Ohjelmointiesimerkkejä kurssilla TIEA211 Algoritmit 2

Antti Valmari

8. toukokuuta 2023

Tässä tekstissä on esimerkkejä, joiden tarkoitus on auttaa opiskelijoita kurssin TIEA211 Algoritmit 2 ohjelmointiharjoitusten tekemisessä. Esimerkeissä havainnollistetaan ohjelmien suunnittelua ja kerrotaan syitä, miksi kannattaa valita jokin ratkaisu jonkin toisen sijaan. Saat suurimman hyödyn esimerkeistä toteuttamalla niitä itse samalla kun luet tekstiä.

Siltä osin kuin esimerkit esitetään ohjelmointikielellä, kielenä on C++. Jotta esimerkeistä ei tulisi kovin kielisidonnaisia, siitä käytetään enimmäkseen vain peruspiirteitä. Voit itse toteuttaessasi käyttää monia muitakin kieliä. Puhtaasti funktionaaliset kielet eivät kuitenkaan sovellu, sillä esimerkeissä tarvitaan sijoituslauseita ja taulukoita.

Tekstin seassa on kysymyksiä. Tarkoitus on, että mietit niitä ensin itse ja sen jälkeen katsot vastauksen klikkaamalla sivun reunassa olevan numeroa. Kun olet katsonut, pääset takaisin klikkaamalla vastauksen vieressä olevaa numeroa. Jos PDF-tiedostonlukijasi on tarpeeksi taitava, niin ei tarvitse edes klikata, vaan riittää siirtää kursori numeron päälle. Vieressä on esimerkki sellaisesta numerosta. 1 Klikkaa sitä nyt!

1 Harjoitustehtäviä

Ellei erikseen käsketä toisin, käytä ohjelmointitehtävissä kokonaislukujen ja merkkijonojen lukemiseen jotakin komentoa, joka ohittaa tyhjän ja sitten lukee yhden kokonaisluvun tai merkkijonon. C++:ssa se on `std::cin >> muuttuja;` ja Javassa `muuttuja = syote.nextInt();` tai `muuttuja = syote.next();`. Ellei erikseen käsketä toisin, tulostuksissa saat käyttää välilyöntejä ja rivinsiirtoja miten tahansa, kunhan jokaisen kahden tekstialkion välissä on ainakin yksi välilyönti ja/tai rivinsiirto.

Perusohjelmointi. Älä käytä tehtävissä 2, . . . , 4 hajautustaulua, binääripuuta tai muuta sellaista, vaan jotakin yksinkertaisempaa.

Matalin Suomessa mitattu ilmanpaine on 940 hPa ja korkein 1066 hPa[1]. Tee yksinkertainen, sekä ajan että muistin käytöltä tehokas ohjelma, joka ottaa vastaan mitattuja ilmanpainelukemia ja sen jälkeen vastaa kysymyksiin muotoa ”esiintyykö lukema x aineistossa”. Ohjelma saa luottaa siihen, että sekä mitatut että kysytyt lukemat ovat kokonaislukuja väliltä 940, . . . , 1066. Syötteessä on ensin mitatut lukemat, sitten 0, sitten kysytyt lukemat x ja lopuksi 0. Esimerkiksi syötteelle `1005 1023 988 1005 1001 0 1000 1001 1002 999 1005 0` ohjelman pitää vastata 2

```
1000: ei
1001: kyllä
1002: ei
999: ei
1005: kyllä
```

Muuta edellinen ohjelma sellaiseksi, että se yhä luottaa, että jokainen syötteessä tuleva paine on vähintään 940 hPa, mutta ei oleta ylärajasta muuta kuin että mitattu paine ei voi olla kovin paljon suurempi kuin 1000 hPa. Jos syöte on iso, niin se on iso siten, että mitattuja ja/tai kysyttyjä paineita on paljon enemmän kuin suurin mitattu paine. Kysytty paine saa olla paljon suurempi kuin 1000 hPa. Nytkin kaikki lukemat ovat positiviisia kokonaislukuja. Käytä yksinkertaista tietorakennetta, joka sopeutuu suurimman syötteessä tulleen mitatun paineen mukaan! 3

Muuta edellinen ohjelma sellaiseksi, että se luottaa vain, että mikään syötteessä tuleva mitattu paine ei ole kovin kaukana arvosta 1000 hPa. Kysytty paine saa olla mikä tahansa positiivinen kokonaisluku. 4

Tehtävissä 5, . . . , 11 suunnitellaan aliohjelma `sama_monikulmio( mk1, mk2 )`, joka vertaa, onko sille annetut kaksi monikulmiota samat. Monikulmio ilmoitetaan antamalla sen kärkipisteet. Ne voidaan antaa mistä tahansa alkaen ja kumpaan suuntaan tahansa. Esimerkiksi $(1,2)(2,3)(3,3)(4,2)$ ja $(2,3)(1,2)(4,2)(3,3)$ ovat sama nelikulmio. Toisaalta $(1,2)(2,3)(3,3)(4,2)$ ja $(1,2)(3,3)(2,3)(4,2)$ eivät ole sama nelikulmio, koska vain toisessa niistä $(1,2)$ ja $(2,3)$ ovat naapurit. Aliohjelma saa luottaa siihen, että sama piste ei koskaan esiinny samassa monikulmiossa useampana kuin yhtenä kärkenä. Yksinkertaisuuden vuoksi kaikki koordinaatit ilmoitetaan kokonaislukuina.

Suunnittele luokka esittämään piste. Sillä tarvitsee olla rakentaja joka saa x- ja y-koordinaatit, sekä yhtäsuuruuden ja erisuuruuden vertailut. 5

Oletetaan, että `monikulmio` on tyypin nimi taulukolle pisteitä. (C++:ssa sen saa aikaan määritelmällä `typedef std::vector < piste > monikulmio;`.) Oletetaan, että verrattavat monikulmiot ovat muuttujissa `mk1` ja `mk2`. Kirjoita ohjel-

[1] `https://www.ilmatieteenlaitos.fi/saaennatyksia`

manpätkä, joka palauttaa oikean vastauksen silloin, kun niissä on eri määrä kärkiä tai ne molemmat ovat tyhjiä. Muussa tapauksessa suoritus jatkuu tavalla, joka suunnitellaan myöhemmin. 6

Luonnostele toimintaperiaate aliohjelman `sama_monikulmio` loppuosalle. Älä vielä ohjelmoi. 7

Kirjoita ohjelmanpätkä, joka tulee vastauksen 6 jatkoksi ja etsii monikulmion `mk2` sen kärjen, joka on sama kuin monikulmion `mk1` ensimmäinen kärki. Jos sellaista ei ole, se palauttaa `false` . Jos sellainen on, niin suoritus jatkuu seuraavan tehtävän vastaukseen. 8

Kirjoita ohjelmanpätkä, joka tulee edellisen vastauksen jatkoksi ja palauttaa `true` , jos ja vain jos `mk2` :n kärjet äsken löydetyn kärjen jälkeen ovat samat kuin `mk1` :n kärjet toisesta alkaen. Muussa tapauksessa suoritus jatkuu seuraavan tehtävän vastaukseen. 9

Kirjoita ohjelmanpätkä, joka tulee edellisen vastauksen jatkoksi ja palauttaa `true` , jos ja vain jos `mk2` :n kärjet takaperin äsken löydetyn kärjen edeltäjästä alkaen ovat samat kuin `mk1` :n kärjet etuperin toisesta alkaen. Muussa tapauksessa se palauttaa `false` . 10

Miksi tyhjät monikulmiot tarvitsee käsitellä erikoistapauksena? 11

Osana matematiikan perusteiden tutkimusta Giuseppe Peano julkaisi 1889 luonnollisille luvuille määritelmän, joka tunnetaan nykyisin *Peanon aksioomina*. Siitä kehitettiin myöhemmin *Peanon aritmetiikka*. Se on käytännön laskuihin liian hidas, mutta sopii havainnollistamaan sitä, kuinka rekursiivisen algoritmin suoritus saattaa aiheuttaa suuren määrän aliohjelmakutsuja. Siinä yhteenlasku ja kertolasku määritellään tavalla, joka muistuttaa seuraavia aliohjelmia. Niissä laskutoimitukset palautuvat viime kädessä toiminnoiksi `++` ja `--` .

```
unsigned plus( unsigned xx, unsigned yy ){
  if( yy == 0 ){ return xx; }
  --yy; xx = plus( xx, yy ); ++xx; return xx;
}

unsigned kerto( unsigned xx, unsigned yy ){
  if( yy == 0 ){ return 0; }
  --yy; return plus( kerto( xx, yy ), xx );
}
```

Kuinka monta kertaa kutsun `kerto(12345, 67890)` suorituksen aikana suoritetaan aliohjelma `plus` ? Vihje: lisää globaali muuttuja, jonka arvo on aluksi nolla, ja jota kasvatetaan yhdellä aina aliohjelman `plus` suorituksen alussa. 12

Luettele aliohjelmien kutsut, jotka syntyvät kutsusta `plus(2,3)` ! Luettelon lyhentämiseksi käytä sanojen `kerto` ja `plus` sijaan kirjaimia `k` ja `p` , ja jätä sulkeet pois. Kutsu `plus(2,3)` on tässä muodossa `p2,3` . Voit ratkaista tehtävän itse tai laittaa tietokoneen tulostamaan vastauksen. 13

Luettele aliohjelmien kutsut, jotka syntyvät kutsusta `kerto(1,2)` ! 14

Piirrä puu, jonka solmuja ovat edellisen tehtävän vastauksena luetellut kutsut, ja solmusta on kaari vinosti alaspäin toiseen jos ja vain jos jälkimmäinen kutsu tehtiin ensimmäistä kutsua suoritettaessa! 15

Luettele aliohjelmien kutsut, jotka syntyvät kutsusta `kerto(2,3)` , sekä piirrä kutsujen muodostama puu! 16

Kuinka monta kertaa kutsun `kerto`(n,m) suorituksen aikana suoritetaan aliohjelma `kerto` , alkuperäinen `kerto`(n,m) mukaan lukien? 17

Kuinka monta kertaa kutsun `kerto`$(n,1)$ suorituksen aikana suoritetaan aliohjelma `plus` ? 18

Kuinka monta kertaa kutsun `kerto`(n,m) suorituksen aikana suoritetaan aliohjelma `plus` ? Vertaa tulosta tehtävän 12 vastaukseen! 19

Perusmuotoisessa linkitetyssä listassa on linkki (osoitin, indeksi tai muu sellainen) listan alkuun, jokaisessa muussa kuin viimeisessä alkiossa on linkki `seur` seuraavaan alkioon, ja viimeisen alkion `seur` ei osoita minnekään. Kahden perusmuotoisen linkitetyn listan yhdistäminen on hidasta, koska jompikumpi lista pitää selata läpi. Kuvaile hieman muutettu listarakenne, jolla yhdistäminen on nopeaa, ja jonka muistin kulutus on sama ja muiden toimintojen nopeus olennaisesti sama kuin perusmuotoisessa linkitetyssä listassa! Kirjoita sille nopea ohjelma- tai pseudokoodi `yhdista( eka, toka )` , joka palauttaa osoittimen listaan, jossa listat `eka` ja `toka` on yhdistetty niin että `eka` :n alkiot ovat ensin! 20

Olkoon jokaisessa alkiossa myös kenttä `nimi` . Kirjoita ohjelma- tai pseudokoodi, joka tulostaa listan alkioiden nimet välilyönneillä toisistaan erotettuina ja lopuksi rivinsiirron. Nimet pitää tulostaa listan mukaisessa järjestyksessä. 21

Pino ja jono. Merkit `(` , `[` ja `{` ovat alkusulkeita, ja `)` , `]` ja `}` ovat niitä vastaavat loppusulkeet. Miten pinon avulla voidaan toteuttaa aliohjelma, joka tarkastaa, että sille annetussa merkkijonossa sulkeet ovat oikein? Jokaisella alkusulkeella on oltava sitä vastaava loppusulje ja päinvastoin, alkusulkeen on oltava aikaisemmin kuin sitä vastaava loppusulje, eivätkä sulkeet saa mennä limittäin. Esimerkiksi `({})` on sallittu, mutta `({)}` on kielletty. Merkkijonossa voi olla muitakin merkkejä kuin sulkeita. 22

Näytä, miten jono voidaan rakentaa kahdesta pinosta siten, että yksittäinen toiminto voi olla hidas, mutta tyhjästä jonosta aloittavan n:n mittaisen toimintojonon suoritusaika on silti $\Theta(n)$. Pinoissa on ja jonossa pitää olla vakioaikaiset toiminnot `on_tyhja` , `laita` ja `ota` . 23

Keko. Kirjoita pseudokoodilla tai jollain ohjelmointikielellä aliohjelma, joka palauttaa true tai false sen mukaan onko taulukko $K[1\ldots n]$ keko, jossa suurin alkio on ensimmäisenä! 24

Jos kekotaulukko halutaankin indeksoida $0 \ldots n-1$, niin mikä on solmun i vanhemman numero ja mitkä ovat solmun i lasten numerot (sikäli kuin on kaksi lasta)? 25

Tee taulukosta $[\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{H}]$ keko seuraavasti. Lopputuloksessa suurimman alkion pitää olla ensimmäisenä. Käy taulukko läpi puolivälistä alkuun. Jokaisessa kohdassa korjaa ko. kohdasta alkava alipuu keoksi ottamalla ko. kohdassa oleva alkio talteen, valuttamalla syntynyttä aukkoa alaspäin kuten keosta poistamisen algoritmin pääsilmukassa, ja sijoittamalla talteen otettu alkio sinne minne aukko lopulta valui. Aina yhden kohdan korjaamisen jälkeen näytä taulukon sisältö! 26

Kuinka nopea heapsort voi nopeimmillaan olla, Θ-merkinnällä ilmaistuna? Anna esimerkki syötteestä, jolla heapsort on niin nopea! 27

Oletetaan, että k on paljon pienempi kuin n. Luonnostele algoritmi, joka ajassa $O(n \log k)$ tulostaa n alkiosta k suurinta. 28

Puurakenteet. Jos binääripuun vasemmassa alipuussa on v solmua ja oikeassa alipuussa on o solmua, niin paljonko koko binääripuussa on solmuja? 29

Kirjoita pseudokoodilla tai jollain ohjelmointikielellä aliohjelma, jolle annetaan osoitin binääripuun solmuun ja joka palauttaa ko. solmusta alkavan alipuun solmujen määrän! 30

Jos epätyhjän binääripuun juuren vasemman alipuun korkeus on v ja oikean alipuun korkeus on o, niin paljonko on koko binääripuun korkeus? 31

Oletetaan, että binääripuun jokaisessa solmussa on ylimääräinen kenttä n, jonka sisältö kertoo, montako solmua on siinä alipuussa, jonka juuri on ko. solmu. Kirjoita hitaimmillaan puun korkeuteen verrannollisessa ajassa toimiva algoritmi, jolle annetaan osoitin p ja luonnollinen luku i, ja joka palauttaa osoittimen solmuun, jota ennen p:stä alkavassa alipuussa on täsmälleen i solmua! Jos i on liian suuri, aliohjelma palauttaa $\perp$. 32

Jokaisessa solmussa on myös osoitin y vanhempaan. Kirjoita hitaimmillaan puun korkeuteen verrannollisessa ajassa toimiva algoritmi, jolle annetaan osoitin solmuun, ja joka palauttaa luvun, joka kertoo, kuinka monta solmua puussa on ennen ko. solmua! 33

Alun perin tyhjään B-puuhun, jonka jokaisessa solmussa pitää olla enintään kolme alkiota, lisätään tässä järjestyksessä E, C, F, A, B ja D. Piirrä kuvat välivaiheista ja lopputuloksesta! 34

Koska B-puun solmuissa ei ole osoitinta vanhempaan, ei voida toteuttaa samanlaista seuraajatoimintoa kuin binääripuilla. Mutta voidaan toteuttaa toiminto $\mathit{seuraava}(p, a)$, jolle annetaan osoitin B-puun juureen ja avain a, ja joka palauttaa seuraavaksi suuremman puussa olevan avaimen tai tiedon, että sellaista ei ole olemassa. Avain a voi itse olla puussa mutta ei välttämättä ole. Oletamme, että

kaikki puussa olevat avaimet ovat keskenään erisuuret. Käytämme seuraavia merkintöjä: B-puun solmun p käytössä olevien avainten määrä on $p\uparrow n$, avaimet ovat $p\uparrow k[1 \ldots p\uparrow n]$, osoittimet lapsiin ovat $p\uparrow c[1 \ldots p\uparrow n+1]$, ja bitti $p\uparrow l$ kertoo onko kyseessä lehtisolmu.

Jos etsittävä avain on lehtisolmussa johon p osoittaa, niin miten se löydetään? Kuvaile sanallisesti ja kirjoita pseudo- tai ohjelmakoodi! 35

Jos ei olla lehtisolmussa, niin miten etsittävä avain löydetään? Kuvaile sanallisesti ja kirjoita pseudo- tai ohjelmakoodi! 36

Yhdistä edellisten tehtävien vastaukset yhdeksi pseudo- tai ohjelmakoodiksi! 37

Männikön luennon 7 ruudussa 21 annettu alkiovieraiden osajoukkojen `union(a, b)` luottaa siihen, että `a` ja `b` ovat juurisolmuja. Ruudussa 23 sen eteen lisätään `a = find(a); b = find(b);` varmistamaan, että `a` ja `b` todellakin ovat juurisolmuja, ja kysytään, kannattaisiko lisäksi varmistaa, että `a` ja `b` ovat *eri* puiden solmuja. Miten Männikön `union` toimii väärin, jos tämä varmistus jätetään tekemättä ja `a` ja `b` osoittavat samaan juurisolmuun? 38

Jos lauseet `pvJoukko[a] = k;` ja `pvJoukko[b] = a;` olisivat vastakkaisessa järjestyksessä, niin mitä `union( a, b )` tekisi, jos `a` ja `b` osoittavat samaan juurisolmuun? 39

Alkiovieraiden osajoukkojen eli union-find-tietorakenteeseen voidaan lisätä osajoukon alkioiden läpikäynti (esimerkiksi osajoukon alkioiden tulostaminen) osajoukon kokoon verrannollisessa ajassa siten, että muiden toimintojen asymptoottinen aikavaativuus ei huonone. Läpikäynti saa tapahtua missä tahansa järjestyksessä. Sitä varten lisätään jokaiseen alkioon yksi linkki (osoitin, indeksi tai muu sellainen). Miten tämä linkki pitää alustaa, ja mitä sille tehdään toiminnoissa union ja find? Jos et keksi, katso ensin tehtävä 20! 40

Virittävä puu. Tehtävissä 41, ..., 50 toteutetaan Primin mahdollisimman lyhyen virittävän puun algoritmi tavalla, joka on hiukan hitaampi mutta huomattavasti yksinkertaisempi kuin nopein tunnettu. Tapa sallii käyttää monissa ohjelmointikielissä valmiina olevaa prioriteettijonoa. Algoritmi saa syötteekseen yhtenäisen suuntaamattoman graafin, jonka kaarilla on pituudet. Graafin solmut tunnetaan juoksevilla numeroilla ykkösestä alkaen. Pituudet ovat ei-negatiivisia kokonaislukuja. Algoritmin pitää tulostaa mahdollisimman lyhyen virittävän puun kaaret (ilman pituuksia). Algoritmi saa itse valita järjestyksen, jossa ne tulostetaan.

