LUT Python ohjelmointiopas 2023 - osa 11 Sisällysluettelo

Luku 11: Algoritmi, pseudokoodi ja rekursio

Ohjelmointi aloitetaan usein puoliohjelmoinnilla, joka ei ole mikään oikea ohjelmointikieli, mutta on lähellä useimpia niistä. Tällaista koodia sanotaan usein pseudokoodiksi. Toisaalta tietty tapa ratkaista ongelma, kuten esimerkiksi alkioiden järjestely tai suurten kokonaislukujen kertolasku, on niin spesifi, että sen toteutustapaa voidaan kutsua algoritmiksi. **Tyypillisesti algoritmit kuvaavat tiettyä tehokkaaksi todettua tapaa ratkaista hyvin määriteltyjä ongelmia, ja niiden esittelyssä kirjallisuudessa käytetään usein pseudokieltä.** Yksi tyypillinen esimerkki algoritmista on rekursiivinen ohjelma, jonka erityispiirteet on syytä huomioida ohjelmia tehdessä.

Pseudokielet ovat ainutlaatuisia siksi, että niiltä puuttuu kokonaan kielioppi. Ne eivät varsinaisesti ole humaanisia kirjakieliä kuten esimerkiksi englanti, saksa tai suomi saatikka teknisiä kuten ohjelmointikielet. Yleisesti pseudokielet ovatkin rakenteeltaan edellisten sekoituksia, ohjelmointikielen kaltaisia esityksiä, joissa tapahtumien kuvaus on normaalisti esitetty kirjakielellä. Lisäksi ne saattavat sisältää jonkinlaisia yksinkertaisia malleja ja pseudorakenteita sekä hyödyntää muuttujien perusrooleja – esim. säiliöitä ja askeltajia – yksinkertaistamaan esitystä.

Tässä luvussa tutustumme näihin asioihin esimerkkien kautta. Tarkempaa tietoa algoritmeista ja pseudokoodeista löytyy esim. Wikipediasta hakusanoilla ”algoritmi” ja ”pseudokoodi”. Kannattaa myös huomata, että Google tarjoaa moneen ongelmaan pseudokielisiä ratkaisuja. Ja käytännössä nämä ovat niin keskeisiä asioita ohjelmoinnissa, että tyypillisesti kaikissa alan koulutusohjelmissa on näihin asioihin keskittyvä tietorakenteet ja algoritmit -kurssi.

Pseudokoodista koodiksi

Katsotaan esimerkkinä *lisäyslajittelualgoritmia* (eng. "insertion sort"), joka lajittelee annetun *n* kokonaisluvun lukujonon suuruusjärjestykseen. Menetelmän idea on ottaa jokainen luku vuorollaan käsiteltäväksi ja siirtää sitä niin kauas vasemmalle (alkuun päin) lajitellussa alkuosassa, kunnes vasemmalla puolella on pienempi tai samankokoinen alkio ja oikealla puolella suurempi alkio. Esimerkissä 11.1 on pseudokoodiesitys tälle algoritmille.

Esimerkki 11.1. Lisäyslajittelu pseudokoodina

```
def Lisayslajittelu(lista t)
   n = listan t alkioiden lukumäärä

   # Käydään taulukon jokainen alkio läpi ensimmäistä lukuun ottamatta
   for i = 2 to n # Siirrytään toisesta alkiosta kohti loppua, alkuosa
                  # pysyy lajiteltuna
       j = i
       # Siirretään alkiota vasemmalle niin kauan kunnes seuraava
       # vasemmalla on siirtyjää pienempi tai samanarvoinen.
       while ((t[j] < t[j-1]) ja (j > 1))
         # Tässä vaihdetaan alkioiden t[j] ja t[j-1] arvot keskenään
         j = j - 1 # while-rakenteen edellyttämä indeksin muutos
       end while
   end for
```

Kuten esimerkistä näkyy, koodi näyttää Pythonilta jossain kohdin ja jossain toisessa kohdin syntaksi ei selvästi ole Pythonia. Miten tästä kannattaa siis jatkaa, kun on tarve toteuttaa tämä algoritmi Pythonilla? Helpoin tapa lähteä hahmottelemaan vastausta on rakentaa koodista jonkinlainen malli omalla ohjelmointikielellä. Esimerkin 11.1 pseudokoodissa näkyy valmiina käskyjen toteutusjärjestys sekä joitain konkreettisia rakenteita, kuten

toistorakenteet `for` ja `while`. ja näiden lisäksi tiedämme, että tästä on tulossa aliohjelma eli se alkaa `def`-määrittelyllä alla olevan esimerkin 11.2 vaihe 1 mukaisesti.

