# Algoritmid ja andmestruktuurid
# Arvutipraktikum I

Algoritmi efektiivsuse hindamisel on üks olulisi kriteeriume algoritmi täitmiseks kuluv aeg. Algoritmi ajaline keerukus on funktsioon $k$, mis algandmete mahule $n$ seab vastavusse programmi keskmise tööaja $k(n)$. Ütleme, et algoritmi ajalise keerukuse hinnang on $f(n)$, kui leiduvad konstandid $c_1 > 0$, $c_2 > 0$ ja $n_0$ nii, et iga $n > n_0$ korral jääb algoritmi tööaeg suuruste $c_1 f(n)$ ja $c_2 f(n)$ vahele.

Algoritmi ajalise keerukuse hinnang määrab ka algoritmi praktilise rakendatavuse piiri: väga kiiresti kasvava keerukusfunktsiooniga algoritmi tasub realiseerida vaid väikesemahuliste ülesannete jaoks.

Ajalise keerukuse empiiriliseks hindamiseks tuleb mõõta programmi tööaega erinevate algandmemahtude korral. Keeles Python saab seda teha näiteks moodulis *time* defineeritud funktsiooni *time()* abil:

```
from time import *

# MISKIT

a=time()
# PROGRAMMIOSA, MILLE TÄITMISAEGA MÕÕDAME
b=time()

print (b-a)
```

**Ülesanne 1**

Fibonacci arvud defineeritakse seostega $f(0) = 0$, $f(1) = 1$ ja iga $n > 1$ korral $f(n) = f(n-1) + f(n-2)$. Leida suurim Fibonacci arv, mida arvuti on võimeline välja arvutama 1 sekundi jooksul rekursiivse meetodiga

```
def fibo(n):
   if n<1: return 0
   if n<3: return 1
   return fibo(n-1)+fibo(n-2)

print (fibo(9))
```

Leitud väärtuse puhul teha kindlaks sellise meetodi tööaeg, kus Fibonacci arvude arvutamine on realiseeritud mitterekursiivse algoritmiga.

**Ülesanne 2**

Järgmine programm leiab sõne sümbolite kõikvõimalikud permutatsioonid ja väljastab need ekraanile.

```
def perm(algus,veel):
   if len(veel)==1:
      print(algus+veel[0])
   else:
      for i in range(0,len(veel)):
         perm(algus+veel[i],veel[:i]+veel[i+1:])

b='ATI'
perm(' ',b)
```

Muutes sõnet b ja vajadusel ka programmi, leida b suurim pikkus, mille puhul

- b kõikvõimalikud permutatsioonid väljastatakse ekraanile vähem kui 1 sekundi jooksul;
- leitakse b kõikvõimalikud permutatsioonid vähem kui 1 sekundi jooksul, väljastamata neid ekraanile.

Hinnata algoritmi täitmiseks kuluvat aega b erinevate pikkuste puhul.

Tavaliselt tekib vajadus hinnata algoritmi tööaega suurte andmemahtude korral. Probleemsituatsioonide jäljendamiseks on otstarbekas kasutada juhuslike väärtustega järjendeid. Meeldetuletuseks järjenditega ja listidega manipuleerimine:

http://www.cs.ut.ee/~varmo/prog10/loeng4.pdf

**Ülesanne 3**

Koostada funktsioon, mis etteantud täisarvude järjendi korral tagastab selles leiduvate unikaalsete (järjendis ainult üks kord esinevate) elementide arvu. Testida algoritmi töökiirust, kui järjendis on 1000, 2000, 4000, 8000 ja 16000 arvu. Selle põhjal hinnata realiseeritud funktsiooni ajalist keerukust.

Vaatleme sortimisalgoritmidest valikumeetodit:

```
def selectionSort (a):
   for i in range(len(a )):
      min=i
      for j in range(i+1,len(a)):   # otsi minimaalne
         if a[ j ] < a[min]: min=j
      if i!=min:                    # vaheta elemendid
         tmp=a[i ]
```

```
        a[i]=a[min]
        a[min]=tmp
```

**Ülesanne 4**

Testida valikumeetodi töökiirust, kui etteantud täisarvude järjendis on 1000, 2000, 4000, 8000 või 16000 arvu.

**Ülesanne 5**

Võrrelda endaloodud valikumeetodi, pistemeetodi, kiirmeetodi ja ühildusmeetodi (vt. `http://www.cs.ut.ee/~varmo/prog10/loeng7.pdf`) töökiirust. Meetodite töökiirust võrrelda järjendi elementide arvu 1000, 2000, 4000, 8000 ja 16000 puhul. Tulemuste võrdlus esitada tabelitöötlusprogrammis graafikuna.

```
def insertionSort(a):
   for i in range(1,len(a )):
      x=a[i]                        # talleta a[i]
      j = i - 1
      while j>=0 and a[j]>x:        # leia pistekoht
         a[j+1]=a[j]                # nihuta element
         j = j - 1
      a[j+1]=x

def qSort(a,left,right):
   if left>=right: return
   i = left : j=right; x=a[right ]
   while True:
      while i<j and a[i]<=x: i+=1 # leia vasakult suurem
      a[j] = a[i]
      while j>i and a[j]>x: j -= 1 # leia paremalt väiksem
      if i==j: break
      a[i]=a[j]
   a[i]=x
   qSort(a,left,i-1)
   qSort(a,i+1,right)

def quickSort(a):
   qSort(a,0,len(a)-1)

def merge(left,right):
   result=[]; i=0; j=0
   while (i<len(left) and j<len(right)):
      if (left[i]<=right[j]):
```

```
            result.append(left[i]); i+=1
         else:
            result.append(right[j]); j+=1
      result+=left[i:]
      result+=right[j:]
      return result

   def mergeSort(a):
      if len(a)<2: return a
      mid=len(a)/2
      left=mergeSort(a[:mid])
      right=mergeSort(a[mid:])
      return merge(left,right)
```

## 1. iseseisev töö

**Ülesanne 6**

Praktikumijuhendajaga kokkuleppel vali ise üks ülesanne, mille lahendusalgoritm vajab kõikide variantide läbivaatamist (kas siis eksponentsiaal- või faktoriaalkeerukusega). Nõutav on, et see ülesanne teeks midagi "kasulikku".

Tähtaeg: 2 nädalat