Primin algoritmi aloittaa löytämällä jonkin solmun (minkä tahansa). Sitten se valitsee mahdollisimman lyhyen kaaren, jonka toinen pää on ja toinen pää ei ole löydetty. Algoritmi tulostaa kaaren ja löytää sen päissä olevista solmuista sen, jota ei ole vielä löydetty. Algoritmi toistaa tätä riittävän kauan. Älä vielä mieti

solmun löytämisen yksityiskohtia. Minkälaisia tietorakenteita tämän kuvauksen perusteella tarvitaan? 41

Miten prioriteettijonon osalta kannattaa menetellä niiden kaarien kanssa, joiden kumpikaan pää ei ole vielä löydetty? 42

Jos prioriteettijonoon laitetaan kaari, jonka yksi pää on löydetty mutta toinen pää ei, niin voi käydä niin että toinen pää löytyy kaaren ollessa prioriteettijonossa. Ohjelmointikielten valmiit prioriteettijonot eivät tarjoa nopeaa keinoa poistaa prioriteettijonosta muuta kaarta kuin ensimmäisen. Miten prioriteettijonon osalta kannattaa menetellä niiden kaarien kanssa, joiden molemmat päät on löydetty? 43

Kirjoita Primin algoritmi aliohjelmana `Prim`. Oleta, että solmut ovat taulukossa `solmut`, jota indeksoidaan solmun numerolla. (`solmut[0]` jätetään käyttämättä.) Kaaritietueen kentät `eka` ja `toka` sisältävät kaaren päissä olevien solmujen numerot. Solmu `eka` on varmasti löydetty. Aliohjelma `loyda_solmu( numero )`, joka suunnitellaan myöhemmin, tekee kaiken mitä solmulle tarvitsee tehdä kun se löytyy. Prioriteettijonon nimi on `lyhin_ensin`. Laita algoritmi lopettamaan kun `lyhin_ensin` on tyhjä. Mietimme myöhemmin, onko tämä hyvä lopetusehto. 44

Primin algoritmi voisi lopettaa jo silloin, kun viimeinenkin solmu on löydetty. Se olisi helppo toteuttaa ylläpitämällä löydettyjen solmujen määrää. Se ei kuitenkaan nopeuttaisi algoritmia olennaisesti, ja se olisi vähemmän vikasietoinen silloin, kun syötteenä annettu graafi ei olekaan yhtenäinen. Mitä vastaus 44 tekee, jos annettu graafi ei ole yhtenäinen? Mitä se tekisi, jos lopetusehtona olisi, että kaikki solmut on löydetty? 45

Graafin yhtenäisyyden tarkastaminen etukäteen on paljon monimutkaisempaa kuin Primin algoritmin ajaminen kunnes `lyhin_ensin` on tyhjä. Siksi voi olla sovellusten kannalta edullista, että Primin algoritmi ei sekoa jos sille annettu graafi ei olekaan yhtenäinen, vaan kertoo että syöte oli virheellinen. Miten vastaus 44 saadaan kertomaan, että syöte oli virheellinen? 46

Aliohjelman `loyda_solmu` kirjoittamista varten tarvitsee suunnitella solmujen esitystapaa nykyistä pitemmälle. Sitä varten on hyödyllistä kuvailla sanallisesti, mitä `loyda_solmu` tekee. Mitä se tekee? 47

Mitä tietoja solmutietueessa kannattaa olla? 48

Kirjoita `loyda_solmu`! 49

Riippuu ohjelmointikielestä, miten prioriteettijono otetaan käyttöön niin että ensimmäisenä on aina mahdollisimman lyhyt kaari. C++:ssa se voidaan tehdä seuraavasti. Samalla näytetään kaaritietue kokonaisuudessaan ja jokainen ohjelman tarvitsema `#include`.

```
#include <iostream>
#include <vector>
#include <queue>

struct kaarityyppi{
```

```
  unsigned eka, toka, pituus;
  kaarityyppi( unsigned eka, unsigned toka, unsigned pituus ):
    eka( eka ), toka( toka ), pituus( pituus )
  {}
  bool operator<( const kaarityyppi & toinen ) const {
    return pituus > toinen.pituus;  // näin päin jotta lyhin tulisi ekaksi
  }
};

std::priority_queue< kaarityyppi > lyhin_ensin;
```

Miten kaari voidaan lisätä tietorakenteisiin, jos sen tiedot ovat `unsigned`-muuttujissa `eka`, `toka` ja `pituus`? Huomaa, että vastaus 44 luottaa siihen, että prioriteettijonosta saadun kaaren `eka` on jo löydetty! 50

Suuntaamattoman graafin *yhtenäinen komponentti* on mahdollisimman suuri aligraafi, jonka jokaisesta solmusta jokaiseen solmuun on polku. Jos suuntaamaton graafi koostuu useammasta kuin yhdestä yhtenäisestä komponentista, niin sillä ei ole virittävää puuta, mutta sen kullakin yhtenäisellä komponentilla on virittävä puu. *Virittävä metsä* on aligraafi, jossa on jokaisesta yhtenäisestä komponentista virittävä puu, eikä muuta. Miten Primin algoritmi saadaan etsimään virittävä metsä? 51

Kruskalin algoritmi muodostaa mahdollisimman lyhyen virittävän puun kokeilemalla kaaret yhden kerrallaan pituusjärjestyksessä mahdollisimman lyhyestä alkaen. Kaari lisätään muodostettavaan puuhun jos ja vain jos sen lisääminen ei aiheuta silmukkaa. Miten Kruskalin algoritmi saadaan pienellä muutoksella muodostamaan mahdollisimman lyhyt virittävä metsä? 52

Osoita vastaesimerkillä, että seuraava algoritmi mahdollisimman lyhyen virittävän puun etsimiseksi ei toimi aina oikein: Jos graafissa on vain yksi solmu, niin se yksinään muodostaa virittävän puun. Muussa tapauksessa graafi jaetaan kahteen mahdollisimman yhtäsuureen yhtenäiseen osaan, etsitään rekursiivisesti mahdollisimman lyhyt virittävä puu kummastakin osasta, ja yhdistetään ne mahdollisimman lyhyellä sellaisella kaarella, että sen toinen pää on toisessa ja toinen toisessa osassa. 53

Ahneet algoritmit, dynaaminen ohjelmointi ja lasketut muistava. Lanttilan rahayksikkö on hilu. Lanttilassa käytettävien kolikoiden arvot ovat kokonaislukuja. Ne on kerrottu aidosti kasvavassa järjestyksessä taulukossa `arvot`, ja `arvot[0]` $= 1$.

Ahne algoritmi muodostaa rahasumman ottamalla mukaan aina suurimman kolikon, joka vielä mahtuu. Jos esimerkiksi `arvot` $= [1, 6, 20, 100]$, niin muodostaakseen 222 hilua, ahne algoritmi ottaa ensin yhden ja sitten toisen 100 hilun

kolikon. Sen jälkeen 100 hilun kolikko ei enää mahdu mukaan, mutta 20 hilun kolikko mahtuu, joten ahne algoritmi ottaa sellaisen.

Kirjoita ahne ohjelmanpätkä, jolle annetaan muodostettava summa muuttujassa `tavoite` ja joka kertoo mukaan otettavat kolikot! Vastaus talletetaan taulukkoon `maarat`, joka on alustettu tarpeeksi suureksi ja täyteen nollaa. 54

Osoita esimerkillä, että ahne algoritmi ei välttämättä johda pienimpään määrään kolikoita! 55

Kirjoita rekursiivinen aliohjelma, joka selvittää, mikä on pienin määrä kolikoita, jolla `tavoite` voidaan muodostaa. Se saa olla hyvin hidas. Sen ei tarvitse täyttää taulukkoa `maarat` (se lisätään seuraavassa tehtävässä). 56

Lisää edellisen tehtävän vastaukseen taulukon `maarat` täyttäminen! 57

Edellinen ohjelma on hidas jo pienehköilläkin tavoitteilla, kuten 100. Nopeuden parantamiseksi lisää siihen jo laskettujen vastausten tallettaminen! 58

Tee ohjelmasta dynaamisen ohjelmoinnin mukainen, ei-rekursiivinen versio! 59

Miten editointietäisyyden algoritmia pitää muuttaa, jos merkin lisääminen maksaa kolme yksikköä ja poistaminen maksaa kaksi yksikköä? 60

Rekursiivinen (ei muistava) editointietäisyyden algoritmi muuntaa $A[1 \ldots i]$:n $B[1 \ldots j+1]$:ksi x kertaa, $A[1 \ldots i+1]$:n $B[1 \ldots j]$:ksi y kertaa ja $A[1 \ldots i+1]$:n $B[1 \ldots j+1]$:ksi z kertaa. Kuinka monta kertaa se muuntaa $A[1 \ldots i]$:n $B[1 \ldots j]$:ksi, kun

$$A[i \ldots i+1] = \texttt{"kl"} \text{ ja } B[j \ldots j+1] = \texttt{"kt"} \text{ ?} \quad 61$$

$$A[i \ldots i+1] = \texttt{"ee"} \text{ ja } B[j \ldots j+1] = \texttt{"he"} \text{ ?} \quad 62$$

Muistava algoritmi sisältää tietorakenteen, jossa on jo lasketut vastaukset. Algoritmi katsoo sieltä, onko vastaus jo laskettu. Jos kyllä, se palauttaa talletetun vastauksen. Jollei, se laskee vastauksen ja tallettaa ja palauttaa sen. Kuinka paljon tällainen muutos algoritmeihin `plus` ja `kerto` pienentäisi tehtävän 12 vastausta, jos tallennusrakenteet ovat alun perin tyhjät? 63

Lähin pistepari. Lähimmän pisteparin algoritmissa pisteet jaetaan x-koordinaatin mukaan kahteen mahdollisimman yhtäsuureen joukkoon. Miksi joukkoa, johon piste kuuluu, ei välttämättä pystytä päättelemään pelkästään pisteen x-koordinaatista? 64

Avuksi voidaan ottaa y-koordinaatti. Oletetaan vähän aikaa, että jos kahdella eri pisteellä on sama x-koordinaatti, niin niillä on varmasti eri y-koordinaatti. Olettaen, että ensimmäinen oikeanpuoleiseen joukkoon kuuluva piste on `raja`, kirjoita ohjelmanpätkä, joka palauttaa `true` tai `false` sen mukaan kuuluuko `piste` oikeanpuoleiseen joukkoon. Tarvittavat koordinaatit saadaan lausekkeilla `piste.x`, `piste.y`, `raja.x` ja `raja.y`.! 65

Nyt ei enää luvata, että jos kahdella eri pisteellä on sama x-koordinaatti, niin niillä on varmasti eri y-koordinaatti. Mitä algoritmi kannattaa laittaa tekemään siinä tilanteessa, että viimeisellä vasemmanpuoleisen joukon pisteellä on sekä sama x-koordinaatti että sama y-koordinaatti kuin ensimmäisellä oikeanpuoleisen joukon pisteellä? 66

Lähimmän pisteparin algoritmin kukin rekursiotaso tarvitsee käsittelemänsä pisteet sekä ensisijaisesti x-koordinaatin ja toissijaisesti y-koordinaatin mukaan järjestettyinä, että ensisijaisesti y-koordinaatin mukaan järjestettyinä. Olettaen, että ne ovat taulukoissa `xJarj` ja `yJarj`, kirjoita ohjelmanpätkä, joka tuottaa rekursiivisia kutsuja varten taulukot `xJarjVasen`, `yJarjVasen`, `xJarjOikea` ja `yJarjOikea`! 67

O- Ω- ja Θ-merkintä. Miksi jokainen järjestämisalgoritmi on nopeimmillaankin $\Omega(n)$? 68

Voiko mikään käytännössä hyödyllinen algoritmi olla nopeimmillaan nopeampi kuin $\Omega(n)$? 69

Miten mikä tahansa järjestämisalgoritmi saadaan muutettua sellaiseksi, että se on nopeimmillaan $\Theta(n)$? Kannattaako sellainen muutos tehdä? 70

Tapana on, että O-, Ω- ja Θ-merkinnöissä sulkeiden sisään kirjoitetaan mahdollisimman yksinkertainen lauseke, jolla saadaan ilmaistua se mitä halutaan. Esimerkiksi yleensä ei kirjoiteta $O(3n^2+2)$ vaan $O(n^2)$, sillä ne tarkoittavat samaa, mutta n^2 on yksinkertaisempi kuin $3n^2+2$. Jossain tilanteessa voi olla erityinen syy kirjoittaa tarpeettoman monimutkainen lauseke. Jollei sellaista syytä ole, niin kirjoita niin yksinkertainen lauseke kuin osaat, jolla merkintä saa sen kasvuvauhdin jonka haluat ilmaista.

Koska O-merkintä ilmaisee kasvuvauhdille ylärajan, on esimerkiksi jokainen funktio, joka on $O(n^2)$, myös $O(n^3)$, $O(n^4)$, $O(n^2 \log n)$ ja niin edelleen. Jatkossa ”mahdollisimman tarkalla” O-merkinnällä tarkoitetaan O-merkintää, joka ilmaisee niin hitaan kasvuvauhdin kuin mahdollista. Esimerkiksi $3n^2+2n+5$ on sekä $O(n^3)$ että $O(n^2)$, se ei ole $O(n)$, ja mahdollisimman tarkka O-merkintä sille on $O(n^2)$.

Hyvin usein, vaikka ei ihan aina, lausekkeesta saadaan mahdollisimman tarkka mahdollisimman yksinkertainen O-merkintä yhdistämällä samanmuotoiset termit, poimimalla lausekkeen eniten merkitsevä eli nopeimmin kasvava termi ja jättämällä vakiokerroin pois. Ilmaise seuraavat funktiot mahdollisimman tarkalla O-merkinnällä.

$$f(n) = 1000 + 100n + 10n^2 + n\log n \quad 71$$

$$f(n) = 2n^2 - 3n + 7 + 8n - 2n^2 \quad 72$$

Ilmaise seuraavat funktiot mahdollisimman tarkalla O-merkinnällä. Luettele kaikki sellaiset parit (i, j), että $f_i(n) = O(f_j(n))$.

$$f_1(n) = n^2 - 3n + 5 \qquad f_4(n) = 3 + n + 12\log n - 2 + 4n$$

$$f_2(n) = 2n + 4 + 5n\log n \qquad f_5(n) = \tfrac{1}{2}n(n+1)$$

$$f_3(n) = 6 + 2\sqrt{n} + \tfrac{1}{3}n^3 + 8n\log n \qquad f_6(n) = n\sqrt{n} + n\log n$$ 73

Vastavaasti ”mahdollisimman tarkalla” Ω-merkinnällä tarkoitetaan Ω-merkintää, joka ilmaisee niin nopean kasvuvauhdin kuin mahdollista. Esimerkiksi $3n^2 + 2n + 5$ on sekä $\Omega(n)$ että $\Omega(n^2)$, se ei ole $\Omega(n^3)$, ja mahdollisimman tarkka Ω-merkintä sille on $\Omega(n^2)$.

Θ-merkinnän yhteydessä ei käytetä ilmausta ”mahdollisimman tarkka”, koska Θ-merkintä on aina mahdollisimman tarkka. Esimerkiksi $3n^2 + 2n + 5$ ei ole $\Theta(n^3)$ eikä $\Theta(n)$, mutta on $\Theta(n^2)$. Funktio on $\Theta(f(n))$ jos ja vain jos se on sekä $O(f(n))$ että $\Omega(f(n))$.

Miksi algoritmien suoritusaikaa tai muistin käyttöä kuvaavissa lausekkeissa ja O-, Ω- ja Θ-merkinnöissä saa käyttää osalauseketta $\log n$, vaikka se ei ole määritelty kun $n = 0$? 74

Funktiolla on O-merkintä jos ja vain jos ei ole olemassa loputtomasti toinen toistaan suurempia n:n arvoja, joilla funktio on määrittelemätön tai saa negatiivisen arvon. Sama pätee Ω-merkintään. Esimerkiksi funktiolla $n^2 - 3n + 1$ on O-merkintä vaikka se saa negatiivisen arvon kun $n = 1$ tai $n = 2$, sillä se ei saa negatiivista arvoa millään $n \geq 3$. Toisaalta funktiolla $3n - n^2 + 100$ ei ole O-merkintää, koska se saa negatiivisen arvon aina kun $n \geq 12$.

Kaikilla syötteillä pysähtyvän ohjelman suoritusaika ja muistin käyttö eivät voi olla negatiivisia eivätkä määrittelemättömiä. Niinpä niillä on aina O-merkintä ja Ω-merkintä. Siksi emme kiinnitä tämän enempää huomiota funktioihin, joille niitä ei ole olemassa.

Usein mahdollisimman tarkka O-merkintä voidaan esittää yksinkertaisesti, mutta ei aina. Jos funktio poukkoilee ylös ja alas, niin mahdollisimman tarkka O-merkintä saattaa olla monimutkainen. Seuraava algoritmi havainnollistaa tätä. Se palauttaa true tai false sen mukaan, onko täsmälleen puolet taulukon $A[1 \ldots n]$ alkioista nollia.

```
if n mod 2 = 1 then return false
m := 0
for i := 1 to n do
    if A[i] = 0 then m := m + 1 else m := m − 1
if m = 0 then return true else return false
```

Ensimmäinen rivi nopeuttaa algoritmin toimintaa silloin kun A:n koko on pariton. Parittomasta määrästä täsmälleen puolet ei ole kokonaisluku, jolloin täsmälleen puolet alkioista ei voi olla nollia. Algoritmin suoritusaika muistuttaa funktiota

$$3(1^n+(-1)^n)(n+\tfrac{1}{2})+3 \;=\; \begin{cases} 3 & \text{kun } n \text{ on pariton} \\ 6n+6 & \text{kun } n \text{ on parillinen} \end{cases}$$

Mahdollisimman yksinkertainen tarkka O-merkintä sille on $O((1^n+(-1)^n)n+1)$. Se ei ole kovin helppo tulkita.

Ylös ja alas poukkoilun tarkka seuranta on algoritmien analyysissä hyödyllistä erittäin harvoin, joten yleensä pyritään antamaan mahdollisimman tarkka *kasvava* O-merkintä, eli sellainen, jossa sulkeiden sisällä on kasvava funktio. Funktio on kasvava, jos ja vain jos n:n kasvaessa funktion arvo ei koskaan pienene. Funktiot $\log n$, $\sqrt{n}$, n, $n\log n$, n^2 ja niin edelleen ovat kasvavia. Myös vakiofunktio 1 on kasvava, sillä senkään arvo ei koskaan pienene. Matematiikassa sana ”kasvava” ei vaadi, että funktion arvo koskaan kasvaa.

Ilmaise funktiolle $3(1^n+(-1)^n)(n+\frac{1}{2})+3$ yläraja mahdollisimman tarkalla kasvavalla O-merkinnällä ja alaraja mahdollisimman tarkalla kasvavalla Ω-merkinnällä! 75

Voiko tämän funktion kasvunopeuden ilmaista kasvavalla Θ-merkinnällä? 76

Suoritusajan suuri poukkoilu ylös ja alas n:n kasvaessa ei ole kovin yleinen ilmiö. Sen sijaan on yleistä, että suoritusaika tai muistin kulutus riippuu ratkaisevasti siitä, mikä erilaisista saman kokoisista syötteistä on kyseessä. Sen vuoksi puhutaan *huonoimmasta tapauksesta*, *parhaasta tapauksesta* ja *jokaisesta tapauksesta*. Vaikka sana ”tapaus” saattaa kuulostaa yhdeltä nimenomaiselta syötteeltä, se ei tässä yhteydessä ole yksi syöte vaan kokoelma, jossa on loputtomasti toinen toistaan isompia syötteitä. Esimerkiksi jos algoritmin syöte on taulukko $A[1\ldots n]$, niin tyhjä taulukko ei ole tapaus (koska se on vain yksi syöte), mutta taulukko, jonka jokainen alkio on 1, on tapaus (koska on olemassa nollan, yhden, kahden ja niin edelleen alkion taulukko, jonka jokainen alkio on 1).

Jos algoritmi pysähtyy kaikilla syötteillä, niin suoritusajan ja muistin kulutuksen huonoimmalla, parhaimmalla ja jokaisella tapauksella on mahdollisimman tarkat kasvavat O- ja Ω-merkinnät. Tehtävän 76 vastauksesta seuraa, että kasvavaa Θ-merkintää ei välttämättä ole. Jos algoritmi pysähtyy kaikilla syötteillä, niin huonoimmalle ja parhaalle tapaukselle on Θ-merkintä (joka ei välttämättä ole kasvava), mutta jokaiselle tapaukselle ei välttämättä ole. Jos sama funktio kelpaa sekä huonoimman tapauksen O-merkintään että parhaan tapauksen Ω-merkintään, niin se kelpaa jokaisen tapauksen Θ-merkintään (ja päinvastoin).

Arvioi seuraavien algoritmien jokaisen tapauksen suoritusaikaa n:n funktiona O-, Ω- ja Θ-merkinnällä. Huomaa, että saattaa olla mahdotonta antaa kaikki kolme.

$i := 1$; **while** $i \le n$ && $A[i] \neq x$ **do** $i := i+1$ 77

$i := 1$; **while** $i < n$ **do** $i := 2i+1$ 78

$z := n$
while $z > 1$ **do**

```
y := z − 1; z := 0
for i := 1 to y do
    if A[i] > A[i+1] then t := A[i]; A[i] := A[i+1]; A[i+1] := t; z := i
```

79

Hitaimman tapauksen suoritusaika antaa usein hyvän kuvan suoritusajasta käytännössä. Toisinaan kuitenkin hitain tapaus on niin harvinainen, että sen käytännön merkitys on vähäinen. Siksi toisinaan ollaan kiinnostuneita *keskimääräisen tapauksen* suoritusajasta.

Keskimääräisestä tapauksesta puhutaan kuitenkin huomattavasti harvemmin kuin hitaimmasta tapauksesta, koska siihen liittyy kaksi vaikeutta.

Ensiksi, sen käyttö edellyttää, että eri syötteille tiedetään todennäköisyydet. Jos järjestettävän taulukon kaikki alkiot ovat erisuuret, niin on luonnollista olettaa, että jokainen alkuperäinen järjestys on yhtä todennäköinen. Mutta jos alkioissa saa olla myös yhtäsuuria, niin vastaavaa luonnollista oletusta ei ole. Esimerkiksi jos kahden alkion taulukon alkiot arvotaan tasajakaumalla joukosta $\{0,1\}$, niin ne ovat yhtäsuuret todennäköisyydellä $\frac{1}{2}$, mutta jos ne arvotaankin joukosta $\{0,1,2\}$, niin ne ovat yhtäsuuret todennäköisyydellä $\frac{1}{3}$.

Toiseksi, jos jakauma on annettu, niin sen mukainen keskimääräinen suoritusaika voi olla hyvin vaikea laskea.

