jamk.fi

Peräkkäis- ja puolitushaku

Tietorakenteet ja algoritmit

Jere Pesonen M3227, TTV18S1
Joel Rinta L5371, TTV18S1

Harjoitustyö
Huhtikuu 2020
Insinööri (AMK), Tieto- ja viestintätekniikka
Tekniikan ja Liikenteen ala

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

Sisällysluettelo

1 Johdanto

Harjoitustyön aiheena on toteuttaa ja ohjelmoida toimiva hakualgoritmi käyttäen sekä peräkkäis- että puolitushakua. Tarkoitus on tutkia toteutettujen algoritmin suoritusaikoja eri n-arvoilla, sekä vertailla niitä teoreettisiin vastaaviin. Ohjelmat on toteutettu python ohjelmointikielellä.

Harjoitustyö on toteutettu osana tietorakenteet ja algoritmit kurssia.

2 Algoritmien tehokkuus

Peräkkäishaku etsii arvoa taulukosta käymällä sen järjestyksessä läpi alkio alkiolta. Peräkkäishaku ei vaadi, että alkiot ovat suuruusjärjestyksessä. Heti kun alkio on löytynyt, sen indeksi palautetaan. Jos alkiota ei löydy, palautetaan indeksin sijaan arvo –1. Jos alkioita ei löydy, algoritmi tekee taulun koon verran arviointeja. Isoissa tauluissa tähän menee paljon aikaa. Eli jos taulukossa on miljoona arvoa, joudutaan huonoimmassa tapauksessa tekemään miljoona vertailua. Peräkkäishaussa suoritusaika on O(n), jossa n on alkioiden lukumäärä. (Algoritmiikkaa 2019.)

Jos alkiot ovat järjestyksessä voidaan käyttää nopeampaa puolitushakua. Tiedon etsiminen aloitetaan taulukon keskimmäisestä arvosta. Tämän jälkeen verrataan, onko haettu arvo pienempi vai suurempi kuin keskimmäinen arvo. Näin saadaan puolet tarkasteltavasta alueesta pois. Puolittamista jatketaan, kunnes etsittävä alkio on taulukon keskimmäinen arvo.

Jos puolitusta jatkettaessa viimeiseen tarkasteltavaan taulukkoon jää vain yksi alkio ja se ei ole etsittävä alkio, voidaan päätellä, että etsittävä alkio ei ole taulukossa. Puolitushaussa huonoimmassa tapauksessa puolitetaan luku miljoona vain 20 kertaa, jolloin jää jäljelle noin arvo 1. Koska jokainen haku puolittaa taulun, algoritmin suoritusaika on $O(\log_2 n)$, jossa n on alkioiden lukumäärä. (Algoritmiikkaa 2019.)

3 Toteutus

Molemmat algoritmit tutkivat lista- tietorakennetta, jossa on listan alkioiden määrä eri numeroita. Numerot ovat uniikkeja, ja ne ovat väliltä 0 – alkioiden lukumäärä. Peräkkäishaussa listan numerot on sekoitettu satunnaiseen järjestykseen (tällä ei ole ohjelman kannalta vaikutusta algoritmin nopeuteen), kun taas puolitushaussa numerot ovat järjestyksessä nousevasti pienimästä suurimpaan. Molemmat algoritmit ovat toteutettu python- ohjelmointikielellä.

3.1 Peräkkäishaku

Peräkkäishakualgoritmi käy läpi listan jokaisen alkion for silmukassa ja vertaa alkiosta löytyvää arvoa haettuun numeroarvoon. Tutkiminen aloitetaan ensimmäisestä alkiosta. Ensin verrataan vastaako kyseisen indeksipaikan alkion arvo haettua, jos ei, niin for silmukka kasvattaa indeksin arvoa niin kauan, kunnes kyseisen numeroarvon sisältävä indeksipaikka löytyy, tai vastaavasti koko lista tulee käytyä läpi. Silmukkaan on tehty myös ehto, joka tarkistaa, onko tutkittava alkio listan viimeinen. Jos on, niin ohjelma ilmoittaa käyttäjälle, että hänen hakemaansa numeroa ei listasta löytynyt.

```
# tuodaan random ja time paketit
import random
import time
lista = [] # alustetaan lista
for x in range(0,100000001):
    lista.append(x) # lisätään listaan haluttu määrä lukuja nollasta eteenpäin
def Peräkkäis(lista = []): # luodaan funktio, joka toteuttaa peräkkäishaun
    etsi = len(lista) -1 # asetetaan etsittäväksi luvuksi listan viimeinen luku
    for i in range (len(lista)): # for silmukka, jossa käydään listan alkiot yksi kerrallaan läpi
        if lista[i] == etsi: # verrataan i:n arvoa etsittävän numeron arvoon
            return("Hakemasi numero löytyi muistipaikalta: " # jos etsittävä numero löytyy, palauetetaan muistipaikan numero
            + str(i))
            break;# silmukka keskeytetään
    return("Hakemaasi numeroa ei löytynyt") # jos lista käydään kokonaan läpi, eikä etsittävää numeroa löydy, palautetaan ilmoitus


start = time.time() # aloitetaan kellotus
print(Peräkkäis(lista))# kutsutaan Peräkkäis muuttujaa lista parametrillä
end = time.time() # pysäytetään kellotus

print("Hakuun kului aikaa", end-start, "sekuntia") # tulostetaan hakuun kulunut aika
```

