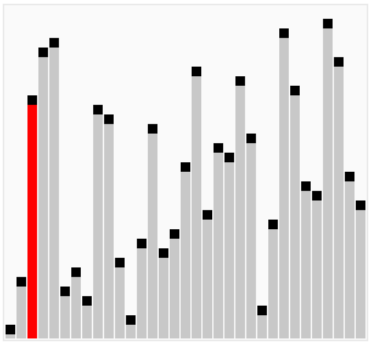
Rendezési algoritmusok

Forrás: wikipedia.hu

Buborékrendezés



Az algoritmus újra és újra végigiterál a listán, összehasonlítja a lista szomszédos elemeit, és ha azok az elvárt rendezéshez képest fordítva vannak, akkor megcseréli őket. Első menetben a lista elejéről indul és a végéig megy. Ennek az első menetnek az eredményeként a legnagyobb elem (mint egy buborék) felszáll a lista végére. Így a következő menetben már elegendő az utolsó előtti elemig elvégezni a szomszédos elemek összehasonlítását és cseréjét. Az ezután következő menetben a lista utolsó két eleme lesz a helyén és így tovább...

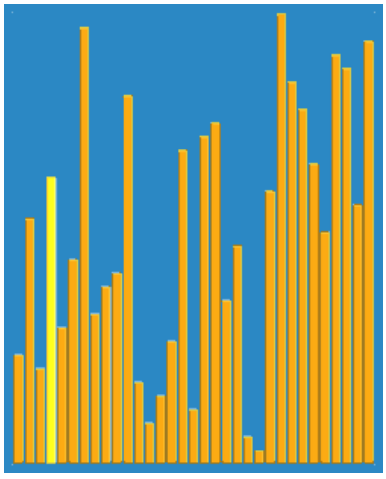
Az algoritmus két egymásba ágyazott ciklusból áll. A tömb első elemének indexe az 1, elemeinek száma pedig n. Az elemek itt számok, és a reláció a > (nagyobb).

```
FUNC bubblesort(lista)
   FOR i = n TO 0
      FOR j = 0 TO i-1
         IF lista[j] > lista[j+1] THEN
            SWAP lista[j] AND lista[j+1]
```

A futás befejezése után a tömb 1-es indexű eleme lesz a legkisebb és az n indexű a legnagyobb.

Az algoritmus onnan kapta a nevét, hogy először a legnagyobb elem „száll fel” a tömb utolsó helyére, utána a második legnagyobb az azt követő helyre, és így tovább, mint ahogy a buborékok szállnak felfelé egy pohárban.

Kertitörpe rendezés

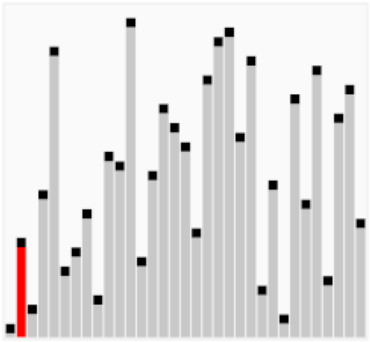


Hasonlít a beszúrásos rendezésre, azonban az elemek a buborékrendezésre emlékeztető módon, sorozatos cserék után kerülnek a helyükre. Elve egyszerű, nem használ beágyazott ciklusokat. Várható végrehajtási ideje erősen függ attól, hogy az elemek eredetileg mennyire rendezettek.

Az algoritmus megkeresi az első olyan helyet, ahol két egymást követő elem rossz sorrendben van, és megcseréli őket. Ha egy ilyen csere után rossz sorrend keletkezik, az csak közvetlenül a legutolsó csere előtt lehet, így ezt is ellenőrizzük. Csere után ismét ellenőrzés következik, ezért a cserék addig folytatódnak, amíg az elem a megfelelő helyre nem kerül.

```
FUNC gnome_sort(lista):
   i = 0                              # ezt vizsgáljuk éppen
   n = HOSSZ(lista)                   # ennyi elemű a lista
   WHILE i < n-1:                     # míg a lista végére nem érünk
      IF lista[i] <= lista[i+1] THEN
         i += 1                       # ha jó a sorrend: index növelése
      ELSE
         SWAP lista [i], lista[i+1]   # ha rossz a sorrend, cserélünk, és
         IF i > 0 THEN i -= 1         # csökkentjük az indexet
```

Koktél rendezés



A koktélrendezés (cocktail sort) a buborékrendezés tökéletesített változata, mely két irányból megy végig a tömbön. Minimálisan bonyolultabb a buborékrendezésnél, ugyanakkor kiküszöböli annak egyik problémáját, miszerint a nagy elemek gyorsan felemelkednek a helyükre (innen a „buborék” név), de a rossz helyen lévő kicsi elemek csak lassan süllyednek a helyükre.

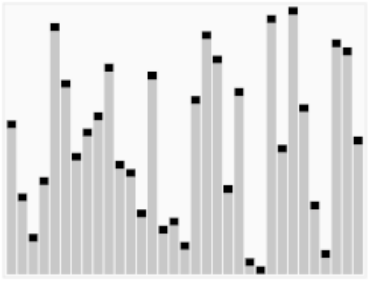
A legrosszabb esetben itt is $O(n^2)$ műveletre van szükség, de ha a lista majdnem rendezett az elején, a műveletek száma közelebb van $O(n)$-hez mint a buborékrendezés esetén.

```
FUNC cocktail_sort(lista):
    start = 0
    end = HOSSZ (lista) - 1

    WHILE start < end
        FOR i = start TO end
            IF lista[i] > lista[i + 1]
                SWAP lista[i] AND lista[i + 1]
        end -= 1

        FOR i = end DOWNTO start
            IF lista[i] < lista[i - 1]
                SWAP lista[i], lista[i - 1]
        start += 1
    RETURN lista
```

Gyorsrendezés



A gyorsrendezés oszd meg és uralkodj elven működik: a rendezendő számok listáját két részre bontja, majd ezeket a részeket rekurzívan, gyorsrendezéssel rendezi. A felbontáshoz kiválaszt egy támpontnak nevezett elemet (más néven pivot, főelem vagy vezérelem), és particionálja a listát: a támpontnál kisebb elemeket eléje, a nagyobbakat mögéje mozgatja. Teljes indukcióval könnyen belátható, hogy ez az algoritmus helyesen működik.

Az algoritmus pszeudokódja:

```
# particionáló függvény
FUNC partition(arr, low, high):
    store_index = (low-1)           # itt tároljuk az utolsó pivot-nál kisebb elem helyét
    pivot_value = arr[high]         # a pivot elem a legutolsó elem lesz
    FOR i = low TO high
        IF arr[i] <= pivot_value THEN  # ha a vizsgált kisebb, mint a pivot, akkor
            store_index += 1        # találtunk egy kisebbet, ezért növekszik az index
            SWAP arr[store_index] and arr[i]
                                    # a vizsgált elemet betesszük az utolsó helyre a "kisebbek" közt
    SWAP arr[store_index+1] and arr[high]
                                    # végül a pivot elemet is helyére tesszük
    RETURN store_index + 1

# quicksort algoritmus
FUNC quicksort(arr, low, high):
    IF low < high:                          # ha egynél több elemet kell rendezni
        pivot_index = partition(arr, low, high)  # particionáljuk a listát

        quicksort(arr, low, pivot_index - 1)     # rendezzük a kisebbeket
```

```
        quicksort(arr, pivot_ index + 1, high)   # rendezzük a nagyobbakat
```

Egy másik megközelítés a particionálás helyett több tömböt használ. Így az algoritmusunk kicsit egyszerűbb, viszont több adatmozgatást igényel:

```
FUNC quicksort(lista)
    n = HOSSZ(lista)
    IF n ≤ 1 THEN
        RETURN lista
    pivot = lista egy eleme
    FOR i = 1 TO n
        IF lista[i] < pivot THEN APPEND lista[i] TO less
        IF lista[i] = pivot THEN APPEND lista[i] TO equal
        IF lista[i] > pivot THEN APPEND lista[i] TO greater
    RETURN CONCATENATE(quicksort(less), equal, quicksort(greater))
```