2 Pisimmät sanat ja kaikki kokonaisluvut

Tämän esimerkin aiheena on tekstimuotoisen syötteen lukeminen ja hieman myös tekstialkioiden tunnistaminen. Syötteessä voi olla ääkkösiä. Kehitämme ohjelmaa vähän kerrassaan ja kokeilemme joka välissä muutamalla syötteellä, toimiiko viimeksi toteutettu piirre. Emme vielä tässä esimerkissä yritä testata ohjelmaa huolellisesti.

2.1 Tehtävän kuvaus

Tehtävänä on laatia ohjelma, joka lukee UTF-8-koodatun tekstin ja etsii kunkin tekstikappaleen pisimmän sanan sekä pienimmän tekstin seasta löytyvän kokonaisluvun.

Sana on mikä tahansa mahdollisimman pitkä epätyhjä jono muita merkkejä kuin välilyöntejä ja rivinsiirtoja. ”Mahdollisimman pitkä” tarkoittaa tässä yhteydessä, että esimerkiksi `voi ei!` sisältää vain kaksi sanaa `voi` ja `ei!`, eivätkä `v`, `o`, `i`, `vo`, `oi` ja niin edelleen ole sen sanoja. *Kokonaisluku* on mikä tahansa mahdollisimman pitkä jono, jossa on vapaaehtoinen etumerkki ja välittömästi sen perässä epätyhjä jono numeromerkkejä. *Etumerkki* on `+` tai `-`. Numeromerkit ovat `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` ja `9`. Kokonaisluku voi olla sana tai osa sanaa. Ohjelma saa luottaa siihen, että kokonaisluvun lukuarvo mahtuu ohjelmointikielten

tavanomaiseen kokonaislukutyyppiin. *Tekstikappale* on mikä tahansa mahdollisimman pitkä epätyhjä jono epätyhjiä rivejä. Rivi on tyhjä jos ja vain jos siinä ei ole yhtään merkkiä, ei edes välilyöntejä. Onko jokaisessa tekstikappaleessa ainakin yksi sana? 80

Kunkin tekstikappaleen, jossa on ainakin yksi sana, pisimmästä sanasta pitää tulostaa rivi muotoa

```
3,28: sana
```

missä luvut ovat rivi ja sarake jolla sana esiintyy. Rivit ja sarakkeet numeroidaan ykkösestä alkaen. Jos tekstikappaleen pisin sana ei ole yksikäsitteinen, niin tulostetaan ensimmäisenä esiintyvä. Lopuksi tulostetaan pienin koko tekstissä esiintyvä kokonaisluku samassa muodossa tai rivi, jossa lukee `ei kokonaislukuja`. Jos pienin kokonaisluku esiintyy useasti, raportoidaan sen ensimmäinen esiintymä.

Tulostuksessa ei saa olla ylimääräisiä välilyöntejä, rivinsiirtoja tai mitään muutakaan ylimääräistä. Kaikki kokonaisluvut tulostetaan tavallisessa esitystavassa. Siis nolla tulostetaan `0`, muut kokonaisluvut tulostetaan ilman etunollia, ja etumerkki tulostetaan vain jos se on `-`.

UTF-8 on sikäli nerokas koodaus, että vaikka ohjelmoija ei tekisi mitään ottaakseen huomioon että se on käytössä, monet asiat toimivat silti oikein. Siksi voimme joksiksin aikaa unohtaa että käytössä on UTF-8, mutta voimme silti käyttää ääkkösiä. Palaamme UTF-8:aan luvussa 2.5.

2.2 Syötteen lukeminen

Toteutamme ensin yhden rivin lukemisen ja tulostamisen.

C++:ssa voi lukea rivin syötteestä merkkijonomuuttujaan `rivi` lauseella `std::getline( std::cin, rivi );`. Rivin lopettava rivinsiirto ei tule mukaan. Muuttujan `rivi` sisällön voi tulostaa lauseella `std::cout << rivi;`. Jotta syötteen lukeminen ja tulostaminen olisivat ylipäänsä mahdollisia, jossakin sopivassa kohdassa ohjelmaa pitää lukea `#include <iostream>`. Muuttuja `rivi` pitää esitellä ennen käyttöä. Sen voi tehdä lauseella `std::string rivi;`. Merkkijonot eivät kuulu C++:n ydinkieleen, joten niiden käyttöön saamiseksi pitää kirjoittaa `#include <string>`.[2] Kun vielä lisätään pääohjelman alku ja loppu, saadaan seuraava:

```
#include <iostream>
#include <string>
```

[2]Merkkijonot saattavat tulla käyttöön jo komennolla `#include <iostream>`. Siihen ei kuitenkaan voi luottaa, joten on suositeltavaa kirjoittaa `#include <string>` vaikka olisi jo kirjoittanut `#include <iostream>`.

```
int main(){
  std::string rivi;
  std::getline( std::cin, rivi );
  std::cout << rivi;
}
```

Koeajo syötteellä `yksi rivi` tuotti muuten oikean tuloksen, mutta ohjelman lopetettua kursori ei siirtynyt uudelle riville vaan jäi sanan `rivi` perään. Sen korjaamiseksi lisäämme tulostuksen loppuun rivinsiirron: `std::cout << rivi << "\n";`.

Seuraavaksi muutamme ohjelmamme lukemaan ja tulostamaan syötteen kaikki rivit. Sen voi tehdä monella tavalla. Yksi tapa perustuu siihen, että C++:n lukemistoiminnon voi kirjoittaa ehdon paikalle. Silloin ehdon tulokseksi tulee `true` tai `false` sen mukaan onnistuiko lukeminen. Niin saadaan seuraava lyhyt ohjelmanpätkä:

```
std::string rivi;
while( std::getline( std::cin, rivi ) ){ std::cout << rivi << "\n"; }
```

Emme kuitenkaan valitse tätä rakennetta, sillä se olisi hieman kömpelö jatkossa. Lisäksi se on joidenkin mielestä vaikeaselkoinen, koska syöterivin lukeminen on ikään kuin piilossa `while` :n ehdossa. Seuraavassa rakenteessa on syöterivin lukeminen erotettu testistä, onnistuiko lukeminen. Se perustuu siihen, että käyttämällä `std::cin` ehtona voi C++:ssa testata, onnistuiko edellinen lukutoiminto:

```
std::string rivi;
std::getline( std::cin, rivi );
while( std::cin ){
  std::cout << rivi << "\n";
  std::getline( std::cin, rivi );
}
```

Tässä rakenteessa `std::getline( std::cin, rivi );` esiintyy kahdesti. Lyhyen ohjelmanpätkän kahdentuminen ei ole iso ongelma, mutta ennemmin tai myöhemmin vastaan tulee tilanne, jossa kahdentuva ohjelmanpätkä olisi pitkä. Jos kahdennettua ohjelmanpätkää tarvitsee myöhemmin muuttaa, niin on vaarana, että muutos tulee tehtyä vain yhteen paikkaan ja jää vahingossa tekemättä toiseen paikkaan.

Kahdentumisen voi välttää hyödyntämällä sitä, että silmukasta voidaan poistua suorittamalla `break`.

```
while( true ){
  std::string rivi;
  std::getline( std::cin, rivi );
  if( !std::cin ){ break; }
  std::cout << rivi << "\n";
}
```

Koeaja ohjelma jollakin monirivisellä syötteellä ja tarkasta, että tulos on oikein! Voit käyttää syötteenä vaikka ohjelmamme lähdekooditiedostoa. Ei haittaa, vaikka syötteessä olisi ääkkösiä.

2.3 Sanojen alku- ja lopunohikohdat

Seuraavaksi lisäämme ohjelmaan yksittäisten sanojen erottamisen syötteestä. Sitä varten lisäämme muuttujan tarkoittamaan paikkaa merkkijonossa `rivi`.

Koska rivin alussa paikka on 0 ja muualla positiivinen, ei paikka saa negatiivisia arvoja. Niinpä luontevin tyyppi sille on etumerkitön kokonaisluku. Mutta etumerkillinen kokonaisluku kelpaa tässä tapauksessa ja monissa muissakin tapauksissa etumerkittömän tilalle. Siksi osa ohjelmoijista ei käytä etumerkittömiä kokonaislukuja lainkaan. Valitse itse, kumpaa haluat käyttää.

```
unsigned paikka = 0;  // sijainti viimeksi luetulla rivillä
```

Sanan alku löytyy ohittamalla välilyönnit. Pitää kuitenkin varoa lukemasta rivin lopun ohi. Ohjelmamme alkaa olla jo niin pitkä, että kannattaa lisätä kommentteja kertomaan mitä mikäkin kohta tekee.

```
/* Etsi seuraavan sanan alku. */
while( paikka < rivi.size() && rivi[ paikka ] == ' ' ){ ++paikka; }
unsigned alku = paikka;
```

Suurin osa ohjelmoijista kirjoittaisi `paikka++` eikä `++paikka`. Erolla ei tässä eikä monissa muissakaan tapauksissa ole mitään merkitystä. Silti, niissä tapauksissa joissa kumpikin vaihtoehto saa ohjelman toimimaan oikein, on lievästi suositeltavaa ottaa tavaksi kirjoittaa `++` kohteensa eteen eikä perään.[3]

Monessa ohjelmointikielessä on tapana käyttää loppukohdan sijaan sitä seuraavaa kohtaa. Kutsumme sitä *lopunohikohdaksi*. Sanan lopunohikohta löytyy ohittamalla muut merkit kuin välilyönnit. Nytkin pitää varoa lukemasta rivin lopun ohi. Kirjoita ohjelmanpätkä! 81

Jotta näkisimme että sana on oikein rajattu, laitamme ohjelmamme tulostamaan sen.

```
/* Tulosta sana. */
std::cout << rivi.substr( alku, paikka - alku ) << "\n";
```

Poista aiemmin lisätty koko rivin tulostus `std::cout << rivi << "\n";`. Koeaja ohjelmaa muutamalla syötteellä, mukaan lukien alla oleva. (Älä lisää rivien alkuun välilyöntejä, vaikka rivit onkin esteettisyyssyistä sisennetty tässä tekstissä.) Riippumatta siitä näyttääkö ohjelma toimivan, lue eteenpäin.

[3] Jos `muuttuja` on monimutkaista tyyppiä, niin `muuttuja++` voi mutta `++muuttuja` ei voi aiheuttaa `muuttuja`:n alkuperäisen arvon kopioinnin tilapäiseen muuttujaan. Se on turhaa työtä.

```
yksi
ohjelma
```

Edellä ei kerrottu, mihin `unsigned paikka = 0;` piti laittaa. Jos se laitettiin pääohjelman alkuun, niin mitä ohjelma tulostaa edellä olevalla syötteellä? 82

Miksi se tulostaa niin? 83

Mihin `unsigned paikka = 0;` pitää laittaa? 84

Siirrä `unsigned paikka = 0;` oikeaan paikkaan, jollei se jo ole siellä!

Tähänastinen ohjelma tulostaa vain kunkin rivin ensimmäisen sanan. Se täytyy yleistää tulostamaan kunkin rivin kaikki sanat. Miksi seuraava silmukka ei tee sitä oikein edes aina silloin, kun syötteessä on vain yksi rivi? Jollet keksi, niin yritä löytää vika testaamalla ohjelmaa muutamalla syötteellä. Jollet onnistu, niin älä masennu, sillä tätä vikaa voi olla vaikea löytää testaamalla! 85

```
/* Selaa rivi. */
while( paikka < rivi.size() ){
  /* Etsi seuraavan sanan alku. */
  ...  // edellä kerrotut rivit
  std::cout << rivi.substr( alku, paikka - alku ) << "\n";
}
```

Suunnittele silmukka, joka käsittelee oikein koko rivin, ja testaa ohjelmaa! 86

Lisäämme vielä kommentit pääsilmukan eteen sekä rivin esittelyn ja lukemisen eteen.

```
/* Lue syöte. */
/* Lue rivi. Jos ei saatu, niin poistu silmukasta. */
```

2.4 Pisin syötteessä esiintyvä sana

Seuraavaksi muutamme ohjelmamme tunnistamaan pisimmän syötteessä esiintyvän sanan ja tulostamaan vain sen. Emme vielä ota huomioon sitä, että syöte on UTF-8-koodattu. Siksi ohjelmamme ei tarvitse toimia oikein, jos syötteessä on ääkkösiä.

Poista jokaisen sanan tulostus.

Otamme käyttöön muuttujan pisimmälle sanalle. Meidän ei tarvitse alustaa sitä, koska C++:ssa merkkijonomuuttujat alustetaan automaattisesti tyhjiksi. Mihin kohtaan sen esittely `std::string pisin_sana;` pitää laittaa, ja miksi? 87

Kirjoita seuraavan kommentin mukainen ohjelmanpätkä! 88

```
/* Jos löytyi pitempi sana kuin aiemmat, niin talleta se. */
```

Mikä menee väärin, jos juuri ennen ohjelman viimeistä `}` laitetaan `std::cout << pisin_sana << "\n";` ? 89

Kirjoita oikein toimiva pisimmän sanan tulostus, ja testaa! 90

Pisimmän sanan tulostukseen pitää lisätä rivin ja sarakkeen numerot, jotta tulostus olisi edellä vaadittua muotoa `3,28: sana`. Mitä lisäyksiä ja muutoksia ohjelmaan tämä edellyttää (älä vielä kirjoita ohjelmaan, siis vastaa sanallisesti)? 91

Tässä vaiheessa muuttujien nimien valitseminen alkaa vaikeutua. Ohjelmassa on jo muuttuja `rivi`, joka sisältää viimeksi luetun rivin merkkijonona. Nyt tarvitaan kaksi riviin viittavaa muuttujanimeä lisää. Jotta muuttujista näkyisi selvemmin mitä tietoja ne sisältävät, systematisoimme nimet. Niiden rivin ja sarakkeen numeron, joista pisin sana löytyi, nimiksi valitsemme `ps_rivi` ja `ps_sarake`, missä `ps_` vihjaa, että kyse on pisimmän sanan tiedoista. Samalla vaihdamme muuttujan `pisin_sana` nimeksi `ps_merkit`, jotta kaikkien pisimpään sanaan viittaavien muuttujien nimet alkaisivat `ps_`. Muuttujan `rivi` nimeksi vaihdamme `vr_merkit` (viimeksi luetun rivin merkit), ja viimeksi luetun rivin numeron tallettavan muuttujan nimeksi annamme `vr_rivi`. Muutosten seurauksena `rivi` ei enää viittaa rivin sisältöön vaan sen numeroon.

Muuta ohjelma käyttämään uusittuja muuttujanimiä, lisää siihen tehtävän 91 mukainen toiminnallisuus ja testaa. Millaisilla lauseilla pisimmän sanan sijainti saadaan muuttujiin `ps_rivi` ja `ps_sarake`, ja miksi lauseet eroavat toisistaan? 92

2.5 Ääkköset, € ja niin edelleen

Unicode on kansainvälinen standardi, jonka tavoitteena on kattaa suuri osa ihmisten käyttämistä kirjoitusmerkeistä, matemaattisista merkeistä ja monista muistakin merkeistä. Siihen sisältyy jopa hymiöitä. Unicode-merkkejä voidaan esittää tietokoneessa monella eri tavalla. Suosituin tapa on *UTF-8*. Se on vuodesta 2008 saakka ollut yleisin www-sivujen merkistökoodaus, ja maaliskuussa 2023 noin 98 % kaikista www-sivuista oli koodattu sillä. Se kykenee esittämään jokaisen Unicode-merkin. Siinä kukin merkki esitetään enintään neljällä 8-bittisellä tavulla.

ASCII on Yhdysvalloissa 1960-luvulla alkunsa saanut merkistökoodaus, joka kykenee esittämään isot ja pienet kirjaimet `a`:sta `z`:aan, numeromerkit, välilyönnin, rivinsiirron sekä pienehkön kokoelman välimerkkejä kuten `,` ja erikoismerkkejä kuten `+`, `-` ja `\`. Yhden ASCII-merkin esittämiseen tarvitaan 7 bittiä. ASCII:sta muunnelmineen tuli 1900-luvulla tietotekniikan vakiomerkistö. Useimmat ohjelmointikielet käyttävät komennoissaan vain sitä, vaikka voivatkin sallia merkkijonovakioissa ja -muuttujissa muitakin merkkejä.

Useimmissa nykyisissä tietokoneissa tieto organisoidaan 8-bittisiksi tavuiksi. ASCII-merkit esitetään tavuina, joiden eniten merkitsevä bitti on 0. Tämä jättää käyttämättä ne 128 tavua, joiden eniten merkitsevä bitti on 1. Niitä on käytetty vaihtelevilla tavoilla laajentamaan merkistöä.

Jokainen ASCII-merkkiä esittävä tavu on sellaisenaan saman merkin esitys myös UTF-8:ssa. Tavut, joiden kaksi, kolme tai neljä eniten merkitsevää bittiä

ovat 1 ja niitä seuraava bitti on 0 aloittavat UTF-8:ssa kaksi-, kolme- tai nelitavuisen koodin. Sen muiden tavujen eniten merkitsevä bitti on 1 ja toiseksi eniten merkitsevä on 0. Osa mahdollisista tavujen yhdistelmistä on jätetty käyttämättä teknisistä syistä tai tulevia laajennoksia varten.

Monille ohjelmille esimerkiksi välilyönti, rivinsiirto ja numeromerkit tarkoittavat sitä mitä ne tarkoittavat melkein aina muuallakin, mutta monet muut merkit ovat vain raakaa dataa, joka siirretään eteenpäin tulkitsematta sitä. Esimerkiksi voi olla, että henkilön nimeä käsitellään vain raakana datana, joka saadaan syötteestä, talletetaan henkilön muiden tietojen yhteyteen ja tulostetaan tarvittaessa, mutta jonka sisältöä ohjelma ei analysoi. Tällöin UTF-8-muotoisesta syötteestä saatu nimi `Björn` on ohjelman kannalta kuuden tavun jono — kuuden, koska `ö` esitetään kahdella tavulla ja `B`, `j`, `r` ja `n` kukin yhdellä. Kun tämä jono tulostetaan, niin jos myös tulosteen näyttävä ohjelma käyttää UTF-8:aa, niin tulosteessa näkyy `Björn`.

Siis UTF-8-koodatut merkit kulkevat tällaisen ohjelman läpi ja näkyvät lopputuloksessa oikein, vaikka ohjelman tekijä ei olisi tehnyt mitään UTF-8:n huomioon ottamiseksi. Riittää, että ohjelma ei yritä tulkita vaan ainoastaan välittää eteenpäin niitä merkkijonomuuttujissa esiintyviä tavuja, joiden eniten merkitsevä bitti on 1. Tällöin kuitenkin merkkijonojen pituudet ja merkkien sijainnit merkkijonoissa eivät ole ohjelman näkökulmasta samat kuin käyttäjän näkökulmasta, sillä käyttäjän näkökulmasta nimessä `Björn` on viisi merkkiä ja `r` on niistä neljäs, mutta ohjelman näkökulmasta siinä onkin kuusi merkkiä ja `r` on viides.

Niinpä lisäämme ohjelmaan sarakkeiden ja sanojen pituuden laskemisen käyttäjän näkökulmasta. Edellä kerrotun mukaan tavu on UTF-8-koodatun merkin ensimmäinen tavu jos ja vain jos sen kaksi eniten merkitsevää bittiä eivät ole 1 ja 0. Tämän perusteella on helppo kirjoittaa aliohjelma, joka testaa, onko tavu UTF-8-koodatun merkin ensimmäinen tavu. Alla on käytetty bittioperaatioita, mutta jos ne eivät ole tuttuja, niin voit niiden sijaan käyttää testiä, onko tavun numeroarvo tietyllä välillä. Tavujen, joiden kaksi eniten merkitsevää bittiä ovat 1 ja 0, numeroarvot riippuvat ohjelmointiympäristöstä. Joissakin ympäristöissä ne ovat vähintään 128 ja enintään 191, ja melkein kaikissa muissa ne ovat vähintään -128 ja enintään -65.

```
bool uc_alkutavu( char tavu ){ return ( tavu & 0xC0 ) != 0x80; }
```

Seuraavaksi lisäämme ohjelmaamme muuttujan `uc_paikka`, joka kertoo paikan viimeksi luetulla rivillä UTF-8-koodattuina Unicode-merkkeinä eikä tavuina laskettuna. Sen pitää käyttäytyä muuten samoin kuin `paikka`, mutta sen pitää kasvaa kunkin Unicode-merkin kohdalla vain kerran. Sitä varten etsimme ohjelmastamme kaikki muuttujan `paikka` esiintymät ja mietimme, mitä muuttujalle `uc_paikka` pitää tehdä siinä kohdassa.