Kuva 1. Peräkkäishakualgoritmin python koodi

3.2 Puolitushaku

Puolitushaku on toteutettu muuttujilla L, ja R (left ja right), jotka rajaavat alueen, jolta etsittävää numeroa haetaan. Aluksi L on listan ensimmäisen alkion luku (tässä tapauksessa 0), Ja R on listan viimeinen luku (huom. numeroiden tulee olla suuruusjärjestyksessä, jotta puolitushaku toimii). Haku aloitetaan puolittamalla L ja R muuttujien summa, jolloin saadaan listan keskimmäisen indeksin arvo. Aina puolituksen jälkeen verrataan keski-indeksiä haettavaan arvoon. Jos haettava arvo löytyy, se tulostetaan konsolille, ja algoritmin suoritus päättyy.

Jos haettava arvo ei vastaa keski-indeksiä, puolet listasta voidaan rajata hakualueesta pois. Jos haettava luku on suurempi kuin keski-indeksi, muuttujan L arvoksi asetetaan keski-indeksi, jos taas luku on pienempi, kuin keski-indeksi muuttujan R arvoksi asetetaan keski-indeksi. Näin listasta voidaan jokaisella hakukerralla rajata puolet alkioiden määrästä, josta haettua lukua ei varmasti löydy.

Ohjelman lopetusehto on se, kun muuttuja R on pienempi, kuin muuttuja L. Tässä tapauksessa lista on käyty kokonaisuudessaan läpi, eikä siitä löydy haettavaa numeroa.

```
# tuodaan random ja time paketit
import random
import time
lista = [] # alustetaan lista

for x in range(0,10001):
    lista.append(x) # lisätään listaan haluttu määrä lukuja nollasta eteenpäin

def Puolitus(lista = []): # luodaan funktio, joka toteuttaa puolitushaun
    etsi = len(lista)-1 # asetetaan etsittäväksi luvuksi listan viimeinen luku
    print(etsi) # tulostetaan luku
    L = lista[0] # alustetaan muuttujaksi L, listan ensimmäisen alkion luku
    R = lista[-1]# alustetaan muuttujaksi R, listan viimeisen alkion luku
    while (L <= R): # while silmukka, joka pyörii niin kauan, kun L on pienempi tai yhtäsuuri, kuin R
        if(lista[int((L+R)/2)] == etsi): # if lause vertaa, että onko löytyykö listan muistipaikasta L+R/2, etsitty luku.
            return("Luku löytyy muistipaikasta: " + str(int((L+R)/2)))# jos löytyy palautetaan kyseinen muistipaikka
        elif(etsi > int((L+R)/2)): # if lause tutkii, onko etsitty luku isompi, kuin listan keskimmäinen luku
            L = int((L+R)/2) + 1 # jos on asetetaan L muuttujaksi listan keskimmäinen luku, ja listan lukujen määrä puolittuu
            #print(L)
        else: # viimeisessä vaihtoehdossa etsitty luku on pienempi, kuin listan keskimmäinen luku
            R = int((L+R)/2) # jos on asetetaan R muuttujaksi listan keskimmäinen luku, ja listan lukujen määrä puolittuu
            #print(R)
    return("Hakemaasi numeroa ei löytynyt") # jos while silmukka pääsee loppuun asti, palautetaa, että etsittyä lukua ei löytynyt

start = time.time()# aloitetaan kellotus
print(Puolitus(lista))# kutsutaan Puolitus funktiota lista parametrillä
end = time.time()# pysäytetään kellotus
print("aikaa kului", end-start, "sekuntia") # tulostetaan hakuun kulunut aika
```

Kuva 2. Puolitushakualgoritmin python koodi

4 Testaus

Testasimme algoritmeja hakemalla vaikeimmin löydettävää numeroa. Käsiteltävien algoritmien ”vaikeimmin” löytävänä numerona voidaan käyttää esimerkiksi listarakenteen viimeistä numeroa. Testatessa käytimme eri n- alkion arvoja, jotka olivat 10000–10000*10^4(Taulukko 1Taulukko 1. Peräkkäishaun tulokset (n)).

