A[i+1], B[j+1]\}$$

a jó választás. Ezt az eljárást *összefésülésnek* nevezzük (jelölése: MERGE). Minden egyes összehasonlítás után egy elem a helyére kerül, kivéve az utolsót, amivel két elem helyét derítjük ki. A költség tehát $k+l-1$ összehasonlítás és $k+l$ mozgatás. Az összefésülés igen hatékony eljárás, hiszen a bemenő adatok hosszával arányos időben, sőt lényegében *egyetlen végigolvasással* elrendezi a két sorozatot. Hatékonysága miatt előszeretettel alkalmazzák a legváltozatosabb adatkezelő feladatokban.

Ilyen alkalmazásnak számít az *összefésüléses rendezés*. Az összefésüléses rendezés alapötlete, hogy először rendezzük külön-külön az $A[1:n]$ tömb első felét és második felét, majd ezek tartalmát fésüljük össze. Itt ismét az *oszd meg és uralkodj* elvvel találkozunk. Az $A[1:n]$ rendezésének a feladatát egy összefésülés árán visszavezetjük két feleakkora feladatra, az $A[1:\lceil n/2\rceil]$ és az $A[\lceil n/2\rceil+1:n]$ résztömbök rendezésére. E résztömböket természetesen ugyanezen ötlettel kívánjuk rendezni. Mindezek a következő rekurzív definíciót sugallják (az eljárás neve

MSORT, paramétere a rendezendő tömb). Ha $n > 1$, akkor

$$\text{MSORT}(A[1:n]) := \\ \text{MERGE}(\text{MSORT}(A[1:\lceil n/2\rceil]), \text{MSORT}(A[\lceil n/2\rceil + 1 : n])).$$

Hogy elvarrjuk a rekurzió alját, legyen $\text{MSORT}(A[i,i])$ az üres utasítás. A módszer költségének elemzéséhez tegyük fel először, hogy $n = 2^k$. Legyen $T(n)$ a felhasznált összehasonlítások maximális száma n-elemű bemenetekre. Figyelembe véve az összefésülés költségét, a rekurzió szerint

$$T(n) \leq n - 1 + 2T(n/2),$$

ha $n > 1$. Ezt $n/2$-re alkalmazva és visszahelyettesítve

$$T(n) \leq n - 1 + 2(n/2 - 1 + 2T(n/4)) = n - 1 + 2(n/2 - 1) + 4T(n/4).$$

A felezést így folytatva végül

$$T(n) \leq n - 1 + 2(n/2 - 1) + 4(n/4 - 1) + \cdots + 2^{k-1}(n/2^{k-1} - 1) \leq n\lceil \log_2 n\rceil.$$

Figyelembe vettük itt, hogy $T(1) = 0$. Az utolsó becsléshez elhagytuk a -1 tagokat, és használtuk, hogy k a legnagyobb egész, melyre $n/2^{k-1} > 1$. (Természetesen itt n választása miatt $n/2^k = 1$ is teljesül, és ezért $\lceil \log_2 n\rceil$ helyett egyszerűen $\log_2 n = k$ is írható.)

Az elemzésnél tulajdonképpen a rekurzió mélysége szerinti *fázisokba* csoportosítva számláltuk össze az összefésüléseknél fellépő összehasonlításokat. Az első fázisban két $n/2$ méretű tömböt fésülünk össze, a következőben két-két $n/4$ méretűt, stb. Összesen k fázis van. Egy fázisban az összefésülések legfeljebb $2n$ mozgatással megvalósíthatók; ez összesen $2n\lceil \log_2 n\rceil$ mozgatás. A tárigény $2n$ cella: az összefésüléseket egy $C[1:n]$ segédtömbbe végezhetjük, majd az eredményt visszaírjuk A-ba.

Megjegyzés: Az összefésülés megoldható helyben is: tegyük fel, hogy az $A[1:2n]$ tömb $A[1:n]$ és $A[n+1:2n]$ részei rendezettek. Ekkor a két rész összefésülhető az A tömbbe $O(n)$ összehasonlítással úgy, hogy A-n kívül még legfeljebb csak konstans számú cellát (pl. változókat) használunk. A módszerek azonban elég bonyolultak és nem praktikusak.

Az összefésüléses rendezés költségére adott elemzésünknél feltettük, hogy n a 2 egy hatványa. Az általános esetben tetszőleges pozitív valós x-re legyen $T(x)$ az $\lceil x\rceil$ elem rendezésénél fellépő összehasonlítások maximális száma. Érvényes a

$$T(x) \leq \lceil x\rceil - 1 + 2T(x/2)$$

egyenlőtlenség, amiből a korábbihoz hasonlóan adódik $T(n) \leq n\lceil \log_2 n \rceil$ tetszőleges pozitív egész n-re.

A programozástechnikában járatos olvasó tudja, hogy a rekurzív eljárások többnyire kevésbé hatékonyak, mint az iterációt használók. (Az érem másik oldala, hogy rekurzióval gyakran egyszerűbb, könnyebben olvasható leírást kapunk.) Az összefésüléses rendezést is érdemesebb iteratív formában megírni. Ekkor az első lépésben az A-ban szomszédos párokat rendezzük. Utána a szomszédos párokból rendezett négyeseket alakítunk ki összefésüléssel, majd ezekből nyolcasokat, stb. Erre a megoldásra is érvényesek a rekurzív változat költségéről mondottak: $\lceil \log_2 n \rceil$ összefésülő fázisban legfeljebb $n\lceil \log_2 n \rceil$ összehasonlítást végzünk.

Az összefésüléses rendezés költsége – mindkét költségtényezőt figyelembe véve – $O(n \log n)$. Ugyanakkor az alsó becslésünk szerint $cn \log n$ összehasonlítás szükséges is. E két tényt összerakva megállapíthatjuk, hogy az összefésüléses rendezés *konstans szorzó erejéig optimális módszer.*

2.2.5. A kupac adatszerkezet és a kupacos rendezés

O brightening glance,
How can we know the dancer from the dance?
WILLIAM BUTLER YEATS

A következő rendezési módszerhez először egy *adatszerkezettel* fogunk megismerkedni. Adatszerkezeten bizonyos adat-fajták – *típusok* – és velük kapcsolatos funkciók, *műveletek* együttesét értjük. Bizonyosan mindenki számára ismerős a *lista* adatszerkezet. Egyetlen típusból áll, amit *elemnek* nevezünk. A műveletek elemek véges halmazainak a kezelésére szolgálnak. Ilyenek például az *első, megelőző, következő* elemeket megadó eljárások.

A *kupac* adatszerkezettel egy $(U, <)$ rendezett halmaz (a kulcsok univerzuma) egy S véges részhalmazát szeretnénk tárolni. Két művelet tartozik a szerkezethez. Az egyik a beszúrás: U egy elemének hozzávétele S-hez. A másik az S halmaz $<$ szerinti minimális elemének a törlése. Legyen ezen műveletek neve BESZÚR és MINTÖR. A kupac adatszerkezet tehát egy $(U, <)$ rendezett típus e két művelettel. Mire jó egy ilyen eljáráspár? A számos fontos alkalmazás közül itt hármat említünk.

1. Bizonyos (komolyabb) operációs rendszerekben a beérkező munkákhoz (jobokhoz) prioritások rendelhetők. A rendszer mindig a legkisebb prioritási értékű munkát indítja el először. A még el nem kezdett munkákat egy kupacban tárolhatjuk. Az operációs rendszerhez érkező munkákat BESZÚR műveletekkel tesszük a kupacba. A következő indítandó munkát egy MINTÖR adja; ezzel a munka egyben

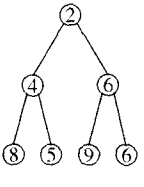
ki is kerül a kupacból. Ebből az alkalmazásból ered a kupac régebbi terminológia szerinti neve: *prioritási sor* (priority queue).
2. Több rendezett adatállomány összefésülése esetén célszerű lehet az egyes bemenő állományok legkisebb még feldolgozatlan elemeit egy kupacban tartani. A következő kiírandó elemet egy MINTÖR adja.
3. Egy gyors és bűbájos rendező eljárás, *a kupacos rendezés*, amit itt bemutatunk.

A kupac adatszerkezet egy hatékony implementációja: a bináris kupac
Emlékeztetünk a *bináris fa* fogalmára. Ez egy olyan fa, amelynek csúcsai szinteken helyezkednek el. A legfelső szinten egyetlen csúcs van, ezt *gyökérnek* nevezzük. Egy tetszőleges x csúcsból legfeljebb két él indul ki. Az élekhez irány tartozik (*bal, jobb*). Ezek eggyel alacsonyabb szinten levő csúcsokhoz vezetnek. A balra menő él végpontja az x *bal fia*, a jobbra menő él végpontja az x *jobb fia*. Egy a gyökértől különböző csúcsba pontosan egy él megy. Ennek megfelelően a gyökértől különböző csúcsoknak egyértelműen meghatározott *apjuk* van. Azokat a csúcsokat, amelyeknek nincs fiuk, *leveleknek* nevezzük. A nem-levél csúcsokat szokás *belső csúcsoknak* is nevezni.

Bizonyos szép, telt bináris fák jól ábrázolhatók tömbben, pontosabban egy tömb tekinthető bináris fának a következők szerint: a fa csúcsai az $A[1:n]$ tömb elemei. Az $A[i]$ csúcs bal fia $A[2i]$, a jobb fia pedig $A[2i+1]$. Ebből adódóan ha $j > 1$, akkor az $A[j]$ csúcs apja $A[\lfloor j/2 \rfloor]$ lesz.

Az ilyen alakban ábrázolható bináris fákat *teljesnek* nevezzük. A teljes fák levelei legfeljebb 2 szinten, a legalsón és esetleg a felette levőn helyezkednek el, és legfeljebb egy kivétellel minden belső csúcsuknak 2 fia van.

A kupac elemeit egy teljes bináris fában tároljuk. A fa csúcsaiban vannak a tárolni kívánt S halmaz elemei Egy csúcsban egy elem van; megeshet viszont, hogy egy $s \in S$ elem több csúcsban is előfordul. A fára teljesül a *kupac-tulajdonság*: egy tetszőleges csúcs eleme nem lehet nagyobb a fiaiban levő elemeknél. A következő példában az elemek természetes számok (a szokásos rendezéssel). A tárolt S halmaz – a többszörös elem jelenlétét is feltüntetve – $\{2, 4, 5, 6, 6, 8, 9\}$.



Ha a kupacnak (multiplicitásokkal számolva) n eleme van, akkor a keretül

szolgáló teljes bináris fa egy n elemű tömbben tárolható. Az előző példának megfelelő $A[1:7]$ tömb így fest:

2	4	6	8	5	9	6

Nézzük meg ezután, hogy miként építhető kupac n elemből. Először az elemeket beletesszük az $A[1:n]$ tömbbe, amit most teljes bináris fának is tekintünk. A sorrend egyelőre tetszőleges; mondjuk abban a sorrendben, ahogy olvassuk őket. Ezután gondoskodni kell a kupac-tulajdonságról.

Az eljárás leírásához hasznosak lesznek a következő jelölések: legyen f egy részfa gyökere, jelölje a az itt tárolt elemet, az f fiai legyenek f_1, f_2, az elemek itt a_1, illetve a_2.

Tegyük fel, hogy a kupaccá alakítás útján már tartunk valahol: az f_1 és f_2 gyökerű részfák már kupacok. Szeretnénk innen elérni, hogy az f gyökerű részfára is teljesüljön a kupac-tulajdonság. Ezt hivatott biztosítani a *felszivárog* eljárás.

felszivárog(f)
{ Ha $\min\{a_1, a_2\} < a$, akkor $\min\{a_1, a_2\}$ és a helyet cserél.
Ha az a elem a_i-vel cserélt helyet, akkor *felszivárog*(f_i). }

A feltételek szerint a részfában a kupac-tulajdonság csak az f csúcsnál sérülhet. Ha ez a helyzet, akkor az elemcsere után a gyökérrel és a másik részfával már nem lehet baj. Elég az f_i gyökerű részfát kupaccá tenni. Erre hivatott a *felszivárog*(f_i) hívás, aminek szintén teljesülnek a feltételei. Valójában az történik, hogy az a elem addig megy lefelé a fában, amíg sérti a kupac-tulajdonságot. Tehát ha előbb nem, egy levélben bizonyosan nyugvópontra jut. Egy lefelé lépés költsége 2 összehasonlítás és egy csere. Ha a részfának l szintje van, akkor ez legfeljebb $l-1$ csere és $2(l-1)$ összehasonlítás.

Legyen most f egy teljes bináris fa (gyökere), a fa csúcsaiban S elemeivel. A következő eljárás az egész fát kupaccá teszi.

kupacépítés(f)
{ Az f fa v csúcsaira lentről felfelé, jobbról balra *felszivárog*(v). }

Vegyük észre, hogy a *felszivárog* hívásainál teljesülnek a hívási feltételek; amikor a *felszivárog*(v) hívás sorra kerül, akkor v fiait már rendbetettük. A hívási sorrend az $A[1:n]$ tömbön nézve azt jelenti, hogy $A[n]$ felől haladunk $A[1]$ felé.
A *kupacépítés* eljárás költsége: tegyük fel, hogy a fának n eleme és l szintje van. (A gyökér az első szint.) Ekkor nyilván $n \leq 2^l - 1$. A fa teljessége miatt azt is tudjuk, hogy $2^{l-1} \leq n$, amiből $l \leq \log_2 n + 1$.

Az i-edik szinten levő csúcsok száma nem több, mint 2^{i-1}. Az i. szinten levő v csúcsra *felszivárog*(v) költsége legfeljebb $l-i$ csere és legfeljebb $2(l-i)$ összehasonlítás. A cserék száma ezért összesen legfeljebb $\sum_{i=1}^{l}(l-i)2^{i-1}$. A $j=l-i$ (azaz $i=l-j$) helyettesítés után a cserék száma így becsülhető:

$$\sum_{j=0}^{l-1} j2^{l-j-1} = 2^{l-1}\sum_{j=0}^{l-1} j/2^j < 2^l \leq 2n.$$

Csak az utolsó előtti egyenlőtlenség érdemel némi indoklást. A $j/2^j$ alakú tagok táblázatba foglalhatók, ahol egy tagnak egy sor felel meg:

$$\begin{array}{llll} 1/2 & & & \\ 1/4 & 1/4 & & \\ 1/8 & 1/8 & 1/8 & \\ \vdots & & & \\ \frac{1}{2^{l-1}} & \frac{1}{2^{l-1}} & \frac{1}{2^{l-1}} & \cdots \quad \frac{1}{2^{l-1}} \end{array}$$

A táblázat oszlopainak összegei kisebbek mint 1, $1/2, \ldots, 1/2^{l-1}$, ezek összege pedig kisebb mint 2. A szükséges cserék száma tehát legfeljebb $2n$. Az összehasonlítások száma az előbbi összeg kétszeresével becsülhető; így nem lehet több, mint $4n$. Összegezésül elmondhatjuk, hogy a *kupacépítés* eljárás lineáris költséggel megvalósítható.

Nézzük ezután a kupacra jellemző alapműveleteket:
A MINTÖR megvalósítása: a törlendő elem a kupac-tulajdonság miatt a fa f gyökerében van. Ezt kivesszük innen, majd f-be tesszük a fa utolsó szintjén a jobb szélső csúcsban levő elemet; az utóbbi csúcsot töröljük, majd *felszivárog* (f) következik. Az összköltség $O(l) = O(\log n)$. Az $A[1:n]$ tömbön szemlélve mindezt: a hívás az $A[1]$ elemet adja vissza, $A[n]$ kerül $A[1]$-be, erre meghívjuk a *felszivárog* eljárást, majd a tömb méretét csökkentjük. Az új kupac $A[1:n-1]$-ben lesz.
BESZÚR(s) megvalósítása: új levelet adunk a fához (ügyelve a teljességre). Ebbe a levélbe tesszük az s elemet. Ezután s-et mozgatjuk felfelé, amíg az őt tartalmazó csúcs apjában levő s' elemre igaz, hogy $s < s'$. Az összköltség $O(l) = O(\log n)$. Mit jelent ez tömbös ábrázolásban? Az $A[1:n]$ tömbhöz új elemet veszünk. Az $A[n+1]$-be tesszük az s elemet, ami innen kezdi meg felfelé vándorlását.

A kupac adatszerkezet itt leírt implementációját, a *bináris kupac*ot szokás egyszerűen csak kupacnak nevezni. Az irodalomban gyakran előfordul, hogy a terminológia nem tesz különbséget a szerkezet és annak valamelyik megvalósítása között.

Bizonyos alkalmazásokban hasznos a FOGYASZT eljárás. Ennek célja egy a kupacban levő elem „kulcsának csökkentése"; pontosabban fogalmazva kisebbel való helyettesítése. Két bemenő paramétere van; az egyik a szerkezet egy s eleme, a másik az U egy s-nél kisebb s' eleme. Az s-ről feltesszük, hogy ismerjük a kupacbeli helyét, tehát nem kell megkeresni. FOGYASZT(s, s') az s helyére az $s' < s$ elemet teszi, majd helyreállítja a kupac-tulajdonságot. Utóbbi a beszúrásnál látott módon az s' felfelé való mozgatásával érhető el. A művelet költsége ezért $O(l) = O(\log n)$.

A kupacos rendezés

A kupacos rendezés módszerét J. W. J. Williams és R. W. Floyd javasolták 1964-ben. Bemenete a rendezendő sorozatot tartalmazó $A[1 : n]$ tömb. Először kupacot építünk, utána n darab MINTÖR adja nem csökkenő sorrendben az elemeket. Az összköltség az előzőek alapján $O(n) + O(n \log n) = O(n \log n)$. A kupacos rendezés elméletileg a leggyorsabb ismert rendező módszer. Ez úgy értendő, hogy az összesített költségére ismert felső becslések a legjobbak a rendező eljárások korlátai közül. Gondos, pontos implementáció és becslés mellett a korlátok $2n\lfloor \log_2 n \rfloor + 3n$ (összehasonlítások száma) és $n\lfloor \log_2 n \rfloor + 2,5n$ (cserék száma).

A kupac egy általánosabb implementációja: d-kupacok

A kupac adatszerkezet egy olyan implementációját ismertük meg, amelyben az elemeket tartalmazó alapstruktúra egy tömbben ábrázolt bináris fa. Pontosabban szólva az $A[1 : n]$ tömb elemeit egy bináris fa csúcsainak tekintettük. Ez a szemlélet könnyen általánosítható. Legyen $d > 1$ egy egész. Az A tömbön egy olyan gyökeres fa-struktúrát definiálunk, amelyben egy csúcsnak legfeljebb d fia van. A fa gyökere legyen itt is $A[1]$. Az $A[i]$ csúcs fiai legyenek az $A[d(i-1)+2], A[d(i-1)+3], \ldots A[di+1]$ elemek, illetve ezek közül azok, amelyek indexe nem nagyobb, mint n. Ennek megfelelően $i > 1$-re az $A[i]$ csúcs apjának indexe $\lceil (i-1)/d \rceil$. Nevezzük az így nyert fát *teljes d-fának*. Egyszerűen meggyőződhetünk róla, hogy a $d = 2$ esetben éppen a teljes bináris fákat kapjuk. Egy teljes d-fa levelei legfeljebb 2 szinten helyezkednek el, és legfeljebb egy kivétellel minden nem-levél csúcsának éppen d fia van. Ennek folyományaként ha a fa l szintből áll, akkor az első $l-1$ szinten $1 + d + d^2 + \cdots + d^{l-1} = (d^l - 1)/(d-1)$ csúcs van. Innen $d^l/(d-1) \leq n$, és d alapú logaritmusra térve $l \leq \log_d n + 1$.

A kupac adatszerkezet *d-kupac* elnevezésű implementációjában az S elemeit egy teljes d-fa csúcsaiban helyezzük el úgy, hogy teljesüljön a kupac-tulajdonság: egy csúcs eleme nem lehet nagyobb a fiaiban levő elemeknél. A korábbi implementációra – amit immár 2-kupacnak is hívhatunk – megismert algoritmusok kézenfekvően általánosíthatók d-kupacokra.

Feladat: Mutassuk meg, hogy egy n elemű d-kupac esetén az alapeljárások megvalósíthatók az alábbi táblázatban foglalt költségkorláton belül. Az f egy olyan csúcsot jelöl, amelynek a részfája k szintből áll:

Eljárás	Műveletigény
felszivárog(f)	$O(dk)$
kupacépítés	$O(n)$
MINTÖR	$O(d \log_d n)$
BESZÚR	$O(\log_d n)$
FOGYASZT	$O(\log_d n)$

(A módszerek a $d = 2$ esetre látottak kézenfekvő általánosításai. A *kupacépítés* elemzéséhez hasznos, hogy a fa i-edik szintjén legfeljebb d^{i-1} csúcs van, és $\sum_{i \geq 0} i/d^i \leq \sum_{i \geq 0} i/2^i \leq 2$.)

2.2.6. Gyorsrendezés

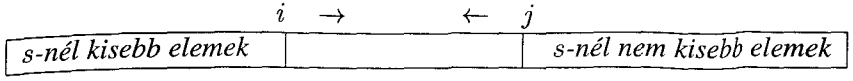
A módszert C. A. R. Hoare találta 1960-ban. Az alapötlet ismét egy szép példa az *oszd meg és uralkodj* elv alkalmazására. A rendezendő $A[1 : n]$ tömb egy véletlen s elemét vesszük. Ezután a tömb elejébe mozgatjuk az s-nél kisebb elemeket, a végébe az s-nél nagyobbakat. A kettő között helyezkednek el az s előfordulásai. Nevezzük PARTÍCIÓnak az eljárást, ami ezt megvalósítja. A PARTÍCIÓ(s) tehát felbontja három részre az A tömböt. Az $A[1 : k]$ résztömb elemei s-nél kisebbek, $A[k + 1 : l]$ minden eleme s, végül $A[l + 1 : n]$ elemei s-nél nagyobbak. A PARTÍCIÓ(s) hívás utáni helyzet így szemléltethető:

s-nél kisebb elemek	s	...	s	*s-nél nagyobb elemek*

Világos, hogy eztán már elegendő az első és utolsó résztömböt rendezni. A GYORSREND eljárás vázlata következik.

GYORSREND($A[1 : n]$)
1. Válasszunk egy véletlen s elemet az A tömbből.
2. PARTÍCIÓ(s); az eredmény legyen az $A[1 : k]$, $A[k + 1 : l]$, $A[l + 1 : n]$ felbontás.
3. GYORSREND($A[1 : k]$); GYORSREND($A[l + 1 : n]$).

A GYORSREND-algoritmus lelke a partícionáló eljárás. Érdemes ezért jobban szemügyre venni. Tegyük fel, hogy az $A[1 : n]$ tömbbel és az s elemmel dolgozunk. Az i és j változók értéke kezdetben 1, illetve n. Először i-t növeljük, amíg $A[i] < s$ teljesül. Utána j-t csökkentjük, amíg $A[j] \geq s$.



Ha mindkettő megáll (nem lehet továbblépni), és $i < j$, akkor kicseréljük $A[i]$ és $A[j]$ tartalmát. Ezután $i := i + 1$ és $j := j - 1$. Ha a két mutató összeér (már nem teljesül $i < j$), akkor s előfordulásait a felső rész elejére mozgatjuk. Innen világos, hogy PARTÍCIÓ(s) költsége $O(n)$.

A GYORSREND eljárás futási ideje nagymértékben függ attól, hogy a partícionáló elem mekkora méretű részekre (mennyire egyenletesen) vágja szét a tömböt. (Gondoljuk meg, mi történik, ha mindig a rendezés szerinti legkisebb elemet használjuk.) Itt van szerepe a véletlen választásnak. Ekkor nagy az esélye, hogy s eléggé egyenletes vágást biztosít.