Muuttujan `paikka` ensimmäinen esiintymä on syöterivin lukemisen lähettyvillä. Siellä se esitellään ja alustetaan nollaksi. Myös Unicode-merkkien laskeminen voidaan aloittaa rivin alussa nollasta, joten lisäämme muuttujan `uc_paikka` esittelyn ja alustuksen nollaksi.

```
unsigned paikka = 0, uc_paikka = 0;  // sijainti viimeksi luetulla rivillä
```

Seuraavaksi `paikka` esiintyy silmukassa, jossa ohitetaan välilyönnit. Siellä se esiintyy kolmesti. Mitä muuttujalle `uc_paikka` pitää tehdä tässä kohdassa? 93

Vähän myöhemmin on muuttujan `alku` esittely ja alustus. Mitä muuttujalle `uc_paikka` tulee tehdä, vai tarvitseeko tehdä mitään? 94

Mitä muuttujalle `uc_paikka` tulee tehdä siinä silmukassa, jossa etsitään sanan lopunohikohta? Kirjoita ohjelmanpätkä, joka tekee sen. 95

Miten ohjelmaa pitää vielä muuttaa, jotta se tulostaisi pisimmän sanan ja sen paikan Unicode-merkkien eikä tavujen mukaan laskettuna? Jos et keksi, niin aja ohjelmasi syötteellä `Väärä tulos, höpö!` ja mieti, mikä sen tulostuksessa on väärin! 96

Tee muutokset ja testaa.

2.6 Kunkin kappaleen pisin sana

Nyt muutamme ohjelman tulostamaan jokaisen kappaleen pisimmän sanan. Muutamme nykyisen pääohjelman käsittelemään vain yhden kappaleen ja vaihdamme sen nimen ja tyypin. Koska tehtävän kuvauksen (luku 2.1) mukaan kappaleet erottaa toisistaan ainakin yksi tyhjä rivi, laitamme aliohjelmaksi muuttuneen entisen pääohjelman lopettamaan myös saadessaan tyhjän syöterivin. Lisäämme loppuun uuden pääohjelman.

```
/* Käsittele yksi tekstikappale. */
void yksi_kappale(){
  ...
    if( !std::cin || vr_merkit.empty() ){ break; }
  ...
}

/* Pääohjelma: käsittele jokainen kappale. */
int main(){
  while( std::cin ){ yksi_kappale(); }
}
```

Ja taas testaamme. Näyttää toimivan hienosti, vai näyttääkö? 97

Ehdota korjausta! 98

Joko nyt kaikki on kunnossa? 99

Mikä aiheuttaa tämän vian? 100

Korjaa ja testaa.

2.7 Pienin kokonaisluku

Aloitamme pienimmän kokonaisluvun tulostamisen toteuttamisen lisäämällä tarvittavan tulostuslauseen pääohjelman loppuun. Jos ainakin yksi kokonaisluku saadaan, niin sopiva tulostuslause on

```
std::cout << pl_rivi << "," << pl_sarake << ": " << pl_arvo << "\n";
```

Jollei saada yhtään, niin `std::cout << "ei kokonaislukuja\n";` tulostaa mitä pitääkin.

Tarvitaan testi valitsemaan, kumpi tulostuslause suoritetaan. Yksi tunnettu keino on alustaa `pl_arvo` suurimpaan mahdolliseen kokonaislukutyypin arvoon, ja tulostaa `ei kokonaislukuja` jos ja vain jos `pl_arvo` :n arvo on yhä se. Sen vikana on, että jos tekstissä esiintyy suurin mahdollinen kokonaislukutyypin arvo mutta ei sitä pienempiä kokonaislukuja, niin ohjelma ei tulosta sitä, vaikka pitäisi. Luvussa 2.1 luvattiin, että tekstissä esiintyvät kokonaisluvut mahtuvat tavalliseen kokonaislukutyyppiin, mutta ei luvattu, että suurin mahdollinen kokonaislukutyypin arvo ei voisi esiintyä tekstissä.

Mutta on olemassa muu helppo keino selvittää tulostuslauseiden kohdalla, kumpi tulostuslause suoritetaan. Mikä se on? 101

Missä kohdassa ohjelmaa `pl_rivi` pitää esitellä? 102

Muuttujien `pl_sarake` ja `pl_arvo` esittelyjen sijainteihin vaikuttavat samat syyt kuin muuttujan `pl_rivi` , ja kohtasimme saman kysymyksen aikaisemmin `vr_rivi` :n yhteydessä. Kaikkien neljän laittaminen aliohjelman `yksi_kappale` parametreiksi olisi kömpelöä. Jossain toisessa ohjelmassa vastaava tarve voi olla vaikka kymmenelle muuttujalle, jolloin niiden laittaminen parametreiksi olisi vielä kömpelömpää.

Tähän on ohjelmointikielestä riippuen useita enemmän tai vähemmän tyylikkäitä ratkaisuja. Yksinkertaisinta on tehdä niistä globaaleja muuttujia, vaikka se onkin suosituksen vastaista. Sen huonona puolena on, että eri ohjelmoijat (tai jopa sama ohjelmoija) saattavat yrittää käyttää samaa nimeä ison ohjelman eri puolilla eri tehtävissä, jolloin syntyy helposti sekaannusta. Joissakin ohjelmointikielissä kuten C++ tämä voidaan ratkaista niin sanotulla nimiavaruudella. Se on keino rajoittaa mitkä nimet näkyvät ohjelman osasta ulos. Saman vaikutuksen saa jossain määrin työläämmin paketoimalla ohjelman osa olioksi. Vaihtoehtoisesti olioksi voidaan paketoida ne muuttujat, jotka tarvitsee välittää viiteparametrina, jolloin tarvitsee välittää monen muuttujan sijaan yksi olio.

Tämänkokoisessa ohjelmassa ei oikeasti ole tarvetta kontrolloida nimien näkyvyyttä, varsinkin jos nimet valitaan riittävän tarkasti kohteitaan kuvaaviksi. Etuliitteillä `ps_` , `vr_` , `uc_` ja `pl_` saatiin nimiin vihje siitä mihin niiden kohteet liittyvät, ja samalla pienennettiin huomattavasti riskiä, että jollekin toiselle muuttujalle annetaan sama nimi. Siksi valitsemme yksinkertaisimman ratkaisun, eli

teemme muuttujista `vr_rivi`, `pl_rivi`, `pl_sarake` ja `pl_arvo` globaaleja sijoittamalla niiden esittelyt ennen aliohjelmaa `yksi_kappale`.

```
/* Nykyinen rivi ja pienimmän löydetyn luvun tiedot */
unsigned vr_rivi = 0, pl_rivi = 0, pl_sarake = 0;
int pl_arvo = 0;
```

Pääohjelma näyttää nyt tältä.

```
/* Pääohjelma: käsittele jokainen kappale. */
int main(){

  /* Lue syöte ja tulosta kunkin kappaleen pisin sana. */
  while( std::cin ){ yksi_kappale(); }

  /* Jos saatiin ainakin yksi luku, niin tulosta pienin luku. */
  if( pl_rivi > 0 ){
    std::cout << pl_rivi << "," << pl_sarake << ": " << pl_arvo << "\n";
  }else{ std::cout << "ei kokonaislukuja\n"; }

}
```

Vielä on toteuttamatta kokonaislukujen tunnistaminen tekstin seasta ja pienimmän kohdatun kokonaisluvun ylläpitäminen. Ne tulevat silmukkaan, joka näyttää tällä hetkellä tältä:

```
/* Etsi sanan lopunohikohta. */
while( paikka < vr_merkit.size() && vr_merkit[ paikka ] != ' ' ){
  ++paikka;
  if( uc_alkutavu( vr_merkit[ paikka ] ) ){ ++uc_paikka; }
}
```

Ratkaisemme ensin pienimmän kohdatun kokonaisluvun ylläpitämisen. Olettaen että kohdatun luvun arvo on kokonaislukutyypisessä muuttujassa `lukuarvo` ja sen alkukohta muuttujassa `uc_luku`, kirjoita ohjelmanpätkä, joka tarvittaessa päivittää muuttujien `pl_arvo` ja niin edelleen arvot! 103

Käyttämässäsi ohjelmointikielessä saattaa olla valmis toiminto, jolla voi testata, onko merkki numeromerkki. Monissa kielissä sen nimi on `isDigit` tai jotain sen kaltaista. Tässä esimerkissä se on `on_numero( char merkki )`.

Merkkijonona esitetyn kokonaisluvun lukuarvon laskemiseksi on olemassa valmiita toimintoja, kuten Javassa `parseInt`. Ennen sellaisen kutsumista täytyy tietää kokonaisluvun alkukohta muuttujassa `vr_merkit`. Kirjoita ohjelmanpätkä, joka tuottaa `true` jos ja vain jos muuttujan `vr_merkit` käsittely saapui kokonaisluvun kohdalle! Voit olettaa, että `vr_merkit` ei ole vielä loppunut. 104

Mallivastaus 104 on kömpelö, koska `vr_merkit[ paikka ]` toistuu. Jos samaa toimintoa tarvitaan useassa kohdassa ohjelmaa, niin siitä kannattaa tehdä aliohjelma `on_numero_tai_etumerkki( char merkki )`. Tässä ohjelmassa sitä tarvitaan

vain yhdessä kohdassa. Siitä saadaan siistimpi ilman erillisen aliohjelman tekemistä kopioimalla `vr_merkit[ paikka ]` uuteen muuttujaan `merkki`.

Kirjoita silmukka `/* Etsi sanan lopunohikohta. */` uusiksi edellä olleiden ajatusten mukaisesti. Sen pitää testata ollaanko tultu kokonaisluvun kohdalle, ja sen perusteella joko käsitellä kokonaisluku tai käsitellä merkki kuten ennenkin. Siltä osin kuin emme vielä ole suunnitelleet kokonaisluvun käsittelyä, kirjoita `// Laske lukuarvo.` ! 105

Jotta rivin selaamista voitaisiin jatkaa kun kokonaisluku on käsitelty, täytyy tietää kokonaisluvun lopunohikohta. Se täytyy tietää myös Unicode-merkkeinä laskettuna. Olettaen, että `paikka` ja `uc_paikka` ilmoittavat kokonaisluvun alkukohdan, kirjoita ohjelmanpätkä, joka päivittää ne ilmoittamaan kokonaisluvun lopunohikohdan! 106

Javan `parseInt` tarvitsee argumentikseen merkkijonon, jossa on kokonaisluvun merkit eikä muuta. Se saadaan komennolla `substring`, joka ottaa argumenteikseen osajonon alku- ja lopunohikohdan. (Se on siis sikäli erilainen kuin C++:n `substr`, että jälkimmäisenä argumenttina ei ole merkkien määrä vaan lopunohikohta.) Kutsuksi tulee `parseInt( vr_merkit.substring( luvun_alku, paikka ) )`.

Jotta tästä esimerkistä ei tulisi Javan ja C++:n sekasotku, käytämme vähän aikaa merkintää `luvuksi( merkkijono, alku, lopun_ohi )` tarkoittamaan toimintoa, joka tuottaa merkkijonon osasta kokonaislukuarvon. Alla oleva ohjelmanpätkä on melkein mutta ei täysin toimiva keino selvittää lukuarvo annettavaksi tehtävän 103 vastauksen käsiteltäväksi. Mikä siinä on vikana? 107

```
unsigned luvun_alku = paikka, uc_luku = uc_paikka;
// tehtävän 106 vastaus
int lukuarvo = luvuksi( vr_merkit, luvun_alku, paikka );
```

Huomaamme, että valmiin toiminnon valjastaminen tuottamaan lukuarvo ei olekaan ihan yksinkertaista. Sitäpaitsi emme ole vielä tehneet kaikkea tarpeellista. Emme nimittäin ole tarkastaneet, onko käyttämämme valmis toiminto (esimerkiksi Javan `parseInt`) täsmälleen samaa mieltä kokonaisluvun syntaksista kuin luku 2.1. Jollei ole, niin ohjelmamme ei toimi täsmälleen vaatimusten mukaisesti. Ehkä suurin epäily kohdistuu etumerkkiin `+`, sillä se on käytännössä melko tarpeeton. Kenties Javan `parseInt` ei sallikaan sitä? Arvaa, tai jos jaksat niin selvitä Javan spesifikaatiosta, onko Javan `parseInt` täsmälleen samaa mieltä kokonaisluvun syntaksista kuin luku 2.1! 108

Niinpä saattaa loppujen lopuksi olla helpompaa toteuttaa luvun tuottaminen merkkijonosta itse. Vaikka ei olisikaan, on merkistökoodauksien ymmärtämisen kannalta hyödyllistä yrittää edes kerran tehdä se itse. Monissa ohjelmointikielissä `char`-tyyppisen arvon voi sellaisenaan sijoittaa `int`-tyyppiseen muuttujaan, ja tuloksena on merkin ASCII-koodin mukainen arvo. Esimerkiksi C++:ssa

```
char ch = 'A'; int nn = ch; std::cout << nn << " " << ch;
```

tulostaa `65 A`. Toisaalta `char ch = '€';` ei kelpaa kääntäjälle tai ei toimi odotetusti, koska € ei ole ASCII-merkki.

Sekä ASCII:ssa että UTF-8:ssa kukin numeromerkki muodostuu yhdestä tavusta, joka on ykkösen verran suurempi kuin edellisen numeromerkin tavu. Sen perusteella on helppo kirjoittaa totuusarvon tuottava funktio `on_numero( char merkki )` käyttämättä valmista toimintoa `isDigit` tai muu sellainen. Kirjoita sellainen! 109

Toisinaan suositellaan, että tällaista ei pidä toteuttaa itse vaan pitää käyttää valmista toimintoa, koska se säilyy toimivana jos merkistökoodaus muuttuu, ja koska se saattaa olla tehokkaampi kuin oma testi olisi. Ensimmäinen peruste on menettänyt merkityksensä, koska UTF-8 on erittäin vakiintunut. Sitäpaitsi luvussa 2.1 nimenomaan luvattiin, että syöte on UTF-8-koodattu. Mitä tehokkuuseroon tulee, kokeessani ohjelma, joka teki 4 miljardia testiä, vei vastausta 109 käyttäen 1,9 s, ja C++:n toimintoa `isdigit` käyttäen 2,1 s. Siis itse tehty oli nopeampi. Kumpikin versio on riittävän nopea melkein mihin tahansa tarpeeseen. Niinpä nykyisin voi käyttää valmista toimintoa tai itse tehtyä sen mukaan minkä kokee parhaaksi.

Millä laskutoimituksella saadaan ei-negatiiviseen kokonaislukuun nolla perään? Toisin sanoen, mikä laskutoimitus tuottaa esimerkiksi luvusta 211 luvun 2110, luvusta 3070 luvun 30700 ja niin edelleen? 110

Olkoon d yhdellä numeromerkillä esitettävissä oleva kokonaisluku, eli mikä tahansa kokonaisluku väliltä $0 \leq d \leq 9$. Millä laskutoimituksella saadaan nollaan loppuva ei-negatiivinen kokonaisluku muutettua muuten samaksi, mutta d:hen loppuvaksi kokonaisluvuksi? Toisin sanoen, jos esimerkiksi $d = 4$, niin mikä laskutoimitus tuottaa luvusta 2110 luvun 2114, luvusta 30700 luvun 30704 ja niin edelleen? 111

Kirjoita ohjelmanpätkä, joka tuottaa muuttujassa `merkit` kohdasta `paikka` alkaen sijaitsevasta numeromerkkien jonosta sitä vastaavan lukuarvon muuttujaan `tulos`. Muuttujan `paikka` pitää kasvaa numeromerkkien jonon lopunohikohtaan. Voit olettaa, että jono on epätyhjä ja että lukuarvo mahtuu tavalliseen kokonaislukutyyppiin! 112

Jos vastaukseen 112 lisätään etumerkkien käsittely ja tehdään siitä aliohjelma `hae_luku`, jossa `paikka` välitetään viite- tai in-out-parametrina, niin kysymyksessä 107 olleet rivit voidaan korvata seuraavilla, ja samalla niissä ollut vika korjaantuu. Aliohjelma kasvattaa parametrin `paikka` arvoa jos ja vain jos sen kohdalla merkkijonossa todella on kokonaisluku. Tehokkuussyistä merkkijono kannattaa välittää mekanismilla joka ei kopioi, kuten C++:n `const std::string & merkit`, jossa `&` tarkoittaa viiteparametria ja `const` (yrittää[4]) estää parametrin arvon muuttamisen.

[4]Kääntäjä ei salli parametriin `merkit` kohdistuvia toimintoja, jotka voivat muuttaa sen arvoa. Mutta jos kutsussa käytettyyn muuttujaan pääsee käsiksi jollain muulla tavalla, niin kääntäjä ei välttämättä tiedä sitä, eikä niin ollen voi estää arvon muuttamista sitä kautta. Jos esimerkik-

```
/* Yritä hakea lukuarvo. */
unsigned luvun_alku = paikka, uc_luku = uc_paikka;
int lukuarvo = hae_luku( vr_merkit, paikka );
if( paikka == luvun_alku ){ ++paikka; ++uc_paikka; continue; }
uc_paikka += paikka - luvun_alku;
```

Jos muuttujassa `vr_merkit` todella on kokonaisluku kohdasta `paikka` alkaen, niin `hae_luku` kasvattaa muuttujan `paikka` kokonaisluvun ohi. Koska `paikka` välitetään viite- tai in-out-parametrina, sen kasvu välittyy kutsujalle. Tehtävän 106 vastaus jää siis tarpeettomaksi. Koska jokainen etumerkki ja numeromerkki on ASCII-merkki, se on samalla yksitavuinen UTF-8-merkki, joten muuttujaa `uc_paikka` tulee kasvattaa saman verran kuin `paikka` kasvoi. Se tehdään edellä olevan ohjelmanpätkän viimeisellä rivillä.

Jos kokonaislukua ei olekaan vaikka `-` tai `+` on, eli jos syötteenä on esimerkiksi `linja-auto`, niin funktion `hae_luku` kutsu ei kasvata muuttujaa `paikka`. Se huomataan testillä `paikka == luvun_alku`. Silloin edetään yhden merkin verran muuttujassa `vr_merkit` ja ohitetaan tehtävän 103 vastaus. Miksi pitää edetä nimenomaan yksi merkki, eikä enempää tai vähempää? 113

Kokonaislukutyypin pienimmän arvon vastaluku ei tavallisesti kuulu samaan kokonaislukutyyppiin. Esimerkiksi 16-bittisten kokonaislukujen arvoalue on tyypillisesti $-32\,768, \ldots, 32\,767$. Niinpä jos kokonaisluvulla on etumerkki `-`, niin tuloksen laskeminen kuten vastauksessa 112 ja lopuksi vastaluvuksi vaihtaminen on periaatteessa epäluotettavaa, sillä jos syötteenä on $-32\,768$, niin välivaihe $32\,768$ saattaa aiheuttaa aritmeettisen ylivuodon.[5] Eräs luotettava tapa ratkaista tämä ongelma on tuottaa merkkijonosta alunperin negatiivinen luku tai nolla, ja vaihtaa se vastaluvukseen jos etumerkkiä ei ole tai se on `+`.

Kirjoita edellä olevien ajatusten mukainen `hae_luku`. Ota huomioon, että parametria `merkit` ei saa lukea sen alun eikä lopun ohi! 114

Vastauksen 105 `if`-lauseen `then`-haarassa on nyt 7 koodiriviä ja 2 kommenttiriviä, ja `else`-haarassa pelkästään 2 koodiriviä. Ohjelma toimii toki näinkin, mutta joidenkin mielestä koodista saadaan selkeämpi valitsemalla lyhyempi haara `then`-haaraksi. Se edellyttää `if`-lauseen ehdon muuntamista negaatiokseen. Se onnistuu maalaisjärjellä tai logiikasta tutulla De Morganin lailla. Koska silmukka loppuu `if`-lauseen jälkeen, `else` voidaan korvata lisäämällä `then`-haaran loppuun `continue;`, jolloin säästetään yksi sisennystaso. Kirjoita näin muunnettu silmukka `/* Etsi sanan lopunohikohta. */` ! 115

Jos osuudessa `if( paikka == luvun_alku ){...}` sijaitseva `++uc_paikka;`

si `vr_merkit` olisi globaali ja määritelty ennen kuin `hae_luku`, niin aliohjelmassa `hae_luku` sijaitseva `vr_merkit.clear();` tyhjentäisi parametrin `merkit`.

[5] Käytännössä tavallisinta on, että kun tulokseksi pitäisi tulla $32\,768$, niin tuleekin $-32\,768$, ja kun siitä otetaan vastaluku, saadaan jälleen $-32\,768$. Saadaan siis oikea lopputulos mutta virheellisen välivaiheen kautta.

muutettaisiin muotoon `if( uc_alkutavu( merkki ) ){ ++uc_paikka; }`, niin osuus `/* Jollei vuorossa ... */` voitaisiin jättää pois. On makuasia, parantaisiko vai huonontaisiko se ohjelmaa. Ohjelma ei lyhenisi kovin monta riviä, koska `if( paikka == luvun_alku ){...}` ei enää mahtuisi tyylikkäästi yhdelle rivillle. Jollei syötteestä valtaosa ole kokonaislukuja, niin ohjelma todennäköisesti hidastuisi, koska `/* Jollei ... */` lisää kokonaisluvun mahdollisesti aloittavien merkkien käsittelyyn korkeintaan kolme testiä ja käsittelee muut merkit nopeasti, mutta `hae_luku` tekee muille merkeille 7 testiä ja 4 sijoitusta tai muuttujan alustusta. (Kääntäjien ja prosessorien tekemien optimointien vuoksi tällainen laskelma ei ole tarkka, mutta voi olla suuntaa antava.) Ohjelma olisi kuitenkin hidastuneenakin hyvin nopea. Ehkä tärkein syy jättää `/* Jollei ... */` paikalleen on se, että sen ansiosta näkyy selvästi, mitä ohjelma tekee muille merkeille kuin kokonaislukuihin sisältyville merkeille.

3 Päivämäärästä viikonpäiväksi

Tässä esimerkissä edetään pitemmälle tekstialkioiden tunnistamisessa ja syötevirheiden käsittelyssä. Lisäksi muunnetaan vuosien 1901–2099 päivämääriä 1.1.1901 alkaen kuluneiden päivien määriksi ja edelleen viikonpäiviksi.

3.1 Tehtävän kuvaus

Tehtävänä on ohjelma, joka saa syötteeksi nolla tai useampia päivämääriä. Kullekin niistä se joko ilmoittaa viikonpäivän tai antaa yksityiskohtaisen virheilmoituksen miksi se ei olekaan päivämäärä. Päivämäärä on muotoa päivä.kuukausi.vuosi, missä päivä ja kuukausi koostuvat yhdestä tai kahdesta numeromerkistä ja vuosi koostuu neljästä numeromerkistä. Päivämäärät on erotettu toisistaan tyhjällä tilalla eli välilyönneillä ja/tai rivinsiirroilla. Ylimääräistä tyhjää tilaa voi olla missä tahansa paitsi päivämäärien sisällä.

Vuoden ja vuorokauden pituudet määräytyvät maapallon kierrosta auringon ja itsensä ympäri. Tähtitieteellinen vuosi ei ole tarkka monikerta vuorokausia. Jotta kalenterivuoden vuorokausien määrä olisi kokonaisluku, ja jotta kalenteri säilyisi pitkälläkin aikavälillä tahdissa vuodenaikojen kanssa, on osassa kalenterivuosia yksi vuorokausi enemmän kuin muissa. Pidennettyjä vuosia kutsutaan karkausvuosiksi.

Suomessa on 1.3.1753 alkaen noudatettu niin sanottua gregoriaanista kalenteria. Siinä karkausvuodet ovat ne neljällä jaolliset vuodet, jotka eivät ole jaollisia sadalla tai ovat jaollisia neljälläsadalla. Jos pysytellään vuosissa 1901–2099, niin riittää testata onko vuosi jaollinen neljällä. Tämä toimii, koska ainoa tälle välille osuva sadalla jaollinen vuosi eli 2000 on jaollinen neljälläsadalla.

Syötteellä (syötteessä ei ole rivin alussa välilyöntejä, vaikka rivi onkin esteettisyyssyistä sisennetty tässä tekstissä)

```
31.5.2023   1.1.1863 14.4.2044  29.2.2038   Hups! 15.5.
```

ohjelman tulostus näyttää tältä:

```
31.5.2023: keskiviikko
Virhe rivillä 1: vuosi ei saa olla alle 1901
31.5.2023   1.1.1863 14.4.2044  29.2.2038   Hups! 15.5.
              ?
14.4.2044: torstai
Virhe rivillä 1: tässä kuussa tänä vuonna ei ole näin montaa päivää
31.5.2023   1.1.1863 14.4.2044  29.2.2038   Hups! 15.5.
                               ?
Virhe rivillä 1: päivä puuttuu
31.5.2023   1.1.1863 14.4.2044  29.2.2038   Hups! 15.5.
                                           ?
Virhe rivillä 1: vuosi puuttuu
31.5.2023   1.1.1863 14.4.2044  29.2.2038   Hups! 15.5.
                                                     ?
```

3.2 Muu kuin syöteluokka

Seuraavassa alaluvussa suunnitellaan luokka, jonka avulla syötteen lukeminen ja virheilmoitusten antaminen saadaan tämän näköiseksi:

```
unsigned alku = syote.paikka(), pv = syote.hae_etumerkiton();
syote.tarkasta( syote.edettiin() > 0, "päivä puuttuu" );
syote.tarkasta( syote.edettiin() <= 2, "liikaa numeroita päivässä" );
syote.tarkasta( 1 <= pv, "päivä ei saa olla 0" );
syote.tarkasta( pv <= 31, "päivä ei saa olla yli 31" );
syote.ohita_merkki( '.', "päivän jälkeen puuttuu piste" );

unsigned kk = syote.hae_etumerkiton();
...

unsigned vv = syote.hae_etumerkiton();
syote.tarkasta( syote.edettiin() > 0, "vuosi puuttuu" );
syote.tarkasta( syote.edettiin() == 4, "vuodessa pitää olla 4 numeroa" );
syote.tarkasta( 1901 <= vv, "vuosi ei saa olla alle 1901" );
syote.tarkasta( vv <= 2099, "vuosi ei saa olla yli 2099" );

const unsigned *kk_pit = vv % 4 > 0 ? kk_pituus1 : kk_pituus2;
syote.aseta_alkupaikka( alku );
syote.tarkasta(
  kk-1 < 12 && pv <= kk_pit[ kk-1 ],
  "tässä kuussa tänä vuonna ei ole näin montaa päivää"
);
```

```
if( syote.virhetilassa() ){
  syote.pois_virhetilasta(); syote.etsi_tyhja(); return;
}
```

Esimerkin alussa otetaan käyttöön muuttuja `alku` tallettamaan päivämäärän alkukohta sekä `pv` tallettamaan syötteessä saatu päivä. Päivästä tarkastetaan, että se on annettu yhdellä tai kahdella numerolla, ja että se on välillä 1, ..., 31. Sitten tarkastetaan, että syötteessä on seuraavaksi päivän ja kuukauden väliin kuuluva piste. Sen jälkeen haetaan ja tarkastetaan kuukausi, sen ja vuoden väliin kuuluva piste sekä vuosi.

Kun vuosi on saatu, valitaan jompikumpi kahdesta taulukosta, joihin on talletettu kuukausien pituudet tavallisena vuonna ja karkausvuonna. Nämä taulukot on luotu aikaisemmassa kohdassa ohjelmaa seuraavasti.

```
/* Kuukausien pituudet normaalivuonna ja karkausvuonna */
const unsigned
  kk_pituus1[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
  kk_pituus2[] = { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Muuttujat `kk_pituus1` ja `kk_pituus2` ovat C-tyylisiä taulukoita ja `kk_pit` on osoitin. Hieman vaivalloisempaa mutta ehkä myös tyylikkäämpää olisi ollut käyttää C++:n `vector`-säiliöitä ja viitetyyppiä sekä versiosta 2011 alkaen C++:ssa ollutta alustusmekanismia.

Määreellä `const` estetään sijoittaminen muuttujiin `kk_pituus1` ja `kk_pituus2`. Sillä saadaan suojaa ohjelmointivirheiltä, joissa kuukausien pituuksia yritetään vahingossa muuttaa. Myös muuttujan `kk_pit` täytyy olla `const`-tyyppiä, jotta siihen voitaisiin sijoittaa osoitin `const`-tyyppiseen tietoon.

Seuraavaksi syöteluokka komennetaan käyttämään mahdollisen virheilmoituksen alkukohtana aiemmin talletettua koko päivämäärän alkukohtaa. Sitten tarkastetaan, että päivä ei ylitä kuukauden pituutta, eli että päivämäärä ei ole esimerkiksi 31.6.2000. Jos päivä on liian suuri, niin alkukohdan asetuksen ansiosta virheen merkki tulee päivän ensimmäisen numeron kohdalle, kuten edellä olleessa esimerkissä `29.2.2038`. Samasta syystä ohjelmassa myöhemmin oleva komento `std::cout << syote.merkit()` tulostaa päivämäärän kokonaan ja vain sen. Ilman alkukohdan asetusta alkukohtana olisi vuoden alku, joten virhemerkki tulisi vuoden ensimmäisen numeron kohdalle ja `std::cout << syote.merkit()` tulostaisi pelkän vuoden.

Jos tarkastettava asia ei päde, syöteluokka antaa siihen liittyvän virheilmoituksen ja siirtyy virhetilaan. Virhetilassa ollessaan syöteluokka ei anna uusia virheilmoituksia eikä etene syötteessä. Sen ansiosta ohjelmassa ei tarvitse vähän väliä testata ollaanko virhetilassa ja ohjata suoritusreittiä testien tulosten mukaan. Vasta kun kaikki tarkastukset on tehty, niin jos syöteluokka on virhetilassa, se komennetaan pois virhetilasta ja etenemään syötteessä kunnes kohdataan välilyönti

tai rivinsiirto, tai syöte loppuu. Tämä selkeyttää ohjelmaa huomattavasti. Niissä ohjelmointikielissä, joissa on poikkeusmekanismi, saman hyödyn saa sitä käyttämällä.

Kun tarkastetaan että päivä ei ylitä kuukauden pituutta, lasketaan `kk_pit[ kk-1 ]` . Koska aiemmat tarkastukset eivät välttämättä menneet läpi, saattaa olla, että `kk` ei sisällä kelvollista kuukauden numeroa. Siksi tarkastuksessa on osuus `kk-1 < 12 &&` . Se varmistaa, että `kk_pit[ kk-1 ]` ei indeksoi ohi taulukon `kk_pit` reunojen. Se toimii myös jos `kk` on nolla, koska silloin `kk-1` on suurin `unsigned` -tyyppinen arvo, ja se on paljon suurempi kuin 12. Osuutta `kk-1 < 12 &&` ei tarvita poikkeusmekanismia käytettäessä, koska silloin suoritus ei tule tähän kohtaan ollenkaan, jos `kk` sisältää kelvottoman kuukauden numeron.

Päivämäärien käsittelyä helpottaa, jos ne muutetaan jostakin vertailupäivästä alkaen kuluneiden päivien määriksi. Tässä tapauksessa eräs hyvä vertailupäivä on 1.1.1901, koska se on pienin syötteessä sallittu päivämäärä. Koska karkauspäiviin voidaan soveltaa yksinkertaistettua sääntöä, jossa ei oteta huomioon muuta kuin että onko vuosi jaollinen neljällä, 1.1.1901 alkaen kuluneiden päivien määrä voidaan laskea seuraavasti.

```
/* Muunna 1.1.1901 alkaen kuluneiden päivien määräksi. */
vv -= 1901; --pv; --kk; pv += 365*vv + vv/4;
for( unsigned ii = 0; ii < kk; ++ii ){ pv += kk_pit[ ii ]; }
```

Lauseke `365*vv + vv/4` tuottaa 1.1.1901 alkaen kuluneiden kokonaisten vuosien yhteensä sisältämien päivien määrän. Osuus `+ vv/4` lisää karkauspäivät. Se ei lisää mitään jos `vv` on 0, 1, 2 tai 3, eli vuosina 1901, 1902, 1903 ja 1904. Vuodesta 1901 alkaen ensimmäinen karkausvuosi oli 1904. Ensimmäinen päivä, johon mennessä vuodesta 1901 alkaen oli ehtinyt kulua kokonainen karkausvuosi, oli 1.1.1905. Siksi vielä vuoden 1904 kohdalla ei pidä tehdä karkausvuosilisäystä, mutta vuoden 1905 kohdalla pitää. Seuraava karkausvuosilisäys pitää tehdä vuoden 1909 kohdalla, sitten vuoden 1913 kohdalla ja niin edelleen. Osuus `+ vv/4` tekee juuri niin.

Komennon `--pv;` jälkeen `pv` sisältää kuluvan kuukauden aikana jo kuluneiden päivien määrän. Esimerkiksi 3.4.2029 tarkoittaa, että on vuoden 2029 huhtikuun kolmas päivä, joten siinä kuussa on ehtinyt kulua kaksi päivää, nimittäin 1.4.2029 ja 2.4.2029. Seuraava rivi lisää muuttujaan `pv` kokonaan kuluneiden kuukausien pituudet. Karkauspäivät tulee otettua huomioon sen kautta, että aiemmin `kk_pit` valittiin kahdesta vaihtoehdosta sen mukaan, onko karkausvuosi.

Vaikka muunnos on ohjelmanpätkänä lyhyt ja sen selitys saattaa vaikuttaa yksinkertaiselta, on näissä asioissa helppo jättää vahingossa huomaamatta jotain olennaista. Siksi luvussa 3.4 kerrotaan, miten päivämäärän hakeminen syötteestä ja tämä muunnos voidaan yhdessä testata perusteellisesti.

Päivän juoksevasta numerosta saadaan viikonpäivä hyödyntämällä sitä, että sama viikonpäivä toistuu aina seitsemän päivän välein. Jos 1.1.1901 olisi ollut maanantai, niin viikonpäivä saataisiin laskemalla päivän juokseva numero modulo 7 ja indeksoimalla sillä taulukkoa, jossa on viikonpäivien nimet maanantaista alkaen.

```
/* Viikonpäivien nimet */
const std::string viikonpaiva[] = {
  "maanantai", "tiistai", "keskiviikko", "torstai", "perjantai",
  "lauantai", "sunnuntai"
};
```

Niin toteutettu ohjelma tuottaa kuitenkin vääriä tuloksia. Sen mukaan esimerkiksi 1.5.2023 olisi ollut sunnuntai, kun se todellisuudessa oli maanantai. Tämän voi korjata monella eri tavalla. Ehkä tyylikkäin tapa on lisätä päivän juoksevaan numeroon 1 ennen kuin lasketaan modulo 7.

```
/* Tulosta viikonpäivä. */
std::cout << syote.merkit() << ": " << viikonpaiva[ (pv+1) % 7 ] << '\n';
```

Tarvitaan nimi aliohjelmalle, jossa suurin osa tähän asti kerrotusta ohjelmakoodista sijaitsee, sekä pääohjelma, joka kutsuu sitä.

```
void kasittele_paivamaara(){
  ...
}

/* Pääohjelma */
int main(){
  syote.ohita_tyhja();
  while( !syote.lopussa() ){
    kasittele_paivamaara();
    syote.ohita_tyhja();
  }
}
```

3.3 Syöteluokka

Tässä alaluvussa kerrotaan, miten voidaan toteuttaa syöteluokka, jota voidaan käyttää edellisessä alaluvussa kuvatulla tavalla. Syöteluokkaa suunniteltaessa otettiin huomioon yleisempiäkin kuin vain tämän ohjelman tarpeita.

Alaluvun 3.1 lopussa näytettiin esimerkki ohjelman tulostuksesta. Siitä näkyy, että virheilmoitukseen tulostetaan virheellinen syöterivi ja sen numero. Kuten jokainen ison ohjelman käännösvirheitä korjannut tietää, rivinumero auttaa virhepaikan löytämisessä. Voidakseen tulostaa rivinumeron ja rivin sisällön, on syöteluokan pidettävä ne tallessa jossain muodossa. Helpointa on, että rivinumero on kokonaislukumuuttujassa ja viimeksi luettu syöterivi merkkijonomuuttujassa.

Hyvien nimien keksiminen muuttujille on toisinaan vaikeaa. Nimien olisi hyvä olla kuvaavia. Toisaalta jos nimet ovat pitkiä, niin riveistä tulee herkästi ylipitkiä tai esimerkiksi **if**-lauseiden ehtoja joudutaan jakamaan usealle riville. Opetuskäyttöön tarkoitetuissa esimerkeissä rivien suuri pituus ja määrä on usein haitallisempaa kuin nimien kryptisyys, joten tässä luvussa suositaan lyhyehköjä nimiä. Isojen ohjelmistojen laatimisessa saattaa olla parempi valita helppotajuiset nimet, vaikka ne olisivat pitkät.

Nimien valintaa saattaa vaikeuttaa sekin, että sama nimi olisi luonteva sekä luokan sisäiselle muuttujalle että funktiolle joka palauttaa sen arvon. Syöteluokassa on muutama sellainen tapaus. Esimerkiksi edellä näimme, että syöteluokassa on funktio nimeltä `lopussa`, joka palauttaa `true` jos ja vain jos syöte on loppunut. Toisaalta syöteluokassa on myös muuttuja nimeltä `loppu`, jonka arvo on `true` jos ja vain jos syöte on loppunut. Itse asiassa `lopussa` ei tee muuta kuin palauttaa tämän muuttujan arvon:

```
bool lopussa() const { return loppu; }
```

Tässä `const` dokumentoi, että aliohjelma ei muuta syöteluokan sisältöä. Koska `lopussa` ja `loppu` tuottavat täsmälleen saman tiedon, olisi luontevaa käyttää niille samaa nimeä. Mutta C++-kääntäjä ei salli sitä, joten niille on keksitty eri nimet.

Nimien valinnassa ajateltiin, että on tärkeämpää, että luokasta ulos näkyvät nimet ovat kuvaavia kuin että luokan sisäiset nimet ovat kuvaavia. Luokan käyttäjät tarvitsevat vain ulos näkyviä nimiä. Erityisen hankala pari olivat funktio ja muuttuja, jotka kertovat, missä kohdassa syöteriviä syöterivin tulkinta on. Funktiolle annettiin nimeksi `paikka`, koska se on melko selkeä. Muuttujalle annettiin nimeksi `pkka`, koska muuttuja esiintyy tiuhaan, joten sen nimen on hyvä olla lyhyt, ettei ohjelman riveistä tulisi pitkiä.

Kaiken kaikkiaan syöteluokassa on kuusi muuttujaa:

```
/* Syötteen tulkinta tekstialkioiksi virheilmoituksineen */
class syote_tulkki{

  std::string rivi;            // viimeisin syöterivi jatkettuna '\n'
  unsigned r_nro, pkka, alku;  // sijainti ja aikaisempi paikka syötteessä
  bool virhe, loppu;

public:

  ...

};
```

Viimeksi luetun rivin sisällön tallettavan muuttujan nimeksi valittiin `rivi`, koska tarvittiin lyhyt nimi ja `merkit` sai tärkeämmän tehtävän ulos näkyvän funktion

nimenä. Muuttuja `r_nro` sisältää viimeksi luetun rivin numeron. Syötteen tulkinta on rivin kohdassa `pkka`, ja tulkittava osakokonaisuus alkoi saman rivin kohdassa `alku`. Bitit `virhe` ja `loppu` kertovat ollaanko virhetilassa ja onko syöte loppunut.

Syöteluokan toteutus noudattaa seuraavia periaatteita.

- Muuttujan `rivi` lopussa on aina rivinsiirtomerkki (paitsi sen lyhyen hetken, jona siihen on sijoitettu uusi arvo mutta sen loppuun ei ole vielä ehditty lisätä rivinsiirtomerkkiä). Kuten jatkossa nähdään, tämän ansiosta voidaan monia toimintoja yksinkertaistaa.
- Jos syöteluokka on virhetilassa, niin mikään toiminto ei etene syötteessä eikä muutenkaan muuta syöteluokan tilaa, paitsi ne, jotka asettavat syöteluokan pois virhetilasta. Syy tälle kerrottiin sivulla 28.
- Muuttujat `alku` ja `paikka` viittaavat aina viimeksi luettuun syöteriviin. Jos viimeksi luettua syöteriviä ei ole, niin niissä on 0.

Kun syötetulkki luodaan, `rivi` saa sisällökseen `"\n"`, `unsigned`-muuttujat saavat arvon `0`, ja `bool`-muuttujat arvon `false`.

```
/* Rakentaja */
syote_tulkki():
  rivi( "\n" ), r_nro( 0 ), pkka ( 0 ), alku( 0 ),
  virhe( false ), loppu( false )
{}
```

Seuraava aliohjelma hakee uuden syöterivin. Ohjelmamme käyttää sitä vain syöteluokan muiden toimintojen toteutukseen, mutta se jätettiin silti ulos näkyväksi, koska se saattaa olla hyödyllinen jossain toisessa ohjelmassa.

```
/* Jollei olla virhetilassa eikä lopussa, niin hae uusi rivi. */
void uusi_rivi(){
  if( virhe || loppu ){ return; }
  std::getline( std::cin, rivi );
  if( !std::cin ){ rivi.clear(); loppu = true; }
  rivi.push_back( '\n' ); ++r_nro; alku = pkka = 0;
}
```

Jos ollaan virhetilassa, niin syötteessä ei saa edetä eikä syöteluokan tilaa saa muttaa, joten uutta riviä ei pidä hakea. Niinpä suoritus ohjataan pois aliohjelmasta. Jos syöte on loppu, niin ei ole olemassa uutta riviä jonka voisi hakea, joten suoritus kannattaa ohjata pois aliohjelmasta. Se on helpompaa kuin miettiä, mitä seuraavat rivit tekisivät jos ne suoritettaisiin kun syöte on loppu. Osuus `|| loppu` saattaa lisätä suoritusaikaa kun syöteluokka ei ole virhetilassa eikä syöte ole loppu, mutta vain mitättömän vähän.

Kun `rivi` on luettu tai syötteen loputtua korvattu tyhjällä merkkijonolla, sen perään lisätään rivinsiirtomerkki `'\n'`. Se on tarpeen, koska C++:n `getline` jättää syötteessä olleet rivinsiirtomerkit pois. Rivinumeroa kasvatetaan myös syötteen loppuessa, jotta mahdollisesti silloin tuleva virheilmoitus ei johtaisi harhaan. Ilman rivinumeron kasvatusta virheilmoitus näyttäisi tyhjän rivin mutta viimeisen, mahdollisesti epätyhjän rivin numeron (tai numeron 0, jos syöte on tyhjä).

Monissa ohjelmissa seuraavan tekstialkion alkukohta etsitään ohittamalla välilyöntejä ja hakemalla uusia rivejä kunnes löytyy jotain epätyhjää tai syöte loppuu. Ohjelmassamme sitä tarvitaan seuraavan päivämäärän alkukohdan löytämiseksi.

```
/* Jollei olla virhetilassa, niin ohita tyhjä tila. */
void ohita_tyhja(){
  if( virhe ){ return; }
  while( rivi[ pkka ] == ' ' ){ ++pkka; }
  while( rivi[ pkka ] == '\n' ){
    uusi_rivi();
    if( virhe || loppu ){ return; }
    while( rivi[ pkka ] == ' ' ){ ++pkka; }
  }
  alku = pkka;
}
```

Silmukat `while( rivi[ pkka ] == ' ' ){ ++pkka; }` pysähtyvät viimeistään muuttujan `rivi` lopussa olevaan rivinsiirtomerkkiin. Siksi ennen indeksointeja `rivi[ pkka ]` ei tarvitse testata, onko `pkka < rivi.size()`.

On helppo tarkastaa, että `uusi_rivi` ei voi asettaa syöteluokkaa virhetilaan. Siksi ei olisi tarpeen testata sen kutsumisen jälkeen, ollaanko virhetilassa. Testi kuitenkin helpottaa ohjelmaa katselmoitaessa sen tarkastamista, että `ohita_tyhja` ei koskaan etene virhetilassa, tekemällä tarpeettomaksi tarkastaa, voiko `uusi_rivi` asettaa syöteluokan virhetilaan. Se myös varmistaa, että `ohita_tyhja` ei muutu virheelliseksi, jos `uusi_rivi` tulevaisuudessa muutetaan jossain tilanteessa asettamaan syöteluokka virhetilaan. Testin hinta on mitätön.

Joissakin ohjelmissa tarvitaan toimintoa, joka ohittaa välilyönnit mutta ei jatka seuraavalle riville. Kirjoita aliohjelma `ohita_valilyonnit`, joka tekee niin, ja muuta `ohita_tyhja` käyttämään sitä! 116

Luvussa 3.2 käytettiin komentoa `syote.edettiin()` kertomaan, kuinka monta merkkiä edettiin syötteessä kun luettiin päivän, kuukauden tai vuoden numero. Sen avulla testattiin, että päivä ja kuukausi on annettu yhdellä tai kahdella numerolla, ja vuosi on annettu neljällä numerolla. Olettaen, että muut toiminnot huolehtivat muuttujan `alku` päivittämisestä, `edettiin()` voidaan toteuttaa hyvin yksinkertaisesti.

```
unsigned edettiin() const { return pkka - alku; }
```

Etumerkittömän kokonaisluvun hakeminen on yksinkertaista, koska ei tarvitse käsitellä etumerkkejä ja koska testi, saatiinko lukua lainkaan, voidaan jättää kutsujalle toiminnon `edettiin` avulla toteutettavaksi. Jotta `edettiin` toimisi oikein, `alku` täytyy päivittää.

```
/* Jollei olla virhetilassa, niin tulkitse seuraavat numeromerkit
  etumerkittömäksi kokonaisluvuksi. */
unsigned hae_etumerkiton(){
  if( virhe ){ return 0; }
  alku = pkka;
  unsigned tulos = 0;
  while( on_numero( rivi[ pkka ] ) ){
    tulos *= 10; tulos += rivi[ pkka ] - '0'; ++pkka;
  }
  return tulos;
}
```

Myöskään `tarkasta` ei sisällä mitään erikoista.

```
/* Jos ehto ei päde eikä olla virhetilassa, niin aseta virhetilaan ja
  tulosta virheilmoitus. */
void tarkasta( bool ehto, const std::string & viesti ){
  if( virhe || ehto ){ return; }
  virhe = true;
  std::cout << "Virhe rivillä " << r_nro << ": " << viesti << '\n' << rivi;
  std::cout.width( alku+1 ); std::cout << '?' << '\n';
}
```

Luvussa 3.2 käytettiin päivämäärien tulostamisessa aliohjelmia `aseta_alkupaikka` ja `merkit`.

```
void aseta_alkupaikka( unsigned ap ){
  if( !virhe ){ alku = ap; }
}

std::string merkit() const { return rivi.substr( alku, pkka - alku ); }
```

Syötteestä voidaan poimia merkkijono seuraavasti:

```
syote.ohita_tyhja(); syote.etsi_tyhja(); mj = syote.merkit();
```

Merkin ohittamisessa täytyy ottaa huomioon, että rivinsiirto ohitetaan eri tavalla kuin muut merkit.

```
/* Jollei olla virhetilassa, niin tarkasta, onko merkki seuraavana. Jos
  on, niin ohita se. Jollei, niin anna virheilmoitus ja aseta virhetila. */
void ohita_merkki( char merkki, const std::string & viesti ){
  if( virhe ){ return; }
  alku = pkka; tarkasta( !loppu && rivi[ pkka ] == merkki, viesti );
  if( virhe ){ return; }
  if( merkki == '\n' ){ uusi_rivi(); }else{ ++pkka; }
}
```

Aliohjelmien `virhetilassa`, `pois_virhetilasta` ja `etsi_tyhja` toteutus noudattaa samoja linjoja. Tarvittaessa voidaan lisätä aliohjelmia, kuten sellainen, joka hakee etumerkillisen kokonaisluvun.

Syöteluokasta tarvitaan vain yksi esiintymä. Sen on luontevaa olla globaali muuttuja, koska ohjelmalla on vain yksi syötekanava.

```
syote_tulkki syote;
```

3.4 Testaus

Tehtävien vastauksia

1. Hienoa, löysit tänne! Tuossa ↑ on linkki takaisin tehtävän 1 kysymyksen kohdalle.

2.
```
#include <iostream>

const int alin = 940, ylin = 1066;
bool esiintyy[ ylin - alin + 1 ] = {};

const char *ei_kylla[] = { "ei", "kyllä" };

int main(){
  int paine = 0;
  while( true ){
    std::cin >> paine;
    if( paine == 0 ){ break; }
    esiintyy[ paine - alin ] = true;
  }
  while( true ){
    std::cin >> paine;
    if( paine == 0 ){ break; }
    std::cout << paine << ": " <<
      ei_kylla[ esiintyy[ paine - alin ] ] << "\n";
  }
}
```

   Tehtävässä luvattiin `alin` $\leq$ `paine` $\leq$ `ylin`. Siksi $0 \leq$ `paine` $-$ `alin` $\leq$ `ylin` $-$ `alin`. Niinpä `paine` $-$ `alin` sopii indeksoimaan taulukkoa, ja taulukon kooksi tarvitaan ja riittää `ylin` $-$ `alin` $+ 1$. Komento `= {}` alustaa taulukon jokaisen alkion `false` :ksi.

3. Otetaan käyttöön venyvä taulukko. C++:ssa se on `std::vector`. Ohjelman alkuun lisätään `#include <vector>`. Rivin `bool esiintyy[ ylin - alin + 1 ] = {};` tilalle kirjoitetaan `std::vector< bool > esiintyy;`, joka luo tyhjän taulukon. Osuus `, ylin = 1066` poistetaan. Pääohjelman silmukat muutetaan muotoon

```
while( true ){
  std::cin >> paine;
  if( paine == 0 ){ break; }
  unsigned paikka = paine - alin;
  if( paikka >= esiintyy.size() ){ esiintyy.resize( paikka+1, false ); }
  esiintyy[ paikka ] = true;
}
while( true ){
  std::cin >> paine;
  if( paine == 0 ){ break; }
```

```
  unsigned paikka = paine - alin;
  std::cout << paine << ": " <<
    ei_kylla[ paikka < esiintyy.size() && esiintyy[ paikka ] ] << "\n";
}
```

Kun mitattu paine on saatu, ohjelma tarkastaa, onko taulukossa jo paikka sille. Tarvittaessa ohjelma kasvattaa taulukkoa. Taulukkoon lisätyt alkiot alustetaan arvoon `false`. Koska C++:ssa `size()` on `unsigned`-tyyppinen, on myös muuttujan `paikka` tyypiksi valittu `unsigned`. Jossakin muussa ohjelmointikielessä `int` saattaa olla parempi. Kun testataan onko saatu paine taulukossa, taulukon oikean reunan ohi menevät paineet ohjataan tuottamaan `ei`.

4. Rivit

   ```
   const int alin = 940;
   std::vector< bool > esiintyy;
   ```

   korvataan riveillä

   ```
   const int raja = 1000;
   std::vector< bool > pienet, suuret;
   ```

   Rivin

   ```
   unsigned paikka = paine - alin;
   ```

   molemmat esiintymät korvataan riveillä

   ```
   std::vector< bool > & esiintyy = paine < raja ? pienet : suuret;
   unsigned paikka = paine < raja ? raja - 1 - paine : paine - raja;
   ```

   Ylempi viimeksi mainituista riveistä asettaa muuttujan `esiintyy` tarkoittamaan `pienet` tai `suuret` sen mukaan, onko `paine` < 1000. Alempi asettaa muuttujan `paikka` arvon vastaavasti. Olennaista on käyttää toimintoa, joka ei kopioi koko taulukkoa vaan ainoastaan osoittimen tai viitteen siihen. Siksi edellä käytettiin C++:n viitetyyppiä, jonka tunnusmerkki on `&`. Javassa ja C#:ssa tätä ei tarvitse ottaa erikseen huomioon, koska niiden olioiden tapauksessa `=` kopioi viitteen eikä sisältöä.

5. ```
   class pistetyyppi{
     int x, y;
   public:
     pistetyyppi( int x, int y ): x(x), y(y) {}
     bool operator==( pistetyyppi toinen ) const {
       return x == toinen.x && y == toinen.y;
     }
     bool operator!=( pistetyyppi toinen ) const { return !( *this == toinen ); }
   };
   ```
   ```

6. ```
   /* Jos monikulmiot ovat erikokoiset, niin palauta false. */
   unsigned nn = mk1.size();
   if( mk2.size() != nn ){ return false; }

   /* Jos monikulmiot ovat tyhjät, niin palauta true. */
   if( nn == 0 ){ return true; }
   ```

7. Etsitään `mk2` :sta se kärki, joka on `mk1` :ssä ensimmäisenä. Jos sellaista ei ole, niin palautetaan `false` . Muutoin kokeillaan, ovatko `mk2` :n kärjet äsken löydetyn kärjen jälkeen samat kuin `mk1` :n kärjet toisesta alkaen. Jos kyllä, niin palautetaan `true` . Jollei, niin kokeillaan ovatko `mk2` :n kärjet takaperin äsken löydetyn kärjen edeltäjästä alkaen samat kuin `mk1` :n kärjet etuperin toisesta alkaen, ja palautetaan sen mukaan `true` tai `false` .

8. ```
   /* Etsi mk1:n ensimmäinen kärki mk2:sta. Jollei löydy, palauta false. */
   unsigned ii = 0;
   for( ; ii < nn; ++ii ){
     if( mk1[ 0 ] == mk2[ ii ] ){ break; }
   }
   if( ii >= nn ){ return false; }
   ```

9. ```
   /* Jos monikulmiot täsmäävät samaan suuntaan, niin palauta true. */
   unsigned jj = 1, kk = ii + 1;
   for( ; jj < nn; ++jj, ++kk ){
     if( mk1[ jj ] != mk2[ kk % nn ] ){ break; }
   }
   if( jj >= nn ){ return true; }
   ```

10. ```
    /* Jos täsmäävät eri suuntaan, niin palauta true, muutoin false. */
    jj = 1; kk = nn + ii - 1;
    for( ; jj < nn; ++jj, --kk ){
      if( mk1[ jj ] != mk2[ kk % nn ] ){ return false; }
    }
    return true;
    ```

    Koska `%` saattaa toimia negatiivisilla luvuilla odottamattomasti[6] ja koska `kk` on `unsigned` -tyyppiä, ei saa käydä niin, että `kk` ohittaa nollan. Osuus `nn +` estää sen, eikä vääristä lukua `kk % nn` .

11. Muutoin `mk1` :n ensimmäinen kärki ei ole olemassa ja vastauksen 8 ohjelmanpätkä palauttaa virheellisesti `false` .

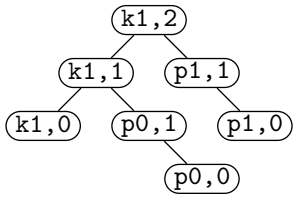
12. 838 169 940

13. `p2,3 p2,2 p2,1 p2,0`

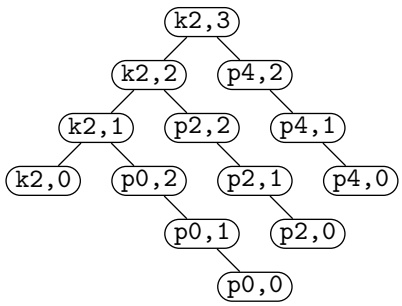
14. `k1,2 k1,1 k1,0 p0,1 p0,0 p1,1 p1,0`

[6] `https://en.wikipedia.org/wiki/Modulo`

15.



16. `k2,3 k2,2 k2,1 k2,0 p0,2 p0,1 p0,0 p2,2 p2,1 p2,0 p4,2 p4,1 p4,0`



17. $m+1$

18. $n+1$

19. $(n+1)m$. Kun $n = 12345$ ja $m = 67890$, kaava tuottaa 838 169 940. Se on sama kuin vastaus 12.

20. Käytetään rengaslistaa. Listan viimeisen alkion `seur` osoittaa listan ensimmäiseen alkioon. Linkki koko listaan osoittaa viimeiseen alkioon.

```
alkio *yhdista( alkio *eka, alkio *toka ){
  alkio *apu = eka->seur; eka->seur = toka->seur; toka->seur = apu;
  return toka;
}
```

21.
```
void tulosta( alkio *a1 ){
  if( a1 ){
    alkio *a2 = a1->seur;
    std::cout << a2->nimi;
    while( a2 != a1 ){ a2 = a2->seur; std::cout << ' ' << a2->nimi; }
  }
  std::cout << '\n';
}
```

22.
```
void tarkasta_sulkeet( const std::string & mj ){
  pino< char > P;
  for( unsigned ii = 0; ii < mj.size(); ++ii ){
    if( mj[ ii ] == '(' ){ P.laita( ')' ); }
    else if( mj[ ii ] == '[' ){ P.laita( ']' ); }
```

```
    else if( mj[ ii ] == '{' ){ P.laita( '}' ); }
    else if( mj[ ii ] == ')' || mj[ ii ] == ']' || mj[ ii ] == '}' ){
      if( P.on_tyhja() ){
        std::cout << mj[ ii ] << " vailla paria\n"; return;
      }else{
        char mrk = P.ota();
        if( mrk != mj[ ii ] ){ std::cout << mrk << " puuttuu\n"; return; }
      }
    }
  }
  if( !P.on_tyhja() ){ std::cout << P.ota() << " puuttuu\n"; return; }
}
```

23. Sitä mukaa kuin alkioita tulee, ne laitetaan ensimmäiseen pinoon. Kun alkio pitää ottaa jonosta, niin jos toinen pino on epätyhjä, niin otetaan sen ylin. Jos toinen pino onkin tyhjä, niin ensimmäisen pinon alkiot siirretään siihen ja sitten otetaan sen ylin. Siirto tapahtuu ottamalla ensimmäisestä pinosta ylin alkio ja laittamalla se toiseen pinoon, ja toistamalla tätä kunnes ensimmäinen pino on tyhjä. Jono on tyhjä jos ja vain jos sen molemmat pinot ovat tyhjiä. Tyhjästä jonosta tai pinosta ottaminen ei ole määritelty. Alla oleva toteutus toimii tyhjästä jonosta otettaessa samoin kuin pino.

```
template< typename Alkiotyyppi > class jono{
  class pino< Alkiotyyppi > P1, P2;
public:
  bool on_tyhja() const { return P1.on_tyhja() && P2.on_tyhja(); }
  void laita( Alkiotyyppi aa ){ P1.laita( aa ); }
  Alkiotyyppi ota(){
    if( P2.on_tyhja() ){
      while( !P1.on_tyhja() ){ P2.laita( P1.ota() ); }
    }
    return P2.ota();
  }
};
```

24. **for** $i := 2$ **to** n **do**
    if $K[i] > K[i \text{ div } 2]$ **then return** false
    return true

25. Solmun i vanhemman numero on $(i-1)$ div 2. Sitä voi testata sillä, että kun $i = 1$ tai $i = 2$ sen pitää olla 0, ja kun $i = 3$ tai $i = 4$ sen pitää olla 1. Niinhän se onkin. Aina kun i kasvaa kahdella sen pitää kasvaa yhdellä, ja niinhän se tekeekin. Solmun i lasten numerot ovat $2i+1$ ja $2i+2$. Kun $i = 0$ niiden pitää tuottaa 1 ja 2, ja niinhän ne tekevät. Kun $i = 1$ niiden pitää tuottaa 3 ja 4, ja niinhän ne tekevät. Solmun $i+1$ lapsien pitää olla heti solmun i lapsien perässä, ja niinhän ne ovatkin, koska $2(i+1)+1 = 2i+3 = (2i+2)+1$.

26. Ensimmäinen käsiteltävä kohta on 4. Siinä on D. Sillä on yksi lapsi, nimittäin kohta 8. Siinä on H. Koska $\mathsf{H} > \mathsf{D}$, nousee H kohtaan 4. Koska kohdalla 8 ei ole lapsia, aukon valutus pysähtyy ja D sijoitetaan kohtaan 8. Taulukko on nyt $[\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{H},\mathsf{E},\mathsf{F},\mathsf{G},\mathsf{D}]$. Seuraavaksi käsitellään kohta 3. Siinä on C. Sen lapset ovat kohdissa 6 ja 7, eli F ja G. Niistä G on suurempi ja se on myös suurempi kuin C, joten se nousee kohtaan 3. Kohdalla 7 ei ole lapsia, joten C sijoitetaan sinne. Taulukko on nyt $[\mathsf{A},\mathsf{B},\mathsf{G},\mathsf{H},\mathsf{E},\mathsf{F},\mathsf{C},\mathsf{D}]$. Kohdan 2 käsittelemisen jälkeen taulukko on $[\mathsf{A},\mathsf{H},\mathsf{G},\mathsf{D},\mathsf{E},\mathsf{F},\mathsf{C},\mathsf{B}]$ ja lopuksi se on $[\mathsf{H},\mathsf{E},\mathsf{G},\mathsf{D},\mathsf{A},\mathsf{F},\mathsf{C},\mathsf{B}]$.

27. Jos kaikki alkiot ovat yhtäsuuria, ei keon muokkaamisessa koskaan tarvitse siirtää aukkoa alas- eikä ylöspäin, jolloin heapsort on $\Theta(n)$. Vastauksen 68 vuoksi se ei voi olla nopeampi kuin $\Theta(n)$.

28. Lisätään k ensimmäistä alkiota kekoon, jossa on pienin ylinnä. Jokaiselle muista alkioista x, jos se on suurempi kuin keon ylin, niin otetaan keon ylin pois ja lisätään x kekoon. Tulostetaan keossa olevat alkiot.

29. $v + o + 1$

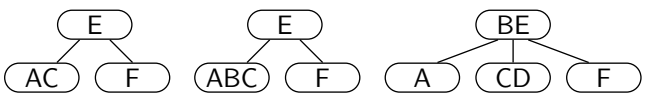
30. *laske-solmut*(p)
if $p = \bot$ **then return** 0
else return $1 +$ *laske-solmut*$(p{\uparrow}v) +$ *laske-solmut*$(p{\uparrow}o)$

31. $1 + \max\{v, o\}$

32. *etsi_mones*(p,i)
$m := 0$
if $p{\uparrow}v \neq \bot$ **then** $m := p{\uparrow}v{\uparrow}n$
if $i < m$ **return** *etsi_mones*$(p{\uparrow}v, i)$
if $i = m$ **return** p
if $p{\uparrow}o \neq \bot$ **then return** *etsi_mones*$(p{\uparrow}o, i - m - 1)$
return $\bot$

33. *monesko*(p)
$m := 0$
if $p{\uparrow}v \neq \bot$ **then** $m := p{\uparrow}v{\uparrow}n$
while $p{\uparrow}y \neq \bot$ **do**
    $q := p$; $p := p{\uparrow}y$
    if $q = p{\uparrow}o$ **then**
        $m := m + 1$
        if $p{\uparrow}v \neq \bot$ **then** $m := m + p{\uparrow}v{\uparrow}n$
return m

34. E CE CEF



35. Se on solmun ensimmäinen avain, joka on suurempi kuin a. Jos solmun jokainen avain on enintään a, niin sitä ei ole.

    $i := 1$
    while $i \leq p\uparrow n$ && $p\uparrow k[i] \leq a$ **do** $i := i + 1$
    if $i \leq p\uparrow n$ **then return** $p\uparrow k[i]$ **else return** "ei ole"

36. Jos a on jokin solmun omista avaimista $p\uparrow k[i]$, niin etsittävä avain on sen oikeanpuoleisesta lapsesta $p\uparrow c[i+1]$ alkavan alipuun pienin avain. Alipuu on varmasti olemassa, koska käsittelemme parhaillaan tapausta jossa ei olla lehdessä, ja se on epätyhjä, koska B-puussa vain juuri voi olla tyhjä. Sen jokainen avain on suurempi kuin $p\uparrow k[i]$, koska avaimet ovat kasvavassa järjestyksessä ja luvattiin, että mikään avain ei toistu. Olemme tapauksessa $a = p\uparrow k[i]$. Niinpä $seuraava(p\uparrow c[i+1], a)$ löytää alipuun pienimmän avaimen. Siksi ei tarvita erillistä pienimmän avaimen etsimiskoodia.

    Jos a ei ole mikään solmun omista avaimista, niin etsittävä saattaa olla nykyisen ja edellisen avaimen välistä alkavassa alipuussa (tai solmun ensimmäisessä tai viimeisessä alipuussa, jos edellistä tai nykyistä avainta ei ole). Jos se ei ole siellä, ja nykyisessä solmussa on avaimia jäljellä, niin se on nykyisen solmun seuraava avain. Muussa tapauksessa etsittävää ei ole olemassa.

    Seuraavassa pseudokoodissa hyödynnetään sitä, että edellisessä vastauksessa on jo etsitty solmun ensimmäinen avain, joka on isompi kuin a.

    if $i > 1$ && $a = p\uparrow k[i-1]$ **then return** $seuraava(p\uparrow c[i], a)$
    $b :=$ $seuraava(p\uparrow c[i], a)$
    if $b \neq$ "ei ole" **then return** b
    if $i \leq p\uparrow n$ **then return** $p\uparrow k[i]$ **else return** "ei ole"

    Tätä voi yksinkertaistaa huomaamalla, että jos $seuraava(p\uparrow c[i], a)$ on olemassa, niin se palautetaan riippumatta ensimmäisellä rivillä olevan testin tuloksesta. Jos $seuraava(p\uparrow c[i], a)$ ei ole olemassa, niin alipuun $p\uparrow c[i]$ jokainen alkio on enintään a. Niinpä jos $p\uparrow k[i-1]$ on olemassa, niin se on pienempi kuin a, koska avaimet ovat järjestyksessä eivätkä toistu. Tällöin ensimmäisen rivin testi tuottaa false. Siksi kummassakaan tapauksessa tulos ei muutu, jos ensimmäinen rivi poistetaan.

    $b :=$ $seuraava(p\uparrow c[i], a)$
    if $b \neq$ "ei ole" **then return** b
    if $i \leq p\uparrow n$ **then return** $p\uparrow k[i]$ **else return** "ei ole"

    Koska alimmalle riville tullaan vain jos $b =$ "ei ole", voidaan kaksi alinta riviä yhdistää seuraaavasti. On makuasia, kumpi versio on selkeämpi.

    if $b \neq$ "ei ole" || $i > p\uparrow n$ **then return** b **else return** $p\uparrow k[i]$

37. Sen ansiosta, että edellisissä vastauksissa esiintyy sama rivi, ne voi yhdistää seuraavasti.

```
seuraava(p,a)
i := 1
while i ≤ p↑n && p↑k[i] ≤ a do i := i + 1
if not p↑l then
    b := seuraava(p↑c[i],a)
    if b ≠ "ei ole" then return b
if i ≤ p↑n then return p↑k[i] else return "ei ole"
```

38. Jos `a` ja `b` osoittavat samaan juurisolmuun, niin Männikön koodi laskee uudeksi puun kooksi alkuperäinen koko plus alkuperäinen koko, sijoittaa sen solmun ylös-linkkiin `pvJoukko[a]` ja lopuksi sijoittaa solmun ylös-linkkiin linkin solmusta itseensä suorittamalla `pvJoukko[b] = a;` . Tämän seurauksena seuraava `find` -toiminto samaan osajoukkoon jää ikuiseen silmukkaan, jossa se jatkaa osajoukon juurisolmusta itseensä.

39. Tässä tapauksessa viimeinen sijoitus olisi `pvJoukko[a] = k;` , joten ylöslinkkiin jäisi negatiivinen luku kuten pitääkin, mutta kaksinkertainen oikeaan verrattuna. Se vääristäisi nopeutusheuristiikka ja niin ollen saattaisi hidastaa ohjelmaa, mutta ei aiheuttaisi muuta haittaa.

40. Linkeillä alkiot yhdistetään rengaslistaksi. Kun alkio luodaan, linkki asetetaan osoittamaan alkioon itseensä. Find-toiminnossa linkeille ei tehdä mitään. Union-toiminnossa vaihdetaan yhdistettävien osajoukkojen edustajien linkit keskenään, eli `apu = linkki[a]; linkki[a] = linkki[b]; linkki[b] = apu;` . Tulostaminen tai muu sellainen tapahtuu selaamalla linkkien muodostama rengaslista ympäri.

41. Kustakin solmusta täytyy tietää, onko se löydetty. Helppo tapa toteuttaa tämä on sisällyttää solmun tietoihin bitti, joka kertoo, onko solmu löydetty. Tarvitaan tietorakenne, jossa on kaaria, ja josta saa nopeasti ulos mahdollisimman lyhyen kaaren. Prioriteettijono on siihen hyvä.

42. Primin algoritmi ei ole kiinnostunut kaarista, joiden kumpikaan pää ei ole vielä löydetty. Niinpä kaari kannattaa laittaa prioriteettijonoon vasta kun sen jompikumpi pää löytyy. (Tämä on olennainen ero Kruskalin algoritmiin, joka aloittaa kaarten käsittelyn laittamalla ne kaikki järjestykseen pituuden mukaan.)

43. Jos kaari on jo prioriteettijonossa kun sen toinenkin pää löytyy, on helpointa antaa sen olla siellä ja prioriteettijonosta kaaria otettaessa hylätä kaaret, joiden molemmat päät on löydetty. Tästä huolimatta ennen kaaren prioriteettijonoon laittamista kannattaa testata, onko sen toinenkin pää jo löydetty, jot-

ta samaa kaarta ei laitettaisi prioriteettijonoon toistamiseen. Testi on hyvin halpa ja nopeuttaa ohjelmaa vähentämällä prioriteettijonon työmäärää.

44.
```
void Prim(){
  if( solmut.size() < 2 ){ return; } // jottei käsiteltäisi olematonta solmua
  loyda_solmu( 1 );
  while( !lyhin_ensin.empty() ){
    unsigned eka = lyhin_ensin.top().eka, toka = lyhin_ensin.top().toka;
    lyhin_ensin.pop();
    if( !solmut[ toka ].loydetty ){
      std::cout << eka << "->" << toka << '\n';
      loyda_solmu( toka );
    }
  }
}
```

45. Vastaus 44 löytää mahdollisimman lyhyen virittävän puun sille mahdollisimman suurelle yhtenäiselle aligraafille, johon solmu 1 kuuluu. Jos lopetusehtona olisi, että jokainen solmu on löydetty, niin aligraafin virittävän puun valmistuttua se lopulta yrittäisi ottaa kaaria tyhjästä prioriteettijonosta. Se saattaa kaataa ohjelman, johtaa ikuiseen silmukkaan tai aiheuttaa jonkin muun virhetoiminnon.

46. Siihen on monta keinoa. Voidaan jälkikäteen käydä solmut läpi ja tarkastaa, että kaikki on löydetty. Algoritmin suorituksen aikana voidaan ylläpitää löydettyjen solmujen tai virittävään puuhun otettujen kaarten määrää ja tarkastaa, että löydettiin kaikki solmut tai otettiin mukaan solmujen määrä miinus yksi kaarta.

47. Se asettaa nykyisen solmun `loydetty`-bitin `true`:ksi ja lisää prioriteettijonoon ne nykyiseen solmuun liittyvät kaaret, joiden toinen pää ei ole vielä löydetty.

48. Solmutietueessa kannattaa olla löydetty-bitti ja solmuun liittyvien kaarten luettelo.

```
struct solmutyyppi{
  std::vector< kaarityyppi > kaaret;
  bool loydetty;
  solmutyyppi(): loydetty( false ) {}
};

std::vector< solmutyyppi > solmut;
```

49.
```
void loyda_solmu( unsigned slm ){
  solmutyyppi & nykyinen = solmut[ slm ];
```

```
  nykyinen.loydetty = true;
  for( unsigned ii = 0; ii < nykyinen.kaaret.size(); ++ii ){
    kaarityyppi & kaari = nykyinen.kaaret[ ii ];
    if( !solmut[ kaari.toka ].loydetty ){ lyhin_ensin.push( kaari ); }
  }
}
```

50.
```
unsigned max = eka > toka ? eka : toka;
if( max >= solmut.size() ){ solmut.resize( max+1 ); }
solmut[ eka ].kaaret.push_back( kaarityyppi( eka, toka, pituus ) );
solmut[ toka ].kaaret.push_back( kaarityyppi( toka, eka, pituus ) );
```

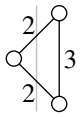
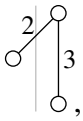
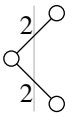
51. Muutetaan rivit

```
if( solmut.size() < 2 ){ return; } // jottei käsiteltäisi olematonta solmua
loyda_solmu( 1 );
...
```

muotoon

```
for( unsigned slm = 1; slm < solmut.size(); ++slm ){
  if( !solmut[ slm ].loydetty ){
    loyda_solmu( slm );
    ...
  }
}
```

52. Jatketaan kunnes jokainen kaari on tutkittu.

53. Graafista  saadaan  ja edelleen  , joka on pitempi kuin  .

54.
```
unsigned puuttuu = tavoite;
for( unsigned ii = arvot.size(); ii--; ){
  maarat[ ii ] = puuttuu / arvot[ ii ]; puuttuu %= arvot[ ii ];
}
```

55. Jos `arvot` $= [1, 6, 20, 100]$ ja tavoite on 24 hilua, niin ahne algoritmi valitsee 20, 1, 1, 1 ja 1 vaikka 6, 6, 6 ja 6 olisi parempi.

56.
```
unsigned rekursiivinen( unsigned puuttuu ){
  if( puuttuu == 0 ){ return 0; }
  unsigned paras = puuttuu+1;  // onnistuu alle tämän yhden hilun kolikoilla
  for( unsigned ii = 0; ii < arvot.size() && arvot[ ii ] <= puuttuu; ++ii ){
    unsigned uusi = rekursiivinen( puuttuu - arvot[ ii ] ) + 1;
    if( uusi < paras ){ paras = uusi; }
  }
  return paras;
}
```

57. Lisätään globaali muuttuja `std::vector< unsigned > valinta`. Muutetaan `paras = uusi;` muotoon `paras = uusi; valinta[ puuttuu ] = ii;`. Lisätään seuraava aliohjelma.

```
void rekurs_paa( unsigned puuttuu ){
  valinta.resize( puuttuu+1, 0 );
  rekursiivinen( puuttuu );
  while( puuttuu > 0 ){
    ++maarat[ valinta[ puuttuu ] ]; puuttuu -= arvot[ valinta[ puuttuu ] ];
  }
}
```

58. Määritellään `std::vector< unsigned > paras` samalla kuin `valinta`. Korvataan `rekursiivinen` seuraavasti.

```
unsigned muistava( unsigned puuttuu ){
  if( puuttuu == 0 ){ return 0; }
  if( !paras[ puuttuu ] ){
    paras[ puuttuu ] = puuttuu+1;  // onnistuu alle tämän 1 hilun kolikoilla
    for( unsigned ii = 0; ii < arvot.size() && arvot[ ii ] <= puuttuu; ++ii ){
      unsigned uusi = muistava( puuttuu - arvot[ ii ] ) + 1;
      if( uusi < paras[ puuttuu ] ){
        paras[ puuttuu ] = uusi; valinta[ puuttuu ] = ii;
      }
    }
  }
  return paras[ puuttuu ];
}
```

Muutetaan `rekurs_paa` alkamaan seuraavasti.

```
void muistava_paa( unsigned puuttuu ){
  paras.resize( puuttuu+1, 0 ); valinta.resize( puuttuu+1, 0 );
  muistava( puuttuu );
```

59.
```
void taulukoiva( unsigned puuttuu ){
  std::vector< unsigned > paras( puuttuu+1, 0 ), valinta( puuttuu+1, 0 );
  for( unsigned jj = 1; jj <= puuttuu; ++jj ){
    paras[ jj ] = puuttuu+1;  // onnistuu alle tämän yhden hilun kolikoilla
    for( unsigned ii = 0; ii < arvot.size() && arvot[ ii ] <= jj; ++ii ){
      unsigned uusi = paras[ jj - arvot[ ii ] ] + 1;
      if( uusi < paras[ jj ] ){ paras[ jj ] = uusi; valinta[ jj ] = ii; }
    }
  }
  while( puuttuu > 0 ){
    ++maarat[ valinta[ puuttuu ] ]; puuttuu -= arvot[ valinta[ puuttuu ] ];
  }
}
```

60. Muutetut kohdat ovat sinisellä.

```
for i := 0 to n do D[i,0] := 2i
for j := 1 to m do D[0,j] := 3j
for i := 1 to n do
    for j := 1 to m do
        if A[i] = B[j] then D[i,j] := D[i−1,j−1]
        else D[i,j] := min{2+D[i−1,j],3+D[i,j−1]}
return D[n,m]
```

61. $x+y$

Viereinen taulukko havainnollistaa laskutapaa. Jos rivillä ja sarakkeella on sama kirjain, niin editointietäisyyden laskeminen rivin ja sarakkeen leikkauskohtaan käyttää vinosti vasemmalla ylhäällä olevan alkion editointietäisyyttä. Muussa tapauksessa se käyttää suoraan vasemmalla ja suoraan yläpuolella olevia editointietäisyyksiä.

		k	t
	↖		↑
k			$\leftarrow x$
l	↑	↑	↑
	$\leftarrow y$		$\leftarrow z$

62. $y+z$

63. Ei yhtään. Kuten vastausten 13, ..., 16 perusteella näyttää ja algoritmien toimintaa tutkimalla on pääteltävissä, niin jos $n > 0$, niin kutsun `kerto`(n,m) suorituksen aikana ei kertaakaan kysytä uudelleen tulosta, joka on jo laskettu. Kun `kerto` :a kutsutaan uudelleen, on jälkimmäinen argumentti pienentynyt, ja kun `plus` :saa kutsutaan uudelleen, on jälkimmäinen argumentti pienentynyt tai ensimmäinen kasvanut.

64. Monella pisteellä voi olla sama x-koordinaatti, joten voi käydä niin, että jollakin vasemmanpuoleisen joukon pisteellä on sama x-koordinaatti kuin jollakin oikeanpuoleisen joukon pisteellä. Niin käy esimerkiksi, jos pisteet ovat $(2,3)$, $(4,1)$, $(4,5)$ ja $(6,1)$.

65. `return piste.x > raja.x || ( piste.x == raja.x && piste.y >= raja.y );`

Vastaus toimisi myös jos y-koordinaatteja verrattaisiin eri suuntaan kuin x-koordinaatteja, mutta se saattaisi hämätä ohjelman ylläpitäjiä.

66. Algoritmi kannattaa laittaa lopettamaan heti ja raportoimaan mainitut kaksi pistettä pisteinä, joiden välinen etäisyys on mahdollisimman pieni. Koska niillä on sama x-koordinaatti ja sama y-koordinaatti, on niiden välinen etäisyys 0. Millään pisteparilla ei voi olla sitä pienempi etäisyys, koska etäisyys ei voi olla negatiivinen.

67.
```
unsigned puolet = xJarj.size() / 2;
for( unsigned ii = 0; ii < puolet; ++ii ){
  xJarjVasen.push_back( xJarj[ ii ] );
```

```
}
for( unsigned ii = puolet; ii < xJarj.size(); ++ii ){
  xJarjOikea.push_back( xJarj[ ii ] );
}
double raja_x = xJarjOikea[0].x, raja_y = xJarjOikea[0].y;
for( unsigned ii = 0; ii < yJarj.size(); ++ii ){
  const pistetyyppi & piste = yJarj[ ii ];
  if( piste.x > raja_x || ( piste.x == raja_x && piste.y >= raja_y ) ){
    yJarjOikea.push_back( piste );
  }else{
    yJarjVasen.push_back( piste );
  }
}
```

68. Järjestämisalgoritmin on katsottava jokainen alkio, jotta se voi olla varma, että se ei ole pienempi kuin edellinen alkio tai suurempi kuin seuraava alkio. Siihen menee aikaa $\Omega(n)$.

69. Kyllä voi. Esimerkiksi puolitushaku on aina (ja siksi myös nopeimmillaan) $O(\log n)$. Sen ei tarvitse katsoa jokaista alkiota voidakseen olla varma vastauksesta.

70. Lisätään algoritmin eteen vaihe, joka selaa taulukon läpi ja tarkastaa, onko se valmiiksi järjestyksessä. Varsinainen järjestämisalgoritmi suoritetaan vain jos taulukko ei ole järjestyksessä. Järjestyksessä olevien taulukoiden osuus kaikista taulukoista on hyvin pieni. Siksi tällainen vaihe melko varmasti kuluttaisi enemmän aikaa kuin säästäisi, joten sitä ei kannata toteuttaa. Se kannattaa toteuttaa vain jos on erityinen syy uskoa, että syötteenä saatava taulukko on melko usein valmiiksi järjestyksessä.

71. Termit ovat 1000, $100n$, $10n^2$ ja $n\log n$. Niistä eniten merkitsee $10n^2$, koska kun $n > 10$, niin sen tuottama arvo on muiden tuottamia suurempi. Niinpä $f(n) = O(n^2)$.

72. Termit $-3n$ ja $8n$ ovat samanmuotoiset. Yhdistettäessä ne tuottavat $5n$. Termit $2n^2$ ja $-2n^2$ ovat samanmuotoiset. Yhdistettäessä ne kumoavat toisensa. Jäljelle jää $5n+7$. Kun $n \geq 2$ on $5n > 7$, joten eniten merkitsevä termi on $5n$. Niinpä $f(n) = O(n)$.

73. Tehtävän lausekkeiden termeistä vakiot merkitsevät vähiten. Sen jälkeen vähiten merkitsevät tässä järjestyksessä ne, joista vakiokerroin pois jätettynä tulee $\log n$, $\sqrt{n}$, n, $n\log n$, $n\sqrt{n}$, n^2 ja n^3. Sulut auki kertomalla lausekkeesta $\frac{1}{2}n(n+1)$ tulee $\frac{1}{2}n^2 + \frac{1}{2}n$.

$f_1(n) = O(n^2)$ $\qquad$ $f_4(n) = O(n)$

$f_2(n) = O(n \log n)$ $\quad$ $f_5(n) = O(n^2)$

$f_3(n) = O(n^3)$ $\quad$ $f_6(n) = O(n\sqrt{n})$

$(1,1)$ $(1,3)$ $(1,5)$ $(2,1)$ $(2,2)$ $(2,3)$ $(2,5)$ $(2,6)$ $(3,3)$ $(4,1)$ $(4,2)$ $(4,3)$ $(4,4)$ $(4,5)$ $(4,6)$ $(5,1)$ $(5,3)$ $(5,5)$ $(6,1)$ $(6,3)$ $(6,5)$ $(6,6)$

74. O-, Ω- ja Θ-merkintöjen määritelmät (esim. Männikön luento 1 ruudut 14–16) vaativat lausekkeen esittämältä funktiolta sen minkä vaativat vain jostakin n:n arvosta alkaen. Ne eivät edes kerro, mikä on tämä n:n arvo, josta alkaen vaatimusten pitää toteutua, vaan sanovat ainoastaan, että sellaisen täytyy olla olemassa. Niinpä pienillä n:n arvoilla saa tapahtua mitä tahansa, kunhan jossakin on raja, jossa huono käyttäytyminen loppuu. Siksi ei haittaa, että $\log n$ käyttäytyy huonosti kun $n = 0$. Se käyttäytyy hyvin kun $n \geq 1$, ja se riittää.

    Koska O-, Ω- ja Θ-merkintöjen kannalta on merkityksetöntä, mitä tapahtuu rajaa pienemmillä n:n arvoilla, ei algoritmin suoritusaikaa tai muistin käyttöä kuvaavaa lauseketta aina viitsitä kirjoittaa niin, että se olisi oikein pienillä n:n arvoilla. Suurillakaan n:n arvoilla sen ei tarvitse olla tarkalleen oikein, koska merkinnät sallivat vaihtelun vakiokertoimen rajoissa. Siksi, kun kaikilla n:n arvoilla oikein olisi esimerkiksi $\lceil \log(n+1) \rceil$, usein kirjoitetaankin $\log n$, sillä se on yksinkertaisempi ja suurilla n:n arvoilla riittävän tarkka.

75. Yläraja on $O(n)$. Se on pätevä, koska millään n:n arvolla funktion arvo ei ole suurempi kuin $6n+6$. Sitä tarkempaa ei ole, koska äärettömän monella n:n arvolla funktion arvo on $6n+6$.

    Alaraja on $\Omega(1)$. Se on pätevä, koska jokaisella n:n arvolla funktion arvo on vähintään 3. Sitä tarkempaa ei ole, koska äärettömän monella n:n arvolla funktion arvo on 3.

76. Ei voi. Jos $g(n)$ kasvaa hitaammin kuin n, niin $3(1^n + (-1)^n)(n + \frac{1}{2}) + 3$ ei ole $O(g(n))$, koska se tuottaa $6n+6$ äärettömän monella n:n arvolla. Jos $g(n)$ kasvaa samaa vauhtia tai nopeammin kuin n, niin $3(1^n + (-1)^n)(n + \frac{1}{2}) + 3$ ei ole $\Omega(g(n))$, koska se tuottaa 3 äärettömän monella n:n arvolla. Ei siis ole olemassa kasvavaa funktiota $g(n)$, jolle $3(1^n + (-1)^n)(n + \frac{1}{2}) + 3$ olisi sekä $O(g(n))$ että $\Omega(g(n))$. Mutta funktio on $\Theta(g(n))$ jos ja vain jos se kasvaa enintään ja vähintään samaa vauhtia kuin $g(n)$, eli jos ja vain jos se on sekä $O(g(n))$ että $\Omega(g(n))$ (esim. Männikön luento 1 ruutu 20).

77. Tehtävän algoritmi selaa taulukkoa kunnes se loppuu tai kohdataan alkio, jonka arvo on x. Hitaimmillaan taulukko joudutaan selaamaan loppuun saak-

ka, jolloin aikaa kuluu n:ään verrannollisesti. Niinpä mahdollisimman tarkka hitaimman tapauksen, ja samalla jokaisen tapauksen, yläraja on $O(n)$. Nopeimmillaan etsitty löytyy ensimmäisestä alkiosta, jolloin aikaa kuluu vakion verran eli ajan kulutus ei riipu n:stä. Niinpä mahdollisimman tarkka nopeimman tapauksen, ja samalla jokaisen tapauksen, alaraja on $\Omega(1)$. Koska ne ovat erit, jokaisen tapauksen ajan kulutusta ei voi ilmaista Θ-merkinnällä.

78. Tehtävän algoritmin suoritusaika ei riipu muusta kuin n:stä. Siksi hitaimman, nopeimman ja jokaisen tapauksen suoritusaika on sama. Jokaisella kierroksella i suunnilleen kaksinkertaistuu. Algoritmi lopettaa, kun $i \geq n$. Niinpä kierrosten määrä on suunnilleen $\log_2 n$. Siksi jokaisen tapauksen suoritusaika on $O(\log_2 n)$, $\Omega(\log_2 n)$ ja $\Theta(\log_2 n)$. Koska eri logaritmit eroavat toisistaan vain vakiokertoimella, ei O-, Ω- ja Θ-merkintöjen kannalta ole väliä, mikä kantaluku logaritmilla on, kunhan se on enemmän kuin yksi. Siksi jokaisen tapauksen suoritusaika on $O(\log n)$, $\Omega(\log n)$ ja $\Theta(\log n)$.

79. Jos $n > 1$, niin tehtävän algoritmi menee **while**-silmukkaan ja suorittaa sen ensimmäisen kierroksen aikana $n-1$ kierrosta **for**-silmukkaa. Siksi jokaisen tapauksen suoritusaika on $\Omega(n)$. Jos $n > 1$ ja A:n jokainen alkio on 0, niin algoritmi lopettaa **while**-silmukan yhden kierroksen jälkeen. Tällöin suoritusaika on $O(n)$, koska **while**-silmukan sisällä oleva **for**-silmukka tekee $n-1$ vakioaikaista testiä ja lopettaa, ja kaikki muu **while**-silmukassa ja sen edessä on vakioaikaista. Siksi $\Omega(n)$ on mahdollisimman tarkka alaraja.

    Koska y saa arvokseen $z-1$ ja sen jälkeen z saa arvokseen 0 tai jonkin luvun väliltä $1, \ldots, y$, pienenee z joka kierroksella ainakin yhdellä. Niinpä **while**-silmukan kierrosten määrä ei voi olla enempää kuin $n-1$ eikä **for**-silmukan kierrosten yhteismäärä voi olla enempää kuin $(n-1)+(n-2)+\ldots+2+1 = O(n^2)$. Jokainen yksittäinen toiminto algoritmissa on vakioaikainen. Siksi jokaisen tapauksen suoritusaika on $O(n^2)$. Jos $A = [n, n-1, \ldots, 2, 1]$, niin jokainen **if**-lauseen ehdon suoritus tuottaa `true`, joten z todella saa arvot $n-1, n-2, \ldots, 1$ ja suoritusaika on $\Omega(n^2)$. Siksi $O(n^2)$ on mahdollisimman tarkka yläraja.

    Koska mahdollisimman tarkalla ala- ja ylärajalla on eri kasvunopeus, ei ole Θ-merkintää joka pätee jokaiselle tapaukselle.

80. Ei välttämättä. Tekstikappale voi muodostua yhdestä tai useammasta rivistä, joissa on välilyöntejä mutta ei muita merkkejä.

81. ```
    /* Etsi sanan lopunohikohta. */
    while( paikka < rivi.size() && rivi[ paikka ] != ' ' ){ ++paikka; }
    ```
    ```

82. `yksi`
    `lma`

83. Muuttujaa `paikka` ei nollata muulloin kuin ohjelman suorituksen alussa. Kun rivi `yksi` on käsitelty, pätee `paikka` $= 4$. Niinpä seuraavan rivin `ohjelma` käsittely ei ala rivin alusta, vaan paikasta 4. Siksi merkit `o`, `h`, `j` ja `e` jäävät käsittelemättä, ja käsitellään vain `l`, `m` ja `a`.

84. Lähellä alkua olevan `while( true ){` ja ensimmäisen `while( paikka < rivi.size() ...` väliin.

85. Jos rivin lopussa on välilyöntejä, niin se tunnistaa rivin lopussa tyhjän merkkijonon sanaksi.

86. ```
    /* Selaa rivi. */
    while( true ){

      /* Etsi seuraavan sanan alku. Jos sitä ei ole, poistu silmukasta. */
      while( paikka < rivi.size() && rivi[ paikka ] == ' ' ){ ++paikka; }
      if( paikka >= rivi.size() ){ break; }
      unsigned alku = paikka;

      /* Etsi sanan lopunohikohta. */
      while( paikka < rivi.size() && rivi[ paikka ] != ' ' ){ ++paikka; }

      /* Tulosta sana. */
      std::cout << rivi.substr( alku, paikka - alku ) << "\n";

    }
    ```

87. Se pitää laittaa ennen ensimmäistä `while( true ){`. Jos sen laittaa ulomman `while`-silmukan sisään, niin se ei ole käytettävissä ohjelman lopussa, jossa pisin sana pitää tulostaa.

    Sen voi periaatteessa laittaa ennen pääohjelmaa, jolloin `pisin_sana`:sta tulee niin sanottu globaali muuttuja eli kaikkien aliohjelmien, luokkien ja olioiden ulkopuolella sijaitseva muuttuja. Se kannattaa kuitenkin laittaa pääohjelman sisään, koska yleensä suositellaan laittamaan muuttujat siten, että ne eivät olisi käytettävissä siellä missä niitä ei tarvita. Se pienentää riskiä, että muuttujaan sijoitetaan vahingossa jotakin, koska se sekottui toiseen samannimiseen tai koska ohjelmoija kirjoitti sen nimen vahingossa kun tarkoitti toista nimeä.

88. ```
    /* Jos löytyi pitempi sana kuin aiemmat, niin talleta se. */
    if( paikka - alku > pisin_sana.size() ){
      pisin_sana = rivi.substr( alku, paikka - alku );
    }
    ```

89. Jos syötteessä ei ole yhtään sanoja, niin ohjelma tulostaa pisimmäksi sanaksi tyhjän merkkijonon. Se on vastoin vaatimuksia. Pisin sana piti tulostaa vain jos syötteessä on ainakin yksi sana.

90. `/* Jos saatiin ainakin yksi sana, niin tulosta pisin sana. */`
`if( !pisin_sana.empty() ){ std::cout << pisin_sana << "\n"; }`

91. Pitää lisätä muuttujat tallettamaan pisimmän sanan rivinumero ja sarake. Rivinumeron tallettamista varten pitää lisätä muuttuja laskemaan luettujen rivien määrä. Näille muuttujille pitää lisätä ulomman `while` -silmukan eteen esittely, jossa ne alustetaan nollaksi. Rivinumerolaskurin kasvatukselle luonteva paikka on heti rivin `if( !std::cin ){ break; }` jälkeen. Rivin ja sarakkeen numeroiden talletus kuuluu tietenkin siihen `if` -lauseeseen, jonka ehto on `paikka - alku > pisin_sana.size()` . Pisimmän sanan tulostuslauseeseen pitää lisätä rivinumeron ja sarakkeen tulostaminen.

92. `ps_rivi = vr_rivi; ps_sarake = alku + 1;`

    Luvun 2.1 mukaan rivit ja sarakkeet numeroidaan ykkösestä alkaen. Muuttujan `vr_rivi` arvo on tämän mukainen, koska se on alustettu nollaksi ja sitä kasvatetaan yhdellä aina kun luetaan rivi. Toisaalta `alku` käyttää C++:n mukaista indeksointia, jossa merkkijonon ensimmäisen merkin indeksi on 0. Jotta siitä saataisiin sarakkeen numero on lisättävä 1, koska sarakkeet numeroidaan ykkösestä alkaen.

93. Koska välilyönti on ASCII-merkki, se on sellaisenaan UTF-8-koodattu Unicode-merkki. Niinpä muuttujaa `uc_paikka` kuuluu kasvattaa jokaisen välilyönnin kohdalla.

    ```
    /* Etsi seuraavan sanan alku. Jos sitä ei ole, poistu silmukasta. */
    while( paikka < vr_merkit.size() && vr_merkit[ paikka ] == ' ' ){
      ++paikka; ++uc_paikka;
    }
    ```

94. Jotta ohjelma voisi tulostaa pisimmän sanan sarakkeen Unicode-merkkeinä laskettuna, sen kannattaa ottaa talteen pisimmän sanan alkukohta myös Unicode-merkkeinä laskettuna, jotta sitä ei tarvitsisi laskea tulostusta varten uudelleen. Niinpä tässä kohdassa esitellään uusi muuttuja `uc_alku` ja alustetaan sen arvoksi `uc_paikka` .

95. Sitä pitää kasvattaa kerran jokaista Unicode-merkkiä kohti. Se onnistuu helposti kasvattamalla sitä kunkin Unicode-merkin ensimmäisen tavun kohdalla.

    ```
    if( uc_alkutavu( vr_merkit[ paikka ] ) ){ ++uc_paikka; }
    ```

96. Pisintä sanaa päivitettäessä sanojen pituuksia ei pidä verrata tavujen mukaan, vaan Unicode-merkkien mukaan. Siksi tarkasteltavan sanan pituutena ei saa käyttää `paikka - alku` vaan `uc_paikka - uc_alku`.

    Samasta syystä pisimmän aikaisemmin löydetyn sanan pituutena ei saa käyttää tavujen määrää. Pisimmän aikaisemmin löydetyn sanan Unicoden mukaisen pituuden laskeminen aina tarvittaessa voi hidastaa ohjelmaa merkittävästi, koska jokaisen löydetyn sanan Unicoden mukaista pituutta verrataan siihen. Niinpä jos tarkasteltavasta sanasta tulee uusi pisin siihen mennessä löydetty sana, kannattaa ottaa talteen sille laskettu `uc_paikka - uc_alku`. Se kelpaa myöhempiin vertailuihin sellaisenaan. Sitä varten tarvitsee esitellä ja alustaa muuttuja. Annamme sille nimeksi `ps_uc_pituus`.

    Pisimmän sanan sarakkeeksi pitää asettaa `uc_alku + 1` eikä `alku + 1`.

97. Ohjelman tulostamat rivinumerot ovat selvästi väärin. Niiden kuuluisi koko ajan kasvaa, mutta ne pienenevät välillä.

98. Koska `vr_rivi` on esitelty aliohjelman `yksi_kappale` sisällä, se ei säilytä arvoaan aliohjelman kutsusta toiseen. Helpoin korjaus olisi siirtää sen esittely aliohjelman `yksi_kappale` eteen. Mutta silloin siitä tulisi globaali muuttuja, ja monien mielestä globaalien muuttujien käyttö on epäsuotavaa. Siksi sijoitamme sen pääohjelman sisään ja välitämme aliohjelmalle parametrina. Jotta sen arvon muutokset välittyisivät pääohjelmalle, käytämme viiteparametria. (Luvussa 2.7 joudumme miettimään uudelleen, onko tämä hyvä ratkaisu.)

    ```
    void yksi_kappale( unsigned & vr_rivi ){
      ...
    }

    /* Pääohjelma: käsittele jokainen kappale. */
    int main(){
      unsigned vr_rivi = 0;
      while( std::cin ){ yksi_kappale( vr_rivi ); }
    }
    ```

99. Nyt rivinumerot johdonmukaisesti kasvavat, mutta jäävät jälkeen oikeista.

100. Koska `++vr_rivi;` on väärässä kohdassa, tyhjät rivit jäävät laskematta.

101. Ainakin yksi kokonaisluku on saatu jos ja vain jos `pl_rivi` ei ole nolla. Tämä edellyttää, että `pl_rivi` on alustettu nollaksi, mutta se on helppo järjestää. Myös muuttujaa `pl_sarake` voi käyttää samalla tavalla.

102. Jotta se olisi käytettävissä aliohjelmassa `yksi_kappale` ja säilyttäisi arvonsa aliohjeman kutsujen välissä, se pitää esitellä joko aliohjelman edessä tai tuoda sille viiteparametrina.

103. ```
/* Ylläpidä pienintä löydettyä kokonaislukua. */
if(lukuarvo < pl_arvo || pl_rivi == 0){
  pl_arvo = lukuarvo; pl_rivi = vr_rivi; pl_sarake = uc_luku + 1;
}
```

Koska `pl_arvo` on alustettu nollaksi, ilman osuutta `|| pl_rivi == 0` ohjelma ottaisi huomioon vain negatiivisia lukuja. Tämä korjaantuisi vain osittain alustamalla `pl_arvo` johonkin muuhun arvoon, koska testin `lukuarvo < pl_arvo` vuoksi ohjelma jättäisi ottamatta huomioon ainakin sen arvon, johon `pl_arvo` on alustettu. Jos testiksi muutettaisiin `lukuarvo <= pl_arvo`, niin ohjelma raportoisi pienimmän kokonaisluvun viimeisen eikä ensimmäistä esiintymää.

”`+ 1`” on selitetty vastauksessa 92.

104. ```
on_numero( vr_merkit[ paikka ] ) || vr_merkit[ paikka ] == '-' ||
vr_merkit[ paikka ] == '+'
```

105. ```
/* Etsi sanan lopunohikohta. */
while(paikka < vr_merkit.size() && vr_merkit[paikka] != ' '){
  char merkki = vr_merkit[paikka];
  if(on_numero(merkki) || merkki == '-' || merkki == '+'){

    // Laske lukuarvo.

    /* Ylläpidä pienintä löydettyä kokonaislukua. */
    if(lukuarvo < pl_arvo || pl_rivi == 0){
      pl_arvo = lukuarvo; pl_rivi = vr_rivi; pl_sarake = uc_luku + 1;
    }

  }else{
    ++paikka;
    if(uc_alkutavu(merkki)){ ++uc_paikka; }
  }
}
```

106. ```
++paikka; ++uc_paikka;
while( paikka < vr_merkit.size() && on_numero( vr_merkit[ paikka ] ){
  ++paikka; ++uc_paikka;
}
```

107. Se toimii väärin esimerkiksi kun syötteessä on vain yksi rivi, ja siinä lukee pelkästään `linja-auto`. Se tulkitsee merkin `-` etumerkiksi ja yrittää muodostaa lukuarvon. Jos se tuottaa minkä tahansa lukuarvon, niin se menee

muuttujaan `lukuarvo`. Sieltä se jatkaa muuttujaan `pl_arvo`, ja `pl_rivi` saa arvon 1. Lopulta se tulostuu ohjelman vastauksena, kun pitäisi tulostaa `ei kokonaislukuja`. Jos ohjelma kaatuu yrittäessään tuottaa lukuarvon merkkijonosta `-`, niin silloinkaan se ei toimi oikein.

108. On se, ainakin Oracle:n Java-dokumentaation mukaan. Tarkka kuvaus on kohdassa `parseInt(String s, int radix)`, ja kohdassa `parseInt(String s)` luvataan sen toimivan kuten `parseInt(s, 10)`. Huomaa että kaiken saaminen ihan oikein on niin vaikeaa, että se ei onnistu edes ohjelmointikielen spesifikaation laatijoilta: siellä lukee "an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value" ikään kuin `-0` olisi kielletty tai tuottaisi negatiivisen arvon, mutta esimerkeissä lukee `parseInt("-0", 10) returns 0`.

109. ```
/* Palauta true, jos ja vain jos merkki on numeromerkki. */
bool on_numero(char merkki){ return '0' <= merkki && merkki <= '9'; }
```

110. Kymmenellä kertominen.

111. Lisätään lukuun $d$.

112. ```
int tulos = 0;
while( paikka < merkit.size() && on_numero( merkit[ paikka ] ) ){
  tulos *= 10; tulos += merkit[ paikka ] - '0'; ++paikka;
}
```

113. Jos ei edetä yhtään merkkiä, niin ohjelma jää ikuiseen silmukkaan sellaisen `+` tai `-` kohdalla, joka ei aloita kokonaislukua. Jos edetään kaksi merkkiä, niin esimerkiksi tapauksessa `+-3` ohjelma jättää huomiotta etumerkin `-`. Siinä `+` ei aloita mutta `-` aloittaa kokonaisluvun. Jos edetään vielä useampi merkki, niin koko `-3` jää huomiotta.

114. ```
/* Palauta kokonaisluku, joka alkaa parametrin merkit kohdasta paikka.
  Jos saatiin kokonaisluku, niin kasvata paikka sen ohi. */
int hae_luku(const std::string & merkit, unsigned & paikka){
  if(paikka >= merkit.size()){ return 0; }
  unsigned ii = paikka;
  if(merkit[ii] == '-' || merkit[ii] == '+'){ ++ii; }
  int tulos = 0; unsigned jj = ii;
  while(ii < merkit.size() && on_numero(merkit[ii])){
    tulos *= 10; tulos -= merkit[ii] - '0'; ++ii;
  }
  if(merkit[paikka] != '-'){ tulos = -tulos; }
  if(ii > jj){ paikka = ii; }
  return tulos;
}
```
```

115.
```
/* Etsi sanan lopunohikohta. */
while( paikka < vr_merkit.size() && vr_merkit[ paikka ] != ' ' ){

  /* Jollei vuorossa ole luvun aloittava merkki, niin etene sanassa. */
  char merkki = vr_merkit[ paikka ];
  if( !on_numero( merkki ) && merkki != '-' && merkki != '+' ){
    ++paikka;
    if( uc_alkutavu( merkki ) ){ ++uc_paikka; }
    continue;
  }

  /* Yritä hakea lukuarvo. */
  unsigned luvun_alku = paikka, uc_luku = uc_paikka;
  int lukuarvo = hae_luku( vr_merkit, paikka );
  if( paikka == luvun_alku ){ ++paikka; ++uc_paikka; continue; }
  uc_paikka += paikka - luvun_alku;

  /* Ylläpidä pienintä löydettyä kokonaislukua. */
  if( lukuarvo < pl_arvo || pl_rivi == 0 ){
    pl_arvo = lukuarvo; pl_rivi = vr_rivi; pl_sarake = uc_luku + 1;
  }

}
```

116.
```
/* Ohita välilyönnit. */
void ohita_valilyonnit(){
  if( virhe ){ return; }
  while( rivi[ pkka ] == ' ' ){ ++pkka; }
  alku = pkka;
}

/* Ohita tyhjä tila. */
void ohita_tyhja(){
  ohita_valilyonnit();
  while( rivi[ pkka ] == '\n' ){
    uusi_rivi();
    if( virhe || loppu ){ return; }
    ohita_valilyonnit();
  }
}
```

Koska `ohita_valilyonnit` ja `uusi_rivi` eivät muuta mitään syöteluokan ollessa virhetilassa, ja koska `ohita_tyhja` muuttaa mitään vain kutsumalla niitä, ei aliohjelmassa `ohita_tyhja` tarvitse testata virhetilaa muuta kuin aliohjelman suorituksen lopettamiseksi. Myöskään muuttujien `alku` ja `pkka` arvoja ei tarvitse asettaa, koska juuri ennen aliohjelman lopettamista on suoritettu `ohita_valilyonnit` tai `uusi_rivi` , ja ne ovat huolehtineet muuttujien `alku` ja `pkka` arvoista.