Algoritmi ajettiin jokaisella n:n arvolla 5 kertaa, joista kellotimme algoritmin suoritusnopeuden ja lopuksi laskimme aikojen keskiarvon.

Taulukoimme saadut tulokset ja laskimme skaalatun ajan vertaamalla suoritusajan muutosta suhteessa alkioiden lukumäärään.

4.1 Peräkkäishaku

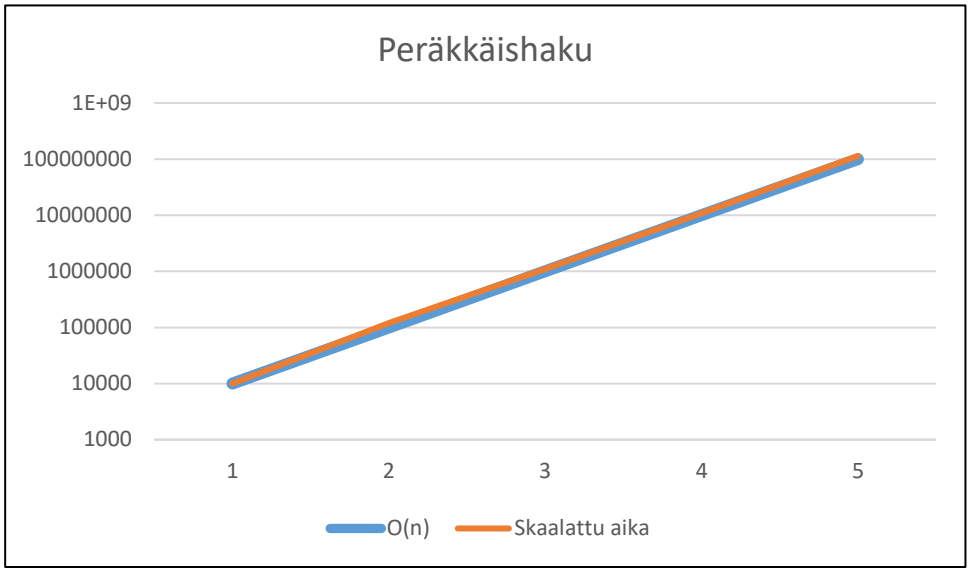
Ohessa taulukko (Taulukko 1), johon on siis listattu algoritmin suoritusnopeus eri n arvoilla, ja laskettu skaalattu aika tämän perusteella. Taulukosta voi helposti nähdä, että tulokset ovat onnistuneita, koska skaalatut ajat ovat hyvin lähellä algoritmin teoreettista nopeutta(O(n)). Saman pystyy toteamaan kaaviosta (

Kaava 1), Josta nähdään, että skaalatun ajan, ja teoreettisen nopeuden viivat ovat hyvin identtisen näköiset.

Taulukko 1. Peräkkäishaun tulokset

n	O(n)	mitattu t	skaalattu aika
10000	10000	0,000568	10000
100000	100000	0,005984	105278,0954
1000000	1000000	0,054861	965174,3442
10000000	10000000	0,542949	9552132,135
1E+08	1E+08	5,694067	100176098,9

Kaava 1. Peräkkäishaku algoritmin teoreettinen ja kokeellinen suoritusnopeus.



4.2 Puolitushaku

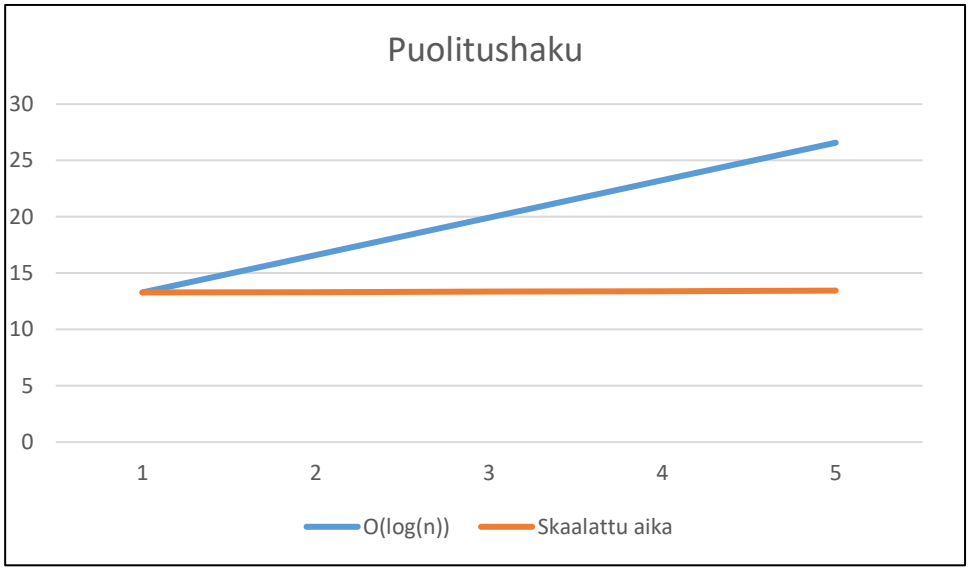
Taulukossa (Taulukko 2) näkyy puolitushaun tulokset. Taulukon log(n) arvosta voidaan päätellä, että algoritmi on hyvin tehokas. Vertailujen määrä kasvaa noin kolmella, kun alkioiden määrä kymmenkertaistuu.