A gyorsrendezés várható költsége

Az egyszerűség kedvéért feltesszük, hogy a rendezendő $A[1 : n]$ tömb elemei az $1, 2, \ldots, n$ számok. Jelölje $C(n)$ az A rendezéséhez felhasznált összehasonlítások várható számát. Ez a mennyiség ugyanaz lesz mind az $n!$ lehetséges bemeneti sorrendre, hiszen az összehasonlítások száma csupán a partícionáló elemek választásától függ. Nyilván igaz, hogy $C(0) = C(1) = 0$. Legyen $C(n, j)$ az összehasonlításokban mért várható költség abban az esetben, amikor a legelőször választott partícionáló elem éppen j. A módszer szerkezetéből látjuk, hogy

$$C(n, j) = n - 1 + C(j - 1) + C(n - j).$$

Az első tag a partícionálás összehasonlításaiból adódik, a következő kettő pedig az alsó, illetve a felső rész rendezéséhez használt összehasonlítások várható száma. A partícionáló elemek (egyenletes eloszlás szerinti) véletlen választása azt jelenti, hogy mindegyik j ugyanakkora eséllyel lesz az első partícionáló elem. Érvényes tehát a következő:

$$C(n) = \frac{1}{n}\left(C(n, 1) + C(n, 2) + \cdots + C(n, n - 1) + C(n, n)\right).$$

Innen az előző egyenleteket használva kiküszöbölhetjük a $C(n, j)$ mennyiségeket:

$$(+) \qquad C(n) = n - 1 + \frac{2}{n}\left(C(1) + C(2) + \cdots + C(n - 1)\right),$$

amiből n-nel való szorzás után

$$nC(n) = n(n - 1) + 2\left(C(1) + C(2) + \cdots + C(n - 1)\right).$$

A kapott kifejezés hátulütője, hogy túl sok $C(i)$ alakú tag szerepel benne. Ezen segíthetünk, ha n helyett $n - 1$-re is felírjuk a formulát, majd ezt kivonjuk (+)-ból:

$$nC(n) - (n - 1)C(n - 1) = 2(n - 1) + 2C(n - 1).$$

Innen átrendezés és $n(n+1)$-gyel való osztás után a következő egyenlőséget nyerjük:

$$\frac{C(n)}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{C(n-1)}{n},$$

ami elég egyszerű szerkezetű rekurziót ad a $C(n)/(n+1)$ mennyiségekre. Mivel csak felső korlátot akarunk, további egyszerűsítést érhetünk el a jobboldal első tagjának kis növelésével. A számlálóban $n-1$ helyett $n+1$-et írva

$$\frac{C(n)}{n+1} < \frac{2}{n} + \frac{C(n-1)}{n}.$$

Ezt ismételten önmagába helyettesítve végül azt nyerjük, hogy

$$\frac{C(n)}{n+1} < \frac{2}{n} + \frac{2}{n-1} + \cdots + \frac{2}{3} + \frac{2}{2} + \frac{2}{1}.$$

A jobboldal a nevezetes $H_n = 1 + 1/2 + 1/3 + \cdots + 1/n$ *harmonikus szám* kétszerese. A H_n számok nagyságrendjéről ismert, hogy $H_n = \ln n + \gamma + o(1)$, ahol $\gamma = 0.55721\ldots$ az Euler-állandó. Mindezeket összeolvasva azt kapjuk, hogy

$$C(n) < 2n \ln n + O(n) \approx 1,39n \log_2 n + O(n).$$

Itt használtuk, hogy $\ln n = (\ln 2) \log_2 n$ és $\ln 2 \approx 0,69$.

A módszer tehát várhatóan csak mintegy 39 százalékkal használ több összehasonlítást annál, amit az alsó korlát megkövetel. Valójában a gyorsrendezés a várható futási időt tekintve az egyik leggyorsabb ismert módszer annak ellenére, hogy a legrosszabb esetben akár cn^2 is lehet a költsége. Gyakorlati viselkedése alapján a *legjobbnak tartott* összehasonlítás alapú módszer memóriában levő állományok (lista, tömb) rendezésére.

2.2.7. A k-adik elem kiválasztása

Tegyük fel, hogy adott az $(U, <)$ rendezett típusból való elemek egy $u_1, u_2, \ldots, u_n$ sorozata, és egy k egész, $(1 \leq k \leq n)$. A feladat az, hogy határozzuk meg a sorozatnak a $<$ rendezés szerinti k-adik legnagyobb elemét.

A $k = n$ eset a maximális elem(ek egyikének) kiválasztását jelenti. Már a buborék-rendezés kapcsán láttuk, hogy ez megtehető $n-1$ összehasonlítással. Ennél kevesebb általában nem elég. Ahhoz ugyanis, hogy a maximálistól különböző u elemről tényleg biztosan tudjuk, hogy nem maximális, szükség van egy olyan összehasonlításra, amelynél u kisebbnek bizonyul. A maximumtól különböző $n-1$ elem mindegyike legalább egyszer alulmaradt. Mivel egy ilyen mérkőzésnek legfeljebb egy vesztese van, a mérkőzések száma legalább $n-1$. Hasonló mondható a $k = 1$ esetről, a minimális elem kiválasztásáról.

A második legnagyobb elemet $n + \lceil \log_2 n \rceil - 2$ összehasonlítással azonosíthatjuk. Egy lehetséges módszer a sportból jól ismert kieséses verseny. Állítsuk az elemeket párokba, hasonlítsuk össze a párok tagjait, majd a győztesekkel járjunk el ugyanígy. Végül $\lceil \log_2 n \rceil$ fordulóban összesen $n - 1$ mérkőzést játszva kiderül, hogy ki a győztes. A második helyezett nyilván azok között van, akik csak a bajnok ellen vesztettek. Ilyen jelöltekből legfeljebb $\lceil \log_2 n \rceil$ lehet; $\lceil \log_2 n \rceil - 1$ mérkőzéssel eldönthető, hogy közülük ki a legjobb. Az is igaz, hogy $n + \lceil \log_2 n \rceil - 3$ összehasonlítás nem mindig elég. Ezt nem részletezzük.

Általában elmondhatjuk, hogy $O(n \log n)$ összehasonlítással tetszőleges k-ra célt érünk. Ekkora költséggel rendezhető az u_i sorozat, ami után a k-adik elem könnyen meglelhető.

Ha $k \leq n/\log n$, akkor lineáris számú, azaz $O(n)$ összehasonlítás elég: építsünk kupacot az u_i elemekből, majd hajtsunk végre k egymás utáni MINTÖR-t. A költség $O(n) + kO(\log n) = O(n) + O((n/\log n)\log n) = O(n)$. Ugyanez érvényes, ha $k \geq n - n/\log n$.

A kiválasztás feladata általában is megoldható lineáris költséggel. Az ilyen módszerek azonban meglehetősen bonyolultak és csak nagy n értékekre tudnak versenyezni az egyszerűbbekkel, mint amilyen a teljes rendezésre építő eljárás. Mindezek ellenére érdemes megismerkedni e módszerek boszorkányos redukciós ötletével.

Tegyük fel tehát, hogy a k-adik elemet akarjuk kiválasztani. Jelölje $T(n)$ az ismertetésre kerülő módszer összehasonlításainak maximális számát n elem és tetszőleges k mellett. Az egyszerűség kedvéért feltesszük még, hogy az u_i elemek mind különbözők. Olyan elemet igyekszünk találni – nevezzük ezt *polgárnak* –, ami mindkét szélsőségtől távol van abban az értelemben, hogy legalább $n/4$ elem kisebb nála, és legalább $n/4$ elem nagyobb nála. Ha van egy ilyen elemünk, akkor $n - 1$ összehasonlítással megkaphatjuk a nála kisebb elemek S_1 és a nála nagyobb elemek S_2 halmazát. Ha $k < |S_1|$, akkor ezután elég S_1 k-adik elemét megtalálni, ha pedig $k > |S_1| + 1$, akkor S_2-nek a $k - |S_1| - 1$-edik eleme lesz az új célpont. Mindkét esetben kevesebb, mint $3n/4$ elem közül kell választani. Egy polgár segítségével tehát drámai módon csökkenthető a feladat mérete. A gondolatmenetből a $T(n)$ függvényre az alábbi egyenlőtlenség következik:

$$T(n) \leq n - 1 + \textit{ a polgár költsége } + T(3n/4).$$

Világos innen, hogy akkor kapunk erős korlátot $T(n)$-re, ha gyorsan tudunk polgárt találni. Ezt pedig a következőképpen tesszük:

1. Az n elemet ötös csoportokba soroljuk, elhagyva a végén keletkező 0-4-elemű csoportocskát. Összesen $\lfloor n/5 \rfloor$ ilyen csoportunk lesz.

2. Minden ötös csoportnak meghatározzuk a rendezés szerinti középső elemét; ez csoportonként legfeljebb 6 összehasonlítás, összesen nem több, mint $6\lfloor n/5 \rfloor$.
3. Az előbb meghatározott $\lfloor n/5 \rfloor$ elem közül kiválasztjuk a középsőt. Legyen ez a polgár. Ez az $\lfloor \frac{n+5}{10} \rfloor$-edik elemet jelenti. A választást az éppen tárgyalt kiválasztó algoritmussal végezzük. Ha úgy tetszik, itt egy *rekurzív hívás* történt.

A polgár nem kisebb az n elem közül $3\lfloor \frac{n+5}{10} \rfloor$-nél. Ugyanis nem kisebb, mint $\lfloor \frac{n+5}{10} \rfloor$ középső elem, amelyek nagyobbak a csoportjukban még további két-két elemnél. Egyszerű számolás mutatja, hogy ha $n \geq 25$, akkor $3\lfloor \frac{n+5}{10} \rfloor > n/4$. Eszerint a választott elemünk nagyobb legalább $n/4$ elemnél. Hasonló érveléssel kapjuk, hogy ha $n \geq 25$, akkor a polgár kisebb legalább $n/4$ elemnél. Tényleg rászolgál tehát a polgár névre. Ha $n \leq 24$, akkor a k-adik elemet közvetlenül (polgár választása nélkül) megkaphatjuk. Ez – összefésüléses rendezéssel – legfeljebb $n\lceil \log_2 n \rceil \leq 5n$ összehasonlítás.

Mibe kerül a polgárválasztás? A második lépés költsége legfeljebb $6n/5$ összehasonlítás, a harmadiké legfeljebb $T(n/5)$. A $T(n)$-re nyert egyenlőtlenség ezekkel az adatokkal így alakul:

$$T(n) \leq \begin{cases} 5n & \text{ha } n \leq 24 \\ 11n/5 + T(n/5) + T(3n/4) & \text{ha } n > 24. \end{cases}$$

Innen egyszerű indukcióval következik, hogy $T(n) \leq 44n$. Ez nyilván igaz, ha $n \leq 24$. Nagyobb n-ekre az indukciós feltevést használva

$$T(n) \leq 11n/5 + T(n/5) + T(3n/4) \leq 11n/5 + 44n/5 + 44 \cdot 3n/4 = \\ = 44n/20 + 4 \cdot 44n/20 + 15 \cdot 44n/20 = 20 \cdot 44n/20 = 44n.$$

Pontosabb és egyszersmind bonyolultabb becsléssel a korlát $7n$-re levihető. Ezt nem részletezzük.

2.3. Kulcsmanipulációs rendezések

Az eddig tárgyalt módszerek igen keveset tételeztek fel az $(U, <)$ rendezett típusról (univerzumról), amihez a rendezendő elemek tartoznak. Kizárólag $s < s'$ alakú összehasonlítások eredményei alapján oldották meg a feladatot. Gyakran a $<$ reláción kívül sok mást is tudunk U-ról. Ismerhetjük például az elemei számát, vagy az elemek (kulcsok) belső szerkezetét. Ilyen ismeretekre építve az eddigieknél hatékonyabb rendezési módszereket adhatunk.

2.3.1. Ládarendezés (binsort)

Arról az eljárásról lesz itt szó, amit feltehetőleg Jim használt a Kincses Szigeten, amikor a sokféle érmét szortírozta. A módszer akkor hasznos, ha a lehetséges elemek U tartománya n-hez, a rendezendő elemek számához képest nem túl nagy. Tegyük fel, hogy az $A[1:n]$ tömböt szeretnénk rendezni, és tudjuk, hogy A elemei az m elemű U univerzumból kerülnek ki. Ekkor lefoglalhatunk egy U elemeivel indexelt B tömböt (U egy rendezett típus!). Ennek elemei - a ládák - U-beli elemek listái lesznek. Kezdetben minden lista (láda) üres. Az eljárás első fázisában végigolvassuk az A tömböt, és az $s = A[i]$ elemet a $B[s]$ lista végére fűzzük. A második fázisban az elejétől a végéig növő sorrendben végigmegyünk B-n, és a $B[i]$ listák tartalmát visszaírjuk A-ba.

Az eljárás költsége $O(n+m)$ elemi lépés. Létrehozzuk az üres listákból álló B tömböt. Erre $O(m)$ költséget számíthatunk. Az első fázis, A végigolvasása és az elemeinek listákba illesztése $O(n)$ elemi lépést jelent. Végül a második fázisban a listák olvasása és az elemek visszatevése A-ba további $O(m+n)$ elemi lépéssel megtehető.

Eszerint, ha az U kulcstartomány n-hez képest lineáris méretű (másként mondva: $m \leq cn$ teljesül egy c állandóval), akkor ládarendezéssel $O(n)$ költséggel, azaz lineáris időben tudunk rendezni. Ez kedvezőbb, mint amit összehasonlítás alapú módszerekkel elérhetünk. Javasoljuk az olvasónak: gondolja meg, miért nem érvényes itt az összehasonlító módszerekre adott alsó becslés.

Példa: Tegyük fel, hogy a rendezendő $A[1:7]$ tömb elemei 0 és 9 közötti egészek:

A :

5	3	1	5	6	9	6

Ekkor egy tízelemű $B[0:9]$ tömböt lesz célszerű használni. Az első fázis után B így képzelhető el:

B :

	1		3		5 5	6 6			9

A $B[0]$, $B[2]$, $B[4]$, $B[7]$, $B[8]$ listák üresek, hiszen A-ban nem fordulnak elő a 0, 2, 4, 7, 8 értékek. A $B[5]$ és $B[6]$ listák pedig kételeműek, mert az 5 és a 6 kétszer szerepelnek A-ban. Ebben az esetben valamivel egyszerűbben is eljárhattunk volna. A B elemei lista helyett lehettek volna egész típusúak. A $B[j]$-t ekkor számlálóként használjuk: $B[j]$ a j kulcs A-beli előfordulásainak a száma. Az első fázis utáni helyzet így nézne ki:

B :

0	1	0	1	0	2	2	0	0	1

A rendezése pusztán e számok ismeretében is elvégezhető. Listákra akkor van szükségünk, ha a rendezés alapját jelentő kulcsok mellett még más információk is vannak az $A[i]$ elemekben; például a kulcs csak egy mező valamely rekord típusú elemben. Rendezéskor a többi mezőt is mozgatni kell.

Feladat: A ládarendezés érdekes speciális esete, amikor $m = n$, más szóval a rendezendő elemek és a lehetséges kulcsok száma egyenlő. Mutassuk meg, hogy a rendezés megoldható $O(n)$ költséggel *helyben* is, vagyis az A tömb mellett még legfeljebb állandó számú további tárcella felhasználásával.

2.3.2. Radix rendezés

Tegyük fel, hogy a rendezendő kulcsok összetettek, több komponensből állnak, és a $<$ reláció a komponensek rendezéséből származó lexikografikus rendezés. Legyenek mondjuk az U elemei k hosszúságú $t_1 \dots t_k$ alakú szavak, ahol a t_i komponens az L_i rendezett típusból való. Tegyük fel még, hogy az L_i típusnak összesen s_i eleme van. Legyen $t_1 \dots t_k$ és $u_1 \dots u_k$ két kulcs U-ból. A lexikografikus (szótárszerű) rendezés definíciója szerint $t_1 \dots t_k > u_1 \dots u_k$ éppen akkor teljesül, ha $t_i > u_i$ a legkisebb i indexre, amelyre $t_i \neq u_i$.

Példa: Legyen $(U, <)$ a *huszadik századi dátumok* összessége az időrendnek megfelelő rendezéssel. U természetes módon tekinthető három komponensből álló összetett típusnak. Az első az évszámokból álló típus:

$$L_1 = \{1900, 1901, \dots, 1999\}, \quad s_1 = 100.$$

A második a hónapokat magába foglaló típus:

$$L_2 = \{\textit{január}, \textit{február}, \dots, \textit{december}\}, \quad s_2 = 12.$$

A harmadik összetevőt a hónapok napjai adják:

$$L_3 = \{1, 2, \dots, 31\}, \quad s_3 = 31.$$

A dátumok rendezése éppen az L_i típusokból származó lexikografikus rendezés lesz.

A radix rendezés ilyen értelemben összetett kulcsok sorozatainak a rendezésére szolgál. A receptje lefegyverzően egyszerű: először rendezzük a sorozatot az utolsó, a k-adik komponensek szerint ládarendezéssel. Utána az eredményül kapott sorozatot ládarendezzük a $k-1$-edik komponens szerint, és így tovább, végül

az első komponens szerint. Arra kell ügyelni, hogy az elemeket mindig a ládájukat jelentő lista *végére* rakjuk. Más szóval, ha a j-edik ládarendezés előtt az X kulcs előbb van a sorozatban, mint az Y kulcs, és a kulcsok $k-j+1$-edik komponensei egyenlők, akkor X megelőzi Y-t a j-edik menet után is.

Az eljárás helyességét elég a $k = 2$ esetben vizsgálni; az érvelés könnyen általánosítható. Legyen $X = x_1x_2$ és $Y = y_1y_2$ két kulcs, és tegyük fel, hogy $X < Y$. Azt kell igazolni, hogy a végső sorozatban X előbb áll, mint Y. Ezt az utolsó, a második rendezés biztosítja, ha $x_1 < y_1$. Ha $x_1 = y_1$, akkor $X < Y$ miatt szükségképpen $x_2 < y_2$. Ekkor az első menet után X megelőzi Y-t. A kiegészítő szabályunk szerint ezen a viszonyon a második menet sem változtat; X a végső sorrendben is Y előtt lesz.

Példa: Nézzük a módszer működését a következő – dátum típusú – $A[1:5]$ tömbre mint bemenetre:

1969. jan. 18.	1969. jan. 1.	1955. dec. 18.	1955. jan. 18.	1918. dec. 18.

Itt $k = 3$. Az első fázisban a napok, utána a hónapok, végül az évek szerint rendezünk. Az egyes menetek utáni helyzeteket mutatja az alábbi ábra.

1969. jan. 1.	1969. jan. 18.	1955. dec. 18.	1955. jan. 18.	1918. dec. 18.

1969. jan. 1.	1969. jan. 18.	1955. jan. 18.	1955. dec. 18.	1918. dec. 18.

1918. dec. 18.	1955. jan. 18.	1955. dec. 18.	1969. jan. 1.	1969. jan. 18.

A radix rendezés költsége a k ládarendezés költségének az összege. A ládarendezésről mondottak szerint ez $O(kn + \sum_{i=1}^{k} s_i)$ elemi műveletet jelent. Ez bizonyos esetekben ismét jobbat adhat az összehasonlító módszerek idejénél. Nézzünk néhány példát.

Példák:

1. Tegyük fel, hogy k = állandó és $s_i \leq cn$ ($c > 0$ egy konstans). Ekkor a költség $O(kn + \sum_{i=1}^{k} cn) = O(k(c+1)n) = O(n)$. Ilyen helyzet például, amikor az $[1, n^k - 1]$ intervallumból való egészeket akarunk rendezni. Ezeket az egészeket k-jegyű számoknak vehetjük az n alapú számrendszerben. Az így felírt számok sorozatainak rendezésére a radix-módszer lineáris költségű megoldást ad.
2. $k = \log n$, $s_i = 2$. Ez a felállás, amikor $\log n$ hosszúságú bitsorozatokat rendezünk. Ekkor a radix rendezés költsége $O(n \log n + 2 \log n) = O(n \log n)$.

2.4. A Batcher-féle páros-páratlan összefésülés

Itt egy hatékony párhuzamos rendező algoritmust ismertetünk. A módszer az összefésüléses rendezés egy változata. Az abban szereplő összefésülő eljárást cseréljük ki egy gyors párhuzamos megoldással.

Az összefésülő módszert a jelölés egyszerűsítésére törekedve sorozatokkal mondjuk el. Legyenek $\mathcal{A} = a_1 < \ldots < a_l$ és $\mathcal{B} = b_1 < \ldots < b_m$ az input sorozatok. Célunk ezek összefésülése egyetlen növekvő $\mathcal{C} = c_1 < \ldots < c_{l+m}$ sorozattá.

A feladatot két egymástól függetlenül és párhuzamosan végezhető részre bontjuk. Az első brigád összefésüli az $a_1 < a_3 < a_5 < \ldots$ és $b_2 < b_4 < b_6 < \ldots$ sorozatokat az $u_1 < u_2 < u_3 < \ldots$ sorozattá. A második brigád összefésüli a két megmaradó részt, az $a_2 < a_4 < a_6 < \ldots$ és $b_1 < b_3 < b_5 < \ldots$ sorozatokat a $v_1 < v_2 < v_3 < \ldots$ sorozatba. Ezzel látszólag ugyanott vagyunk, ahonnan elindultunk. Van két növekvő sorozatunk, az $\{u_i\}$ és a $\{v_j\}$, amiket össze kellene fésülni. A következő állítás azt mondja, hogy a látszat csal. Az utóbbi két sorozat párhuzamosan *egyetlen lépésben* összefésülhető. A végeredményt jelentő sorozat c_j eleme egy összehasonlítással megkapható.

Állítás: *Érvényesek a* $c_{2i-1} = \min\{u_i, v_i\}$ *és* $c_{2i} = \max\{u_i, v_i\}$ *egyenlőségek* $(1 \leq i \leq (l+m)/2)$.

Bizonyítás: Először azt látjuk be, hogy tetszőleges $1 \leq k \leq (l+m)/2$ egészre a $\mathcal{C}_k := c_1, \ldots, c_{2k}$ részsorozatban ugyanannyi u_i van, mint v_j. Ezt úgy is mondhatjuk, hogy $\{c_1, \ldots, c_{2k}\} = \{u_1, \ldots, u_k\} \cup \{v_1, \ldots, v_k\}$. A $\mathcal{C}$ sorozat a növekvő $\mathcal{A}$ és $\mathcal{B}$ sorozatok összefésülése. Emiatt a $\mathcal{C}$ első $2k$ elemét az $\mathcal{A}$ sorozat első néhány eleme és a $\mathcal{B}$ sorozat első néhány eleme adja. Legyen mondjuk $\mathcal{C}_k = \{a_1, \ldots, a_s\} \cup \{b_1, \ldots, b_{2k-s}\}$. Ekkor a $\mathcal{C}_k$-ba eső u_i elemek közül $\lceil s/2 \rceil$ tartozik az $\mathcal{A}$, és $\lfloor \frac{2k-s}{2} \rfloor$ tartozik a $\mathcal{B}$ sorozatba. Ugyanígy a v_j elemek közül $\lfloor s/2 \rfloor$ tartozik az $\mathcal{A}$, és $\lceil \frac{2k-s}{2} \rceil$ tartozik a $\mathcal{B}$ sorozatba. A $\mathcal{C}_k$-ba kerülő u_i elemek száma $\lceil s/2 \rceil + \lfloor (2k-s)/2 \rfloor$, az ilyen v_j elemeké pedig $\lfloor s/2 \rfloor + \lceil (2k-s)/2 \rceil$. A kettő egyenlősége következik az egyszerű számolással igazolható

$$\lceil s/2 \rceil + \lfloor (2k-s)/2 \rfloor = \lfloor s/2 \rfloor + \lceil (2k-s)/2 \rceil$$

azonosságból.

Nézzük ezután az állítást. Az észrevétel szerint $\{c_1, \ldots, c_{2(i-1)}\} = \{u_1, \ldots, u_{i-1}\} \cup \{v_1, \ldots, v_{i-1}\}$ és $\{c_1, \ldots, c_{2i}\} = \{u_1, \ldots, u_i\} \cup \{v_1, \ldots, v_i\}$. Ezeket összevetve $\{c_{2i-1}, c_{2i}\} = \{u_i, v_i\}$, amiből az állítás $c_{2i-1} < c_{2i}$ miatt nyilvánvaló. □

A fentiekben vázolt páros-páratlan felbontással az összefésülés feladatát visszavezettük két párhuzamosan végezhető feleakkora méretű feladatra és ezen felül párhuzamosan állandó időben elvégezhető további munkára, a $\min\{u_i, v_i\}$ és $\max\{u_i, v_i\}$ elemek meghatározására. Ezzel egy rekurzív eljárást körvonalaztunk. A két brigád ugyanígy osztja tovább a problémát. Legyen az algoritmus neve PM.

Tegyük fel, hogy az $\mathcal{A}$ és $\mathcal{B}$ sorozatok összhossza n, és van $\lfloor n/2 \rfloor$ processzorunk. Valójában úgynevezett CE (compare-exchange) elemek is megteszik. Egy CE elem két kulcsot összehasonlít, és ha kell, felcseréli őket. A PM eljárás összehasonlításokban mért párhuzamos költségének maximumát adott n-re jelölje $T_p(n)$. A PM eljárás szerkezetéből rögvest adódik a $T_p(n) \leq T_p(\lceil n/2 \rceil)+1$ egyenlőtlenség. Az első tag a két félméretű részfeladat párhuzamos költségét, a második pedig az $\{u_i\}$ és $\{v_j\}$ sorozatok összefésülését könyveli. Ebből az összefésüléses rendezés elemzésénél alkalmazott visszahelyettesítős érveléssel adódik, hogy $T_p(n) \leq \lceil \log_2 n \rceil$.

A PM eljárást használhatjuk az összefésüléses rendezésben. A definíció így módosul:

$$\text{MSORT}(A[1:n]) := \text{PM}(\text{MSORT}(A[1:\lceil n/2 \rceil]), \text{MSORT}(A[\lceil n/2 \rceil + 1 : n])).$$

Jelölje $t_p(n)$ a módszer párhuzamos költségét n elemű bemenetre. A tömb alsó, illetve felső felének a rendezése párhuzamosan végezhető. Ezért $t_p(n) \leq t_p(\lceil n/2 \rceil) + \lceil \log_2 n \rceil$. A második tag a párhuzamos összefésülés ideje. Ezt ismét a visszahelyettesítő módszerrel fejthetjük ki:

$$t_p(n) \leq 1 + 2 + \cdots + (\lceil \log_2 n \rceil - 1) + \lceil \log_2 n \rceil = \frac{(\lceil \log_2 n \rceil + 1)(\lceil \log_2 n \rceil)}{2}.$$

A Batcher-rendező párhuzamos költsége eszerint $t_p(n) = O(\log^2 n)$.

2.5. Külső tárak tartalmának rendezése

Az eddig tárgyalt rendező eljárások kapcsán hallgatólagosan feltettük, hogy az adataink a számítások menete során végig egy nagy elérési sebességű *belső memóriában* vannak. A feltételezés ott bújt meg az eddigi elemzésekben, hogy nem szenteltünk túl nagy figyelmet az adatok mozgatásához szükséges időnek. Bizonyos esetekben csak az összehasonlításokat számoltuk. Máskor figyeltük ugyan a cserék és más mozgatások idejét, de ezeket az összehasonlításokéval egyező nagyságrendű tényezőként kezeltük. Ez a szemlélet jól működik a belső memóriás számítások esetén. Az így nyert becslések megfelelnek a tényleges számítások

időviszonyainak. *Külső tárakon* (mágneslemezek, dobok, szalagok) elhelyezkedő adatok kezelésére viszont a feltételezés általában nem alkalmazható.

Egy belső memóriában levő szó elérése és olvasása legfeljebb néhány mikroszekundumot jelent. A mágneslemezek elérési ideje viszont az ezredmásodperces nagyságrendben van, diszketteknél pedig a másodpercet is meghaladhatja. A külső tárak elérési ideje tehát több ezerszerese is lehet a belsőkének. Külső tárakon levő adatok kezelésénél ezért komoly figyelmet kell fordítanunk az adatelérés/mozgatás költségeire.

A külső tárak másik fontos vonása, hogy egy írás/olvasás során nem csak egy szót mozgatunk, hanem egy nagyobb területet. Használható modellt kapunk az ilyen tárak működésére, ha úgy képzeljük, hogy az állományok az elérhetőség szempontjából folytonosan egymás után következő *lapokon* helyezkednek el. Egy lap egyetlen I/O művelettel kezelhető (olvasható, írható) terület a táron. A szokásos lapméret $512 \cdot k$ byte valamely kicsi k-val. A folytonosság bizonyos esetekben tényleges fizikai folytonosságot jelent, máskor pedig magasabb szintű (pl. az operációs rendszer által biztosított) eljárások adnak módot a következő lap értelmezésére. Egy lapon általában több rekord is elfér. A lapok beolvasása, kiírása igen költséges a belső memóriában végzett elemi lépésekhez képest. Átlagos alkalmazásoknál az I/O műveletek időigénye a futási idő döntő része. Ezért a külső tárat jelentős mértékben használó programok esetén alapvető hatékonysági szempont a *lapelérések számának minimalizálása.* A korábban tárgyalt rendezési módszerek, ötletek általában nem adnak jó eredményt, ha a rendezendő adatok külső tárban vannak. Tulajdonképpen egyetlen kivétel az összefésüléses rendezés.

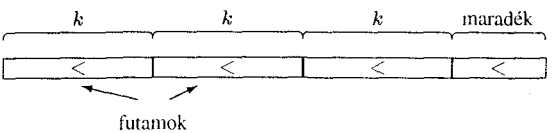
2.5.1. Összefésüléses rendezés külső tárakon

A bevezetőben említetteknek megfelelően az algoritmusok értékeléséhez egyetlen költségtényezőként a *lapelérések számát* használjuk. A módszerek, amiket ismertetünk, működnek mágnesszalagok esetén is, mert az állományokat lapjaik sorrendjében (szokásos szakkifejezéssel: *szekvenciálisan*) használjuk. Ez annyit tesz, hogy a bemeneti állományokat szekvenciálisan olvassuk, és így végezzük a kiírásokat is a kimeneti állományokba.

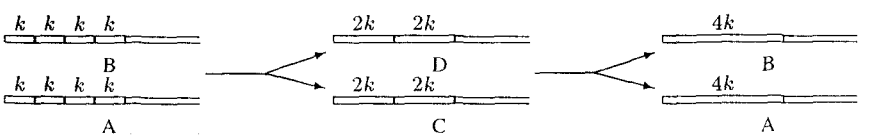
A célunk egy a háttértáron levő n lapból álló állomány rendezése valamilyen adott kulcs szerint (pl. személyek adatait tartalmazó rekordoké a személyi szám szerint).

Definíció: *Egy állomány k szomszédos lapjából álló rész k hosszú futam, ha a rekordok ebben a részben a kulcs szerint (nem csökkenően) rendezetten helyezkednek el.*

Először alkalmas kis k mellett (ha más nem, $k = 1$ mindig lehetséges) k hosszúságú kezdőfutamokat hozunk létre; ezeket egyenletesen elhelyezzük az A és B állományokba. Ez többnyire úgy történik, hogy egyszer végigolvassuk a bemenő állományt. A memóriában k lapnyi futamokat alakítunk ki; amint egy futamot befejeztünk, azt kiírjuk az A és B állományok egyikébe. Ennek a műveletnek a költsége $2n$ lapelérés. A futamok közül az utolsó esetleg k-nál kevesebb lapot tartalmaz.



Ezután az A és B állományok elejéről indulva sorban összefésüljük a futamokat – mindig egyet A-ból és egyet B-ből véve – $2k$ hosszúakká; ezeket egyenletesen elhelyezzük a C és D állományokba. Egy ilyen menet költsége $2n$ I/O művelet. Ezután C és D futamait fésüljük ugyanígy össze $4k$ hosszú futamokká A-ba és B-be, és így tovább.



Mint a kezdeti helyzetnél, később is előfordulhat, hogy a menetben utoljára kapott futam rövidebb a többinél. Ha az i-edik összefésülő menet után még egynél több futam van, akkor ezek közül az első hossza legalább 2^i. Ez pedig csak addig lehetséges, amíg $2^i < n$ teljesül. Ha a j-edik az utolsó menet, akkor a $j-1$. menet után még legalább két futam van, tehát $2^{j-1} < n$, amiből logaritmusra térve $j \leq \lceil \log_2 n \rceil$. A szükséges menetek száma nem több, mint $\lceil \log_2 n \rceil$. Az összes lapelérések száma ennek megfelelően legfeljebb $2n(\lceil \log_2 n \rceil + 1)$. Ezt az algoritmust, illetve változatait, finomításait használják leggyakrabban külső rendezésre.

