REKURSIIVISTEN ALGORITMIEN TOTEUTTAMINEN LABVIEW-OHJELMOINTIYMPÄRISTÖSSÄ

Paavo Pakoma
Pro gradu -tutkielma
Tietojenkäsittelytiede
Kuopion yliopisto
tietojenkäsittelytieteen laitos
Tammikuu 2008

KUOPION YLIOPISTO, informaatioteknologian ja kauppatieteiden tiedekunta
Tietojenkäsittelytieteen koulutusohjelma

PAKOMA PAAVO, K.: Rekursiivisten algoritmien toteuttaminen LabVIEW-ohjelmointiympäristössä
Pro gradu -tutkielma, 70 s., 5 liitettä (8 s.), CD-ROM
Pro gradu -tutkielman ohjaaja: FM Maija Marttila-Kontio
Tammikuu 2008

Avainsanat: LabVIEW, rekursio, iteraatio, visuaalinen ohjelmointi, tietovuo.

LabVIEW on visuaalinen tietovuo-ohjelmointiympäristö, joka käyttää graafista G-kieltä. Vaikka se sallii aliohjelmarutiinit, ei rekursiivisten algoritmien toteuttaminen ole suoraan mahdollista. Tämä johtuu siitä, että virtuaali-instrumentit kuvakkeineen muodostavat suunnatun syklittömän graafin.

Tässä tutkielmassa etsitään ratkaisumalleja algoritmien rekursiiviseen tai rekursiivisluonteiseen toteuttamiseen LabVIEW-ympäristössä. Laitetasolla rekursiiviset kutsut muodostavat aktivaatiotietueen, joka on pino. Tätä tietoa hyväksikäyttäen tutkielmassa tehtiin pinorakenteeseen perustuvia rekursiivisia ratkaisuja, käyttäen hyväksi toisto- ja taulukkorakenteita.

LabVIEW-ympäristöön on rakennettu mekanismeja, joiden avulla myös tekstuaalisen ohjelmointikoodin liittäminen siihen on mahdollista. Call Library Function Node- ja Code Interface Node -kuvakkeiden avulla rekursiivisten algoritmien toteuttaminen onnistuu LabVIEW-ympäristössä.

LabVIEW-ympäristössä on myös valmiita elementtejä, joilla aikaansaadaan rekursio. VI Server -funktiot on tarkoitettu alunperin palvelinkäyttöön, mutta hitaudestaan huolimatta rekusiiviset kutsut toteutetaan palvelinkutsujen tapaan.

Jokaisella yllämainitulla menetelmällä toteutettiin testausmielessä rekursiivinen pikalajittelualgoritmi. Ratkaisujen nopeuksia lajittelussa verrattiin toisiinsa ja verrokkina olleeseen LabVIEW-ympäristön omaan lajittelualgoritmiin. Ratkaisujen nopeuserot olivat huomattavat. Ainoastaan tekstuaaliseen ohjelmakoodiin perustuvat algoritmit päihittivät nopeudessa verrokkiratkaisun.

Tutkielman yhtenä päätelmänä todetaan, ettei rekursiivisia ratkaisuja kannata tehdä, ellei se ole aivan välttämätöntä. Iteratiivinen ratkaisu on LabVIEW -ympäristössä helpommin toteutettavissa ja se on usein myös helppotajuisempi. Kun halutaan optimoida ohjelmaa, voidaan rekursiota käyttää tuomalla ulkopuolista tekstuaalista koodia virtuaali-instrumenttiin.

Esipuhe

Tämä tutkielma on tehty Kuopion yliopiston tietojenkäsittelytieteen laitokselle helmikuun 2005 ja tammikuun 2008 välisenä aikana. Tutkielman ohjaajana toimi Maija Marttila-Kontio, jolle haluan osoittaa erityiskiitoksen minua ja tutkielmaani kohtaan osoittamasta kärsivällisyydestä. Kiitän Mauno Rönkköä ja Konstantin Hyppöstä, jotka auttoivat minua rekursion teoriaa koskevissa kysymyksissä. Haluan kiittää myös tietojenkäsittelytieteen laitoksen henkilökuntaa, siitä vuosia kestäneestä valtavasta tuesta ja kiinnostuksesta tätä tutkielmaa kohtaan. Ilman oman perheen tukea ei opiskelu ja tämä, tältäosin viimeisin, koitos olisi koskaan ollut mahdollista. Kiitän vielä vanhempaa ja nuorempaa sisartani siitä, etteivät he saaneet omia tutkielmiaan valmiiksi ennen minua.

Kuopiossa 08.01.2008

Paavo Pakoma

Käsitteet ja lyhenteet

Block Diagram	Kaaviodiagrammi, LabVIEW:n lähdekoodin sijaintipaikka
Connector pair	Liitinpari, kuvakkeen langoituspiste
Data flow	Tietovuo
Front panel	Etupaneeli, LabVIEW:n käyttöliittymä
Icon	Kuvake
Iteraatio	Toistoon perustuva ongelmanratkaisu
Kognitiivinen	Ihmisen tiedonkäsittelyyn liittyvät toiminnot
LabVIEW	Visuaalinen ohjelmointiympäristö
Rekursio	Funktion itsensäkutsuun perustuva ongelmanratkaisu
Semantiikka	Merkitysoppi eli ohjelmointikielen lauseiden merkitys
Solmu	LabVIEW:n langoitetava kuvake
SubVI	LabVIEW:n aliohjelmarutiini
Syntaksi	Lauseoppi eli ohjelmoitikielen lauseiden tunnistus
UML	Unified Modelling Language
Wire	Kuvakkeiden välinen langoitus
VI	Virtuaali-instrumentti
Virtuaali-instrumentti	LabVIEW ohjelma

Sisältö

1 JOHDANTO **6**

2 VISUAALINEN OHJELMOINTI JA LabVIEW **8**

2.1 Historiaa 8

2.2 Visuaalinen kieli 9

2.3 LabVIEW 12

3 REKURSIO **16**

3.1 Iteraatio ja rekursio 16

3.2 Rekursion teoriaa 19

3.3 LabVIEW ja rekursio 24

4 REKURSION TOTEUTUS JA TESTAUS **30**

4.1 Pikalajittelu 30

4.2 Pikalajittelu VI Serverin avulla 35

4.3 LabVIEW ja tekstuaalinen ohjelmakoodi 39

4.3.1 Call Library Function Node -pikalajittelu 42

4.3.2 Code Interface Node (CIN) -pikalajittelu 47

4.4 Pinoperustainen pikalajittelu 51

4.5 Rekursiivisten ratkaisujen nopeustesti 54

5 POHDINTA **58**

VIITTEET **61**

A VI Server pikalajittelun LabVIEW koodi **63**

B Call Library Function Code pikalajittelun lähdekoodit **64**

B.1 etsipivot.h osoitetiedosto: 64

B.2 partion.h osoitetiedosto: 64

B.3 etsipivot.c lähdekoodi: 64

B.4 partition.c lähdekoodi: