Sorteerimisalgoritmi tõhusust võib mõõta mitme näitaja alusel: tööks kuluv aeg (eraldi keskmisel ja halvimal võimalikul juhul), vajaminev mälu (jälle keskmiselt ja maksimaalselt). Siinkohal tasub ära märkida ka seda, et sageli järjestatakse tegelikult mitte arve, vaid suuremaid kirjeid ja võtmeks on ainult mingi osa kirjest. Sel juhul on kahe kirje vahetamine töömahukam operatsioon kui nende võrdlemine. Niisuguste olukordade tarvis uuritakse sorteerimismeetodite hindamisel eraldi võrdlemiste ja vahetuste arvu. Tegelikult on enamasti võimalik vahetusoperatsiooni hinda vähendada (näiteks kasutades viitu kirjetele ja järjestades neid – viida omistamine ei ole kuigi kallis), aga selleks on vaja lisamälu.

Lisaks kvantitatiivsetele omadustele erinevad algoritmid üksteisest ka kvalitatiivselt. Nii mõndagi neist saab tõhusalt realiseerida ainult kindlat tüüpi andmestruktuuride abil. Näiteks mõni algoritm võib olla väga tõhus järjendite sorteerimiseks, kuid ebasobiv ahelate jaoks. Kaks suurt gruppi moodustavad sisemised ja välised sorteerimismeetodid. Sisemised meetodid eeldavad, et sorteerimise ajal on kõik andmed operatiivmälus ja seetõttu ei sobi nad suuremate andmehulkade järjestamiseks. Välised aga lepivad sellega, et korraga on mälus ainult osa andmetest ja saavad seega sorteerida praktiliselt piiramatu suurusega andmehulki. Oluline omadus on ka algoritmi stabiilsus. Stabiilseks nimetatakse sorteerimisalgoritmi, mis ei muuda võrdsete võtmetega elementide esialgset järjestust. Ebastabiilse algoritmi saab stabiilseks muuta, kui märkida iga kirje juurde tema indeks esialgses järjestuses ja kasutada seda viimase võrdlemiskriteeriumina, kuid selleks kulub täiendavalt aega ja mälu.

Ruutkeerukusega

```
procedure vahetusmeetod (var a : massiiv; n : integer);
var i, j : integer;
begin
  for i := 1 to n - 1 do           { vaatleme esimesest eelviimaseni }
    for j := i + 1 to n do       { võrdleme kõigi tagapool olevatega }
      if a[i] > a[j] then                  { kui vales järjekorras }
        vaheta (a[i], a[j]);                       { siis vahetame }
end;
```

- väga lihtne
- n(n-1)/2 võrdlust
- halvimal juhul ja keskmiselt $O(n^2)$ vahetust
- ei ole stabiilne

```
procedure valikumeetod (var a : massiiv; n : integer);
var i, j, k : integer;
begin
  for i := 1 to n - 1 do begin     { vaatleme esimesest eelviimaseni }
    k := i;                         { jätame meelde jooksva elemendi }
    for j := i + 1 to n do       { võrdleme kõigi tagapool olevatega }
      if a[k] > a[j] then             { kui on meelesolevast väiksem }
        k := j;                      { siis jätame väiksema meelde }
    if k > i then                   { kui tagapool oli mõni väiksem }
      vaheta (a[i], a[k]);         { vahetame jooksva meelesolevaga }
  end;
end;
```

- n(n-1)/2 võrdlust
- n-1 vahetust
- ei ole stabiilne

```
procedure pistemeetod (var a : massiiv; n : integer);
var i, j : integer;
begin
  for i := 2 to n do begin             { vaatleme teisest viimaseni }
    j := i - 1;           { alustame jooksvale eelnevast elemendist }
    repeat
      if a[j] > a[j + 1] then begin       { kui on eelnevast väiksem }
        vaheta (a[j], a[j + 1]);                    { siis vahetame }
        j := j - 1;                  { ja jätkame sellele eelnevaga }
      end else
        j := 0;                     { muidu on oma kohal ja lõpetame }
    until j = 0;
  end;
end;
```

- halvimal juhul n(n-1)/2 võrdlust ja vahetust
- keskmiselt (n+2)(n-1)/4 võrdlust ja n(n-1)/4 vahetust
- sorteeritud jadal n-1 võrdlust ja 0 vahetust
- stabiilne

```
procedure pistemeetod2 (var a : massiiv; n : integer);
var i, j, k : integer;
begin
  for i := 2 to n do begin             { vaatleme teisest viimaseni }
    k := a[i];            { jätame uue, "pistetava" väärtuse meelde }
    j := i - 1;      { alustame sorteeritud osa viimasest elemendist }
    repeat                      { otsime uuele koha ning teeme ruumi: }
      if a[j] > k then begin                  { kui on uuest suurem }
        a[j+1] := a[j];                  { siis nihutame tahapoole }
        j := j - 1;                         { ja vaatleme eelnevat }
      end else
        j := 0;              { muidu on uuele koht leitud, lõpetame }
    until j = 0;
    a[j+1] := k;               { pistame uue väärtuse õigesse kohta }
  end;
end;
```

- vahetuse (järjendis kahe elemendi ülekirjutamine) asemel üks järjendisse kirjutamine

```
procedure loendamismeetod (var a : massiiv; n : integer);
var b : massiiv; i, j : integer;
```

```
begin
  for i := 1 to n do begin                  { vaatleme kõiki elemente }
    b[i] := 0;                        { esialgu pole väiksemaid leidnud }
    for j := 1 to i do                     { võrdleme eespool olevatega }
      if a[j] <= a[i] then                              { kui pole suurem }
        b[i] := b[i] + 1;                        { siis jääb ettepoole }
    for j := i + 1 to n do                { võrdleme tagapool olevatega }
      if a[j] < a[i] then                                { kui on väiksem }
        b[i] := b[i] + 1;                       { siis läheb ettepoole }
  end;
  i := 1;                                             { alustame esimesest }
  while i < n do                                 { ja liigume eelviimaseni }
    if b[i] <> i then begin                           { kui pole oma kohal }
      vaheta (a[b[i]], a[i]);                        { vahetame oma kohale }
      vaheta (b[b[i]], b[i]);                       { vahetame ka indeksid }
    end else
      i := i + 1;                                  { muidu vaatame järgmist }
end;
```

- kaheosaline: 1) leiame igale elemendile õige koha, 2) paigutame elemendid ümber
- näitab, kuidas saab vahetuste arvu vähendada kogu järjendist koopiat tegemata
- näites teostatud kujul küll valikumeetodist viletsam, kuid osana 1) võime indeksite järjestamiseks kasutada suvalist sorteerimisalgoritmi
- kui vahetustest lahti saada nagu `pistemeetod2`-s, teeb vähima võimaliku arvu järjendisse kirjutamisi, mälukulule aga lisandub ühe elemendi suurus.

```
procedure mullimeetod (var a : massiiv; n : integer);
var i : integer; v : boolean;
begin
  repeat
    v := false;                                  { seni pole vahetusi olnud }
    for i := 1 to n - 1 do                { vaatleme esimesest eelviimaseni }
      if a[i] > a[i + 1] then begin              { kui on järgnevast suurem }
        vaheta (a[i], a[i + 1]);                            { siis vahetame }
        v := true;                               { ja registreerime vahetuse }
      end;
  until not v;        { kui ei olnud vahetusi, on massiiv sorteeritud }
end;
```

- suured arvud liiguvad lõpu poole nagu mullid pinnale
- stabiilne
- halvimal juhul n läbimist, n(n-1) võrdlust/vahetust: kui vähim arv järjendi lõpus, liigub ta iga läbimisega vaid ühe võrra
- parandusmõte: läbime järjendit vaheldumisi mõlemas suunas — saame **raputusmeetodi**, ikka $O(n^2)$

O(n log n) keerukusega

```
procedure shellimeetod (var a : massiiv; n : integer);
var s, k, i : integer; v : boolean;
begin
  s := 1;                                          { alustame sammust üks }
  while s < n do                             { leiame suurima võimaliku sammu }
    s := 3 * s + 1;
  while s > 1 do begin                     { kordame järjest kahaneva sammuga }
    s := s div 3;                                    { arvutame uue sammu }
    for k := 1 to s do                          { tekib s sõltumatut gruppi }
      repeat                              { sorteerime iga grupi mullimeetodil }
        v := false;
        for i := 1 to (n - k) div s do
          if a[k + (i - 1) * s] > a[k + i * s] then begin
            vaheta (a[k + (i - 1) * s], a[k + i * s]);
            v := true;
          end;
      until not v;
  end;
end;
```

- mulli- ja raputusmeetodi ruutkeerukuse peapõhjuseks on, et alati võrreldakse ja vahetatakse kõrvutiolevaid elemente, seega ei liigu ühegi vahetuse käigus ükski element rohkem kui ühe koha võrra
- Shelli meetod (Donald Shell, 1959) — algul rakendame mullimeetodit suure sammuga, hiljem väiksemaga (mullimeetodi asemel võib kasutada ka nt. pistemeetodit)
- keskmine keerukus $O(n \cdot \log(n))$, halvimal juhul suurem (kui palju suurem, sõltub sammupikkuste jada valikust)
- ebastabiilne

```
procedure kiirmeetod (var a : massiiv; n : integer);
var k : element; { järjendi elemendituup võib olla arv, aga ka nt. sõne }
  procedure kiirmeetodisisu (v, p : integer);
  var i, j : integer;
  begin
    if v < p then begin                  { kui üldse on midagi sorteerida }
      k := a[(v + p) div 2];           { jätame meelde keskmise elemendi }
      i := v; j := p;                                  { alustame otstest }
      repeat
        while a[i] < k do            { kuni vasakul keskmisest väiksemad }
          i := i + 1;                             { nihutame järge paremale }
        while k < a[j] do             { kuni paremal keskmisest suuremad }
          j := j - 1;                             { nihutame järge vasakule }
        if i <= j then begin                { kui pole üksteisest möödunud }
          vaheta (a[i], a[j]);                               { siis vahetame }
          i := i + 1; j := j - 1;           { ja nihutame järgesid edasi }
        end;
      until i > j;                      { kuni järjed üksteisest mööduvad }
      kiirmeetodisisu (v, j);               { sorteerime väikesed omaette }
```

- C. A. R. Hoare, 1960
- keskmiselt võrdlusi ja vahetusi $O(n \cdot \log(n))$, lisamälu $O(\log n)$
- halvimal juhul ajakulu $O(n^2)$, mälukulu $O(n)$
- töödeldes pikemat poolt rekursiivse väljakutse asemel samas meetodis, saame ka halvima juhu mälukulu $O(\log n)$
- väikese n korral võib kombineerida $O(n^2)$ meetodiga
- praktikas üks kiireimaid
- ebastabiilne
- C-s `qsort`, Javas `java.util.Arrays.sort` (primitiivtüübi

```
      kiirmeetodisisu (i, p);                       { ja suured omaette }
    end;
  end;
begin
  kiirmeetodisisu (1, n);
end;
```

järjendite jaoks), Turbo Pascali ning Free Pascaliga on kaasas kiirmeetodi näide (Free Pascalis fail `demo/text/qsort.pp`)

```
procedure kuhjameetod (var a : massiiv; n : integer);
var i, j, k : integer;
begin
  for i := 2 to n do begin             { lisame kuhja teisest viimaseni }
    j := i;
    while j > 1 do                              { kuni ei ole kuhja tipp }
      if a[j] > a[j div 2] then begin          { kui on ülemusest suurem }
        vaheta (a[j], a[j div 2]);                       { siis vahetame }
        j := j div 2;                    { ja jätkame ülemuse ülemusega }
      end else
        j := 1;                          { muidu on oma kohal ja lõpetame }
  end;
  for i := n downto 2 do begin       { massiivi moodustame tagant alates }
    vaheta (a[1], a[i]);                          { kuhja tipp oma kohale }
    j := 1;                                      { hakkame kuhja taastama }
    while j < i do begin
      k := j;                            { jätame meelde jooksva elemendi }
      if 2 * j < i then                             { kui leidub vasak alluv }
        if a[k] < a[2 * j] then          { ja see on meelesolevast suurem }
          k := 2 * j;                        { siis jätame suurema meelde }
      if 2 * j + 1 < i then                         { kui leidub parem alluv }
        if a[k] < a[2 * j + 1] then      { ja see on meelesolevast suurem }
          k := 2 * j + 1;                    { siis jätame suurema meelde }
      if k > j then begin                    { kui jooksev ei ole suurim }
        vaheta (a[k], a[j]);                   { siis vahetame suurimaga }
        j := k;                         { ja jätkame jooksva uuest kohast }
      end else
        j := i;                        { muidu on jooksev oma õigel kohal }
    end;
  end;
end;
```

- (kahend)kuhi — (kahend)puu, kus iga tipp on oma alamatest suurem. Kahendkuhja järjendis hoidmine: kui juurtipu indeksiks 1, siis iga tipu k alamateks tipud 2*k ja 2*k+1
- võrdlusi ja vahetusi alati O(n·log(n)), lisamälu O(1)
- kuhjameetod on ebastabiilne
- kuhjameetod ei sobi väliseks sorteerimiseks
- STL-is `make_heap`, `pop_heap`, `push_heap`, `sort_heap`, Javas `java.util.PriorityQueue`
- **sisekaemusmeetod** (ingl. k. *introspective sort*) (David Musser, 1997) — kiirmeetodi ja kuhjameetodi ristsugutis, mis kasutab halbadel juhtudel kuhjameetodit ja muidu kiirmeetodit. Introsort on keskmiselt sama kiire kui kiirmeetod, kuid ka halvimal juhul ajakuluga O(n log n) — STL-is meetod `sort`

```
procedure yhildusmeetod (var a : massiiv; n : integer);
var b : massiiv; k, i, l1, l2, j1, j2 : integer;
begin
  k := 1;                                 { alustame lõikudest pikkusega 1 }
  while k < n do begin          { kuni lõigupikkus on väiksem jadapikkusest }
    i := 1;                               { esimene vaba koht abimassiivis }
    l2 := 1;                          { esimene sorteerimata lähtemassiivis }
    while l2 <= n do begin                 { kuni on sorteerimata elemente }
      j1 := l2; l1 := j1 + k;               { järgmise lõigu algus ja lõpp }
      if l1 > n then l1 := n + 1;          { kui lõpp läks massiivist välja }
      j2 := l1; l2 := j2 + k;             { ülejärgmise lõigu algus ja lõpp }
      if l2 > n then l2 := n + 1;          { kui lõpp läks massiivist välja }
      while (j1 < l1) and (j2 < l2) do    { kuni mõlemas lõigus elemente }
        if a[j1] < a[j2] then begin         { kui esimeses lõigus väiksem }
          b[i] := a[j1]; j1 := j1 + 1; i := i + 1;     { see abimassiivi }
        end else begin                                          { muidu }
          b[i] := a[j2]; j2 := j2 + 1; i := i + 1;   { teine abimassiivi }
        end;
      while (j1 < l1) do begin                  { kui esimeses lõigus jääk }
        b[i] := a[j1]; j1 := j1 + 1; i := i + 1;      { siis abimassiivi }
      end;
      while (j2 < l2) do begin                   { kui teises lõigus jääk }
        b[i] := a[j2]; j2 := j2 + 1; i := i + 1;      { siis abimassiivi }
      end;
    end;
    for i := 1 to n do                  { abimassiiv põhimassiivi tagasi }
      a[i] := b[i];
    k := 2 * k;                                        { uus lõigupikkus }
  end;
end;
```

- võrdlusi ja vahetusi alati O(n·log(n)), lisamälu O(n)
- stabiilne
- sobib väliseks sorteerimiseks
- sobib ka ahelate sorteerimiseks: siis mälukulu vaid O(1)
- STL-is `stable_sort`, Javas `java.util.Arrays.sort(Object[])`, `Arrays.sort(Object[], Comparator)`

On võimalik näidata, et iga elementide võrdlemisel põhineva sorteerimisalgoritmi (sealhulgas siis kõigi senivaadeldute) halvimal juhul tehtavate võrdluste arv on vähemalt O(n log n). Nimelt, n erinevat elementi on võimalik järjestada n! erineval moel. Olgu k halvimal juhul tehtav võrdluste arv. Et igal võrdlemisel on kaks võimalikku tulemust ning algoritm ei tee kunagi üle k võrdluse, saab ta eristada ülimalt 2^k juhtu. Et juhul, kui anname ette n erinevast elemendist koosneva järjendi, peab algoritm suutma eristada sisendi kõiki n! võimalikku järjestust, saame, et $2^k \geq n!$ ehk $k \geq \log_2 n!$. Stirlingi valemist $n! \approx (2\pi n)^{1/2}\, n^n / e^n$, kust $\log_2 n! = O(n \log n)$.

Kui kasutame keskmise keerukuse arvutamisel seni vaikimisi tehtud eeldust, et sisendid on piisavalt “juhuslikud”, s. o. ühtlase jaotusega (näiteks kui sisendid koosnevad n erinevast elemendist ning kõik järjestused on võrdvõimalikud), saab näidata, et piir O(n log n) kehtib ka keskmise ajalise keerukuse jaoks. Tegelikult sõltub keskmine keerukus aga sisendite jaotusest: kui sisendite jaotus on väga ebaühtlane (näiteks kui väga suure tõenäosusega on sisend juba peaaegu õiges järjekorras), võib keskmine keerukus olla ka väiksem kui O(n log n).

Siiski on olemas sorteerimismeetodid, mis kulutavad n elemendi sorteerimiseks alati vaid O(n) tehet. Kuidas on see võimalik? Asi on selles, et need meetodid ei võrdle elemente omavahel niisama, vaid kasutavad ära mingit lisainformatsiooni, mis neil elementide kohta on.

Lineaarse keerukusega

```
procedure bitimeetod (var a : massiiv; n : integer);
var b : massiiv; i, k : integer; m : arv;
begin
   m := 1;                                  { alustame üheliste bitist }
   while m > 0 do begin                           { kuni on veel bitte }
      k := 1;                           { esimene vaba koht abimassiivis }
      for i := 1 to n do                     { käime lähtemassiivi läbi }
         if a[i] and m = 0 then begin       { valime 0-bitiga elemendid }
            b[k] := a[i]; k := k + 1;       { ja paneme nad abimassiivi }
         end;
      for i := 1 to n do                     { käime lähtemassiivi läbi }
         if a[i] and m = m then begin       { valime 1-bitiga elemendid }
            b[k] := a[i]; k := k + 1;       { ja paneme nad abimassiivi }
         end;
      for i := 1 to n do                { abimassiiv põhimassiivi tagasi }
         a[i] := b[i];
      m := 2*m;                                        { järgmine bitt }
   end;
end;
```

- ajakulu O(n·k), kus k järjendi elemendi tüübi bittide arv; mälukulu O(n)
- stabiilne (antud näide töötab küll vaid arvudega, kuid on üldistatav suvalistele tüüpidele, mille võtmeks täisarv: sel juhul tuleb avaldis `a[i] and m` asendada avaldisega `voti(a[i]) and m`)
- kuidas tuleks antud näidet muuta, et ta ka negatiivsete väärtustega töötaks?

```
{ järjendi a elementideks on täisarvud lõigust [D, E] }
procedure sagedustemeetod (var a : massiiv; n : integer);
var b : array [D..E] of integer; i, j : integer;
begin
  for i := D to E do
    b[i] := 0;
  for i := 1 to n do              { leiame iga väärtuse esinemissageduse }
    b[a[i]] := b[a[i]] + 1;
  j := D;                     { hakkame järjendit b algusest läbi vaatama }
  for i := 1 to n do begin                  { järjendi a iga koha jaoks }
    while b[j] = 0 do  { senikaua, kui väärtuse j esinemissagedus on 0, }
      j := j + 1;                           { liigume järjendis b edasi }
    a[i] := j;    { paneme leitud vähima esineva väärtuse järjendisse a }
    b[j] := b[j] - 1  { ja vähendame tema järelejäänud esinemissagedust }
  end
end;
```

- ajakulu O(n + M), mälukulu O(M), kus M = E - D + 1 on võimalike väärtuste arv
- stabiilne, sest elementideks arvud: võrdsete võtmetega elemendid (s. t. võrdsed täisarvud) on siin eristamatud

```
{ järjendi a elementideks kirjed, mille võtmeteks täisarvud lõigust [D, E] }
procedure votmeloendusmeetod (var a : massiiv; n : integer);
var b : array [D..E] of integer; vastus : massiiv; i : integer;
begin
  for i := D to E do                        { algväärtustame järjendi b }
    b[i] := 0;
  for i := 1 to n do   { leiame järjendisse b võtmete esinemissagedused }
    b[a[i].voti] := b[a[i].voti] + 1;
  for i := D+1 to E do     { leiame võtme viimase esinemiskoha vastuses }
    b[i] := b[i] + b[i-1];
  for i := n downto 1 do begin            { läbime järjendi a tagurpidi }
    vastus[b[a[i].voti]] := a[i];  { paneme jooksva elemendi vastusesse }
    b[a[i].voti] := b[a[i].voti] - 1;   { juhuks, kui mitu sama võtmega }
  end;
  for i := 1 to n do                { abimassiiv põhimassiivi tagasi }
    a[i] := b[i];
end;
```

- aja- ja mälukulu O(n + M), kus M = E - D + 1 on võtme võimalike väärtuste arv
- stabiilne