Kodutöö esitamise tähtaeg: 17. september, 23:59

Algoritmi efektiivsuse hindamisel on üks olulisi kriteeriume algoritmi täitmiseks tehtavate operatsioonide arv sõltuvalt sisendi suurusest.

Algoritmi ajaline keerukus on funktsioon f, mis algandmete mahule n seab vastavusse algoritmi täitmiseks tehtavate operatsioonide keskmise arvu $f(n)$.

Algoritmi ajaline keerukus määrab ka algoritmi praktilise rakendatavuse piiri: väga kiiresti kasvava keerukusfunktsiooniga algoritmi tasub realiseerida vaid väikesemahuliste ülesannete jaoks.

Ajalise keerukuse empiiriliseks hindamiseks (ja vastavate graafikute joonistamiseks) tuleb mõõta programmi tööaega erinevate algandmemahtude korral.

Javas saab seda teha näiteks järgnevalt:

```
long startTime = System.nanoTime();
// ... the code being measured ...
long estimatedTime = System.nanoTime() - startTime;
```

Järgneva ülesande eesmärk on näidata kui oluline võib olla algoritmi valik suuremahuliste probleemide lahendamisel.

Ülesanne 1

a) (20%) Implementeerida liides *Fibonacci* kasutades otse Fibonacci arvu definitsioonist tulenevat rekursiivset algoritmi (`https://et.wikipedia.org/wiki/Fibonacci_jada`). Leida suurim indeks, millele vastavat Fibonacci arvu on arvuti võimeline välja arvutama 1 sekundi jooksul, kasutades seda implementatsiooni. Rakendada implementatsiooni programmis ja lisada tulemus Moodlesse kommentaarina.

b) (10%) Implementeerida sama liides, kasutades järgnevat algorimi:

```
public int fibonacci(int n) {
       if (n == 0){
           return 0;
       }
       if (n == 1){
           return 1;
       }
       int bigger = 1;
       int smaller = 0;
       int tmp;
       for (int i = 1; i < n; i++){
           tmp = bigger + smaller;
           smaller = bigger;
           bigger = tmp;
       }
       return bigger;
   }
```

Teha kindlaks, kui kaua aega kulub antud implementatsiooniga eelmises punktis leitud indeksiga Fibonacci arvu leidmiseks. Lisada tulemus Moodlesse kommentaarina.

Ülesanne 2

a) (30%) Implementeerida liides *Sorter*, kasutades implementatsioonis üht ruutkeerukusega (keskmise ajalise keerukuse hinnanguga $\Theta(n^2)$) algoritmi järjendi elementide ümberpaigutamiseks mittekahanevasse järjekorda. Sellisteks algoritmideks on näiteks mullimeetod (*bubble sort*), valikumeetod (*selection sort*), pistemeetod (*insertion sort*) jt. Implementatsiooni testimisel mitte unustada nn äärejuhte (järjendid pikkusega 0 või 1, sorditud järjend, vastupidises suunas sorditud järjend).

b) (30%) Implementeerida liides *Sorter*, kasutades üht keskmise ajalise keerukuse hinnanguga $\Theta(n \log n)$ algoritmi järjendi elementide ümberpaigutamiseks mittekahanevasse järjekorda. Sellisteks algoritmideks on näiteks kiirmeetod (*quick sort*), ühildusmeetod (*merge sort*), Shelli meetod (*Shell sort*) jt. Implementatsiooni testimisel pidada silmas ka äärejuhte.

c) (10%) Koostada kahe implementatsiooni võrdlusgraafik(ud), mis ilmekalt demonstreeriksid kahe valitud algoritmi tööaja kasvu vastavalt sorditavate järjendite pikkuste kasvule. Kolmandaks olgu joonisele lisatud ka süsteemse sortimismeetodi tööaja graafik (*java.util.Collections.sort*).

   Kui võimalik, siis tuua graafik ka mingi äärejuhu kohta, kus esimeses punktis kasutatud algoritm on kiirem teises punktis kasutatud algoritmist.

`https://github.com/ut-aa/aa2016-lab1`