Esimerkki 11.2 Vaihe 1. Pseudokoodista koodiksi

```
def lisays
    for
        while
```

Tämän jälkeen voimme hahmotella algoritmiin tarvittavia tietoja eli aliohjelman parametreja. Funktio ottaa selvästi vastaan ainoastaan yhden parametrin, joka on lajiteltava lista. Lisäksi voimme miettiä, miten esim. `for`-lauseen toistomäärä saadaan oikein, eli käymään toisesta alkiosta viimeiseen asti. Nämä asiat on lisätty koodiin vaiheessa 2.

Esimerkki 11.2 Vaihe 2. Pseudokoodista koodiksi

```
def lisays(Lista):
    for Askel in range(1,len(Lista)):
        while
```

Nyt huomaamme, että `for`-lausetta varten tekemämme muuttuja `Askel` on itse asiassa sama kuin pseudokoodin `i`, joten otetaan sen sisältämä tietue talteen ja tehdään samalla muuttuja `j` ohjeen mukaisesti, jotta voimme käyttää tietueiden indeksejä myöhemmin. Nämä näkyvät vaiheessa 3.

Esimerkki 11.2 Vaihe 3. Pseudokoodista koodiksi

```
def lisays(Lista):
    for Askel in range(1,len(Lista)):
        Muisti = Lista[Askel]
        j = Askel
        while
```

Nyt meiltä puuttuu enää `while`-rakenteen ehdot, joten lisäämme ne vaiheessa 4.

Esimerkki 11.2 Vaihe 4. Pseudokoodista koodiksi

```
def lisays(Lista):
    for Askel in range(1, len(Lista)):
        Muisti = Lista[Askel]
        j = Askel
        while ((Lista[j - 1] > Muisti) and (j > 0)): # Hakee tallennuspaikan
```

Nyt funktiomme osaa käydä läpi listaa sekä löytää tiedolle uuden paikan, joten tarvitsemme enää varsinaiset luku- ja kirjoitusoperaatiot vaiheen 5 mukaisesti.

Esimerkki 11.2 Vaihe 5. Pseudokoodista koodiksi

```
def lisays(Lista):
    for Askel in range(1, len(Lista)):
        Muisti = Lista[Askel]
        j = Askel
        while ((Lista[j - 1] > Muisti) and (j > 0)): # Hakee tallennuspaikan
            Lista[j] = Lista[j - 1]
            j = j - 1
        Lista[j] = Muisti # Tallentaa sijoitettavan muuttujan arvon.
```

Vaiheen 5 jälkeen huomaamme, että olemme luoneet pseudokoodista lisäyslajittelufunktion Pythonilla. Python on sen verran uusi ja selkeä kieli, että usein pseudokoodin muuttaminen Pythoniksi ei ole kovin vaikeaa. Lähtökohtaisesti tämä on pseudokoodin idea eli se muistuttaa useimpia ohjelmointikieliä sen verran, että muokkaus tarvittavalle ohjelmointikielelle ei tyypillisesti ole kovin suuri työ vaan edellyttää lähinnä tarkkaa koodin läpikäyntiä ja pieniä muutoksia sopivissa kohdin.

Yleisesti ottaen pseudokoodilla kirjoitettujen algoritmien lukemisen osaaminen helpottaa ohjelmointia myös vaikeita asioita toteutettaessa. Tämän taidon, niin kuin monet muutkin ohjelmoinnin taidot, oppii ainoastaan harjoittelemalla. Pseudokielen luku- ja kirjoitustaito helpottaa lähdekoodin suunnittelua huomattavasti sekä mahdollistaa ei-natiivikielisten esimerkkien hyödyntämisen. Usein ongelmaan löytyy joku aikaisempi ratkaisu, josta voi muokata itselle sopivan ratkaisun tällä pseudokoodi-idealla. Ja aikaisempia ratkaisuja kannattaa etsiä ja katsoa, sillä niissä on yleensä mukana jotain teknisiä yksityiskohtia, jotka pitää tehdä oikein toimivan ratkaisun saamiseksi.