Két gyorsítási lehetőség:

1. Érdemes minél nagyobb kezdő futamokat létrehozni, más szóval nagy k értékkel dolgozni. Az előbbi gondolatmenet pontosítható: ha k hosszúságú futamokkal kezdjük a fésülést, akkor $k2^{j-1} < n$ is teljesül, amiből a menetek számára $j \leq \lceil \log_2(n/k) \rceil$ adódik.

2. m-felé fésülés, ahol $m > 2$. Ekkor minden menetben m input és m output állományt használunk, és egyszerre m futamot fésülünk össze egyetlen futamba. Egy menet után a futamok mérete m-szereződik. A menetek száma legfeljebb $j = \lceil \log_m n \rceil = \lceil (\log_2 n)/\log_2 m \rceil$. Itt még további $2m$-szeres gyorsítás érhető el, ha van $2m$ párhuzamosan használható adatcsatornánk (minden állományhoz egy). Ekkor ugyanis egy I/O művelet ideje alatt olvashatunk egy-egy lapot az input-állományokból, és ugyanekkor írhatunk is egy-egy lapot a kimenő állományokba. Az m felé fésülésnél ügyelni kell arra, hogy m növekedtével a futamok összefésülésének költsége is nő. Egyre több idő kell, és egyre nagyobb területet használunk a belső memóriában.

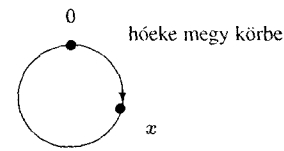
Kezdeti futamok létrehozása kupaccal

Itt egy érdekes algoritmust ismertetünk, amivel átlagos értelemben viszonylag hosszú kezdőfutamokat lehet építeni. A futamok hossza változhat, így előfordulhatnak rövid futamok is.

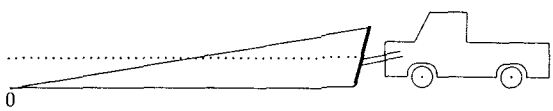
A belső memóriában lefoglalunk egy területet, amin K kulcs elfér. Mint látni fogjuk, K értékét érdemes minél nagyobbnak választani. A lefoglalt területen két – kulcsokat tartalmazó – kupacot üzemeltetünk. A két kupacban összesen legfeljebb K kulcsot tartunk. Az egyik lesz az aktív kupac a, a másik az alvó kupac A. Az aktív kupacban tartjuk az éppen kiírandó futam kulcsait, az alvóban pedig a következő futam kulcsait gyűjtjük. Kezdetben mindkettő üres. Először a-t feltöltjük a rendezendő állományból K kulccsal, A pedig üres.

Ezután amíg a nem üres, ismételjük a következő lépést. Az s =MINTÖR(a) kulcsú rekordot beírjuk az aktuális futamba. Ezután vesszük az input állomány következő rekordját. Legyen ennek a kulcsa t. Ha $t < s$, akkor a rekord már nyilván nem tehető az aktuális futam végére, ezért t-t A-ba szúrjuk be. Ha $t \geq s$, akkor a t kulcs az a-ba kerül, hiszen ekkor a rekord még elhelyezhető valahol később az aktuális futamban. Ha a kiürül, akkor a futam befejeződik. A két kupac szerepet cserél, A lesz aktív, és a megy aludni. Új futam kezdődik, amit A-ból töltünk az előzőek szerint.

Megmutatható, hogy – rekordokban mérve – az átlagos futamhossz $2K$ lesz, ha a kulcsok alkalmas értelemben egyenletesen érkeznek. Ennek a jelenségnek kissé regényes magyarázatát adja a következő *hóeke modell* (E.F. Moore 1961). Egy kör alakú pályán hóeke megy körbe.



A hó egyenletesen hull a pályára, a hóeke az előtte levő hómagassággal fordítottan arányos sebességgel halad. Megmutatható, hogy van a rendszernek stabil állapota, amihez a kezdő helyzetből konvergál. Ekkor minden pillanatban ugyanaz a P hómennyiség van a pályán, a hóeke sebessége állandó és a hó magassága lineárisan függ az ekétől való (pálya mentén mért) távolságtól. Ebből következik, hogy egy körben a hóeke $2P$ hómennyiséget takarít el.



A modellben a hó játssza a kulcsok szerepét. Az aktív kupac az eke jelenlegi helyzetétől az induló helyzetig menő ív, a maradék az alvó kupac. Így az egy körben eltakarított hó felel meg egy futamnak.

Minél előbbre tart a hóeke a pályán – vagyis minél nagyobb az aktív kupac minimális eleme –, annál nagyobb az esélye, hogy egy érkező hópehely a hóeke elé, és nem mögé esik. Másként mondva: a következő kulcs egyre növekvő eséllyel kerül az alvó kupacba.

Többfázisú (polifázisú) összefésülés

Vannak olyan k-rétű összefésülő sémák, melyek csak $k+1$ állományt használnak. A k input állományt egyetlen kimenő állományba fésüljük. Ha valamelyik input állomány üres lesz, akkor megállunk, és ez veszi át az output állomány szerepét. Itt csak a $k = 2$ esetre vetünk egy rövid pillantást. Ekkor 3 állományunk van, és minden fázisban kettő tartalmát fésüljük a harmadikra.

Nézzünk először egy példát. Legyen a három állomány f_1, f_2, f_3. Tegyük fel, hogy f_1 egy, f_2 pedig 5 ugyanolyan (mondjuk 1) hosszúságú futamból áll, f_3 üres. A következő táblázat szemlélteti a helyzetet. Az oszlopok adatai mutatják az összefésülő menetek utáni állapotot, az első oszlop a kezdő állapot. Egy bejegyzés első száma a futamok száma az állományban, a zárójelben levő érték pedig a futamhossz.

f_1	1(1)	$\emptyset$	1(3)	$\emptyset$	1(5)	$\emptyset$
f_2	5(1)	4(1)	3(1)	2(1)	1(1)	$\emptyset$
f_3	$\emptyset$	1(2)	$\emptyset$	1(4)	$\emptyset$	1(6)

Érezhetően túl sok mozgatást végzünk. Nem szerencsés a rekordok eloszlása a két állományban. Megmutatható, hogy ez a módszer akkor lesz a leggyorsabb, ha a futamok száma a két input állományban két szomszédos *Fibonacci-szám*. Először – emlékeztetőül – néhány hasznos tény a Fibonacci-számokról.

A Fibonacci-számokat a következő rekurzióval definiálhatjuk:

$$F_0 = 0, F_1 = 1, \quad F_{i+1} = F_i + F_{i-1}, \text{ ha } i \geq 1.$$

Hasznos az alábbi általános alak (indukcióval igazolható):

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right).$$

Innen látjuk, hogy F_n az $\frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n$ számhoz legközelebbi egész.
Legyen $\phi = \frac{1+\sqrt{5}}{2}$ (az aranymetszés aránya).
A $\phi^2 = \phi + 1$ egyenlőségből adódik $n \geq 2$ esetén, hogy

$$\phi^{n-2} \leq F_n \leq \phi^{n-1}.$$

Ezután nézzük, miként alakul a futamok száma, ha a kezdő futamszámok F_{n-1} és F_n. A táblázatban most nem tüntetjük fel a futamok hosszát.

f_1	F_{n-1}	$\emptyset$	F_{n-2}	F_{n-4}	$\ldots$
f_2	F_n	F_{n-2}	$\emptyset$	F_{n-3}	$\ldots$
f_3	$\emptyset$	F_{n-1}	F_{n-3}	$\emptyset$	$\ldots$

Kezdetben a futamok száma összesen $N = F_{n+1}$. Ebből $\phi^{n-1} \leq N$ és $n-1 \leq \log_\phi N = \log_2 N / \log_2 \phi \approx 1.44 \log_2 N$ adódik. A szükséges menetek száma ezért legfeljebb $1.44 \log_2 N + 1$. Ez valamivel rosszabb ugyan, mint a 4 állományt használó összefésülésé, de még mindig elég jó.