Kuten taulukosta, sekä kaaviosta (Kaava 2) voidaan nähdä, myös suoritusajat ovat erittäin minimaaliset, sekä niiden keskinäiset erot ovat hyvin pieniä.

Taulukko 2. Puolitushaun tulokset

(n)	log(n)	mitattu aika	skaalattu aika
10000	13,28771	0,001958418	13,28771238
100000	16,60964	0,001959038	13,29191827
1000000	19,93157	0,001968622	13,35694786
10000000	23,2535	0,001971769	13,37830086
1E+08	26,57542	0,001981831	13,44656575

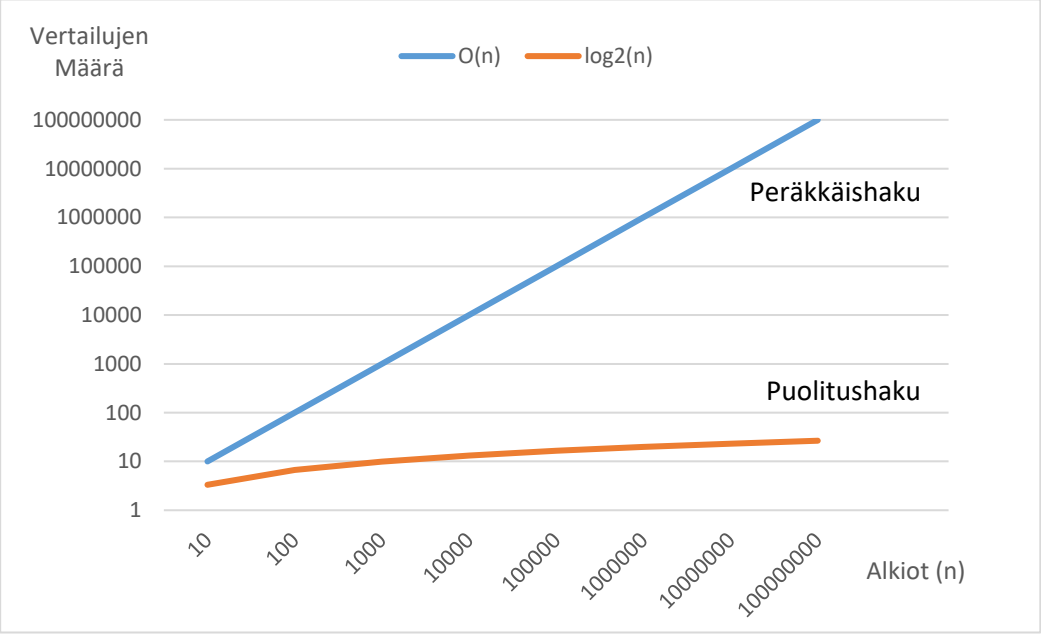
Kaava 2. Kaavio puolitushakualgoritmin suoritusnopeudesta.



5 Yhteenveto

Mitattujen aikojen tuloksista nähdään suuri nopeusero näiden kahden hakualgoritmin välillä (Kaava 3). Peräkkäishaussa mitattu aika nousee lineaarisesti taulukon koon muuttuessa. Puolitushaussa vertailujen määrä, kasvaa todella hitaasti, sekä mitattu aika pysyy melkein samassa, vaikka taulukon kokoa muutetaan radikaalisti. Riippuu aivan tilanteesta, että kumpi näistä haku algoritmeista on parempi. Jos dataa on vähemmän ja se on sekaisin, niin kannattaa käytetään peräkkäishakua, koska puolitushakua ei voida käyttää tässä tilanteessa. Tietenkin jos data on järjestyksessä, niin puolitushaku on tässä hakemisessa huomattavasti parempi.

Kaava 3. Hakualgoritmien vertailujen määrät vertailtavana



6 Lähteet

Algoritmiikkaa. Ohjelmoinnin MOOC 2019. Helsingin yliopisto. Viitattu 24.3.2020. https://ohjelmointi-19.mooc.fi/osa-7/2-algoritmiikkaa.