Käytännön ongelmasta ohjelmaksi

Ensimmäinen esimerkkimme oli aika suoraviivainen, koska lähtökohtanamme oli valmis pseudokoodi. Toisessa algoritmiesimerkissä selvitämme, kuinka saamme aikaan parhaan makuista simaa vapuksi.

Ongelma on seuraava. Teija Teekkarilla on käytössään 3 kiloa sokeria, saimaallinen vettä, 1 kilo hiivaa ja 10 litran ämpäri. Siman laatu määräytyy seuraavasti: Sima on laadukkainta, kun hiivaa on mukana $-(h-10)^2+100$, missä h on hiivan prosenttiosuus koko seoksesta. Sokerille vastaava kaava on $-(s-15)^2+100$, missä s on sokerin prosenttiosuus. Lisäksi hiivaa tulee olla kolmasosa sokerin määrästä. Tehtävänä on nyt selvittää, kuinka paljon ämpäriin tulee laittaa hiivaa ja sokeria.

Saamme ongelmasta kaksi matemaattista ehtoa:

$$-(h-10)^2+100 - (s-15)^2+100 = \text{mahdollisimman suuri}$$

$$h = 1/3\ s$$

Sijoitetaan h s:n paikalle:

$$-(h-10)^2+100 - (3*h-15)^2+100$$

Nyt tarvitsee enää selvittää, millä h:n arvolla yhtälö antaa suurimman arvon. Tämän voisimme ratkaista derivoimalla, mutta koska olemme huomanneet, että Python on nopea laskemaan, voimme luoda ohjelman, joka ratkaisee ongelman.

Lähdetään ratkomaan tätä `for`-silmukalla. Silmukka lähtee liikenteeseen nollasta, koska negatiiviset tilavuusprosentit ovat vaikeita toteuttaa fysikaalisesti, ja päättyy luonnollisesti sataan. Ensimmäisessä vaiheessa hahmottelemme koodia ja saamme aikaiseksi alla näkyvän esimerkki 11.3 vaihe 1 koodin.

Esimerkki 11.3 Vaihe 1. Ongelmasta ohjelmaksi

```
MaxLaatu = 0 # tätä etsitään, otetaan mukaan
OikeaProsentti = None # tätä etsitään, otetaan mukaan ohjelmaan
for i = 0 to 100
    UusiLaatu = LaskeLaatu # yllä kaava, otetaan se mukaan myöhemmin
    if (UusiLaatu > MaxLaatu)
        MaxLaatu = UusiLaatu
        OikeaProsentti = i
```

Tarvitsemme ohjelmaamme muuttujat `MaxLaatu`, joka pitää tallessa parhaimman laadun ja `OikeaProsentti`, jossa on tallessa prosenttiarvo parhaalle laadulle. Enää ei puutu kuin pseudokoodin kääntäminen Pythoniksi. Lisätään siis laskenta ohjelmaan ja muutetaan `for`-

lauseelle Python-syntaksi, jolloin saadaan vaiheessa 2 näkyvä alustava ohjelma.

Esimerkki 11.3 Vaihe 2. Ongelmasta ohjelmaksi

```
MaxLaatu = 0
OikeaProsentti = None
for i in range(0, 101):
    UusiLaatu = -(i - 10) ** 2 + 100 - (3 * i - 15) ** 2 + 100
    if (UusiLaatu > MaxLaatu):
        MaxLaatu = UusiLaatu
        OikeaProsentti = i
```

Hiomme vielä ohjelman käyttäjäystävälliseksi, jotta siman tekeminen onnistuu keltä vain. Käytännössä kysytään tietoja käyttäjältä ja tulostetaan tulokset käyttäjälle, jottei käyttäjän tarvitse muokata koodia. Tämä pieni hyötyohjelma ei tarvitse pääohjelmaa jne., mutta ne on helppo lisätä siltä varalta, että ohjelman käyttö laajenee ja sitä kehitetään myöhemmin monipuolisemmaksi. Valmis ohjelma näkyy vaihe 3 koodina.

Esimerkki 11.3 Vaihe 3. Valmis simalaskuri-ohjelma

```
def laske():
    MaxLaatu = 0
    OikeaProsentti = None
    for i in range(0, 101):
        UusiLaatu = -(i - 10) ** 2 + 100 - (3 * i - 15) ** 2 + 100
        if (UusiLaatu > MaxLaatu):
            MaxLaatu = UusiLaatu
            OikeaProsentti = i
    return OikeaProsentti

def paaohjelma():
    Optimi = laske()
    Vetta = int(input("Anna veden määrä litroissa: "))
    print("Sima tarvitsee", Optimi / 100 * Vetta * 1000, "grammaa \ hiivaa")
    print("ja", Optimi / 100 * 3 * Vetta * 1000, "grammaa \ sokeria.")
    return Not

paaohjelma()
```

Tuloste

```
Anna veden määrä litroissa: 10
Sima tarvitsee 500.0 grammaa hiivaa
ja 1500.0 grammaa sokeria.
```

Todellisuudessa hyvän siman tekeminen ei ole näin helppoa. Ohjeestamme puuttuu jo simauutekin aivan täysin, eikä rusinoitakaan liemeen tullut. Haiskahtaa siis hyvin epämääräiselle koko ohje... Kenties tehtävänannossa oli jopa määrittelyvirhe?

Rekursiivinen ohjelma

Rekursio on tehokas tapa ratkaista tiettyjä ohjelmointitehtäviä. Olemme tähän mennessä kirjoittaneet paljon erilaisia ohjelmia ja aliohjelmien myötä ratkaisuissa tahtoo olla mukana rekursiivisia ohjelmia, joten tutustumme seuraavaksi rekursion perusteisiin ja oikean tavan sen toteuttamiseen eli katsomme perusrekursion algoritmin. Rekursio ei ole tämän oppaan ydinasioihin ja rekursiota ei käytetä muulla kuin tässä luvussa.

Erään määritelmän mukaan ”Rekursiivinen algoritmi on algoritmi, jonka toiminta perustuu rekursion käyttöön.” Vaikka tällainen määritelmä on tyypillisesti hyödytön, sopii se

rekursioon oikein hyvin, sillä Tieteen termipankin määritelmän mukaan ”[rekursio on] kielen ominaisuus, että sama rakenne voidaan toistaa periaatteessa rajattoman monta kertaa.” Puuttumatta kielitieteeseen sen enempää matematiikassa rekursio mahdollistaa funktion määrittelyn niin, että ”funktion arvo tietyssä pisteessä riippuu funktion arvosta edellisessä pisteessä.” Kertoman laskenta on tyypillinen esimerkki rekursiosta ja alla oleva kaava esittää luvun *n* kertoman laskentakaavan matemaatikkojen esitystavalla:

$$n! = \prod_{k=1}^{n} k = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n$$

Näin ollen esimerkiksi luvun 5 kertoma lasketaan seuraavalla tavalla:

$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$$

Tämä esitystapa näyttää jo tutulta lukiomatematiikan pohjalta ja kertoma voidaankin laskea helposti esimerkin 11.4 mukaisella toistorakenteella.

Esimerkki 11.4. Kertoman laskenta while-toistorakenteella

```
Kertoma = 1
Alku = 5
N = Alku
while (N > 0):
    Kertoma = Kertoma * N
    N = N - 1
print("Luvun {0:d} kertoma on {1:d}.".format(Alku, Kertoma))
```

Näin saimme laskettua kertoman ilman rekursiota. Tehdään seuraavaksi algoritmi kertoman laskemiseen käyttäen rekursiota. Tieteen termipankin mukaan toistoja voi olla rajattoman monta ja ohjelmoija alkaa tällöin miettiä ikisilmukkaa, joten ohjelmassa pitää olla kaksi haaraa:

1. Saman ohjelman kutsuminen uudestaan eli rekursio.
2. Ohjelman lopettaminen hallitusti eli rekursion päättäminen.

Saman ohjelman kutsuminen uudestaan tarkoittaa sitä, että aliohjelman sisällä on saman aliohjelman kutsu eli tosiaankin *aliohjelma kutsuu itseään*. Tämä kuulostaa oudolta ja siksi tämä ohjelmatyyppi on saanut oman nimen – rekursiivinen ohjelma. Rekursiivinen aliohjelma kertoman laskemiseksi näkyy esimerkissä 11.5.

Esimerkki 11.5. Rekursiivinen aliohjelma eli aliohjelma kutsuu itseään

```
def kertoma(x):
    if (x > 0):                    # Lopetusehtoa ei saavutettu vielä
        return (x * kertoma(x-1)) # Kutsuu itseään, parametri muuttuu
    else:
        return 1    # Lopetuksen paluuarvo on tyypillisesti numerovakio

def paaohjelma():
    Luku = int(input("Minkä luvun kertoman haluat laskea: "))
    print(kertoma(Luku)) # Ensimmäinen rekursiivisen ohjelman kutsu
    return None

paaohjelma()
```

Esimerkissä 11.5 näkyy, miten `kertoma`-aliohjelma kutsuu valintarakenteen ensimmäisessä haarassa itseään ja toisessa haarassa ohjelman suoritus lopetetaan palauttamalla luku 1. Keskeinen asia on, että kutsuttaessa aliohjelmaa uudestaan sitä ei

kutsuta samalla arvolla vaan *parametrin arvo on muuttunut*. Toisella kierroksella parametrin arvo on yhden pienempi, kolmannella kierroksella kaksi pienempi jne. kunnes parametrin arvo on lopulta 0 ja rekursiivisen kutsun sijaan ohjelma palauttaa arvon 1. Tämän seurauksena saadaan suoritettua haluttu kertolasku, 1 * 2 * 3 * … * n, ja saadaan laskettua n:n kertoma n!. Itse laskenta tapahtuu vastaavalla tavalla esimerkkien 11.4 toistorakenteella ja 11.5 rekursiivisella aliohjelmalla, mutta ohjelmien rakenteet eli toiston periaate poikkeaa näissä ratkaisuissa.

Rekursion osalta esimerkissä 11.5 on suora rekursio eli aliohjelma kutsuu itse itseään suoraan. Toinen vaihtoehto on epäsuora rekursio, jolloin tyypillisesti kaksi aliohjelmaa kutsuu toisiaan vuorotellen. Suora rekursio on usein harkinnan ja suunnittelun tulos, kun taas epäsuoraan rekursioon päädytään usein huomaamatta ja ohjelman suorituskyky kärsii, kun ohjelma käyttää aikaa aliohjelmakutsujen ketjuttamiseen.

Tärkein syy rekursiivisen aliohjelman läpikäyntiin tässä oppaassa on pystyä välttämään hallitsematon rekursio. Oikein toteutettuna rekursiivinen algoritmi on tyypillisesti lyhyt ja helppo ymmärtää kokeneelle ohjelmoijalla, mutta kokemattomat ohjelmoijat päätyvät usein rekursioon vahingossa ja tällöin ohjelman suorituskyky voi kärsiä. Rekursion keskeisimpiä asioita on sen hallittu lopettaminen, mikä pätee kaikkiin aliohjelmiin eli kaikki aliohjelmat tulee lopettaa `return` -käskyyn tyypillisesti paluuarvon kanssa. Näin paluu aliohjelmasta tapahtuu kutsun jälkeiseen kohtaan, aliohjelman varaamat resurssit palautetaan käyttöjärjestelmän käyttöön ja aliohjelman suoritus päättyy suunnitellusti. Tyypillisesti rekursio on yksi tietorakenteet ja algoritmit -kurssein keskeisiä aiheita.

Yhteenveto

Osaamistavoitteet

Tämän luvun keskeiset asiat on nimetty alla olevassa listassa. Tarkista sen avulla, että tunnistat nämä asiat ja sinulla on käsitys siitä, mitä ne tarkoittavat. Mikäli joku käsite tuntuu oudolta, käy siihen liittyvät asiat uudestaan läpi. Tämän luvun asiat keskittyvät korkeamman tason konseptien ymmärtämiseen eikä niiden soveltaminen ole välttämättä vielä sujuvaa tämän oppaan laajuudessa vaan ne tulevat tutuiksi ohjelmointikokemuksen kertymisen myötä.

- Ymmärrät pseudokoodi-käsitteen ja miten siitä tehdään Python-koodia (E11.1, 11.2)
- Ymmärrät, miten ongelma muotoillaan algoritmiksi ja algoritmi ohjelmaksi (E11.3)
- Ymmärrät rekursio-käsitteen ja pystyt välttämään sen turhaa käyttöä (E11.4, 11.5)

Pienen Python-perusohjelman tyyliohjeet

Tähän lukuun liittyvät tyyliohjeet on koottu alle. Koska tämä luku keskittyy korkeamman tason konseptien ymmärtämiseen, tyyliohjeita on vain yksi.

1. Rekursiota käytetään vain tehtävänannossa niin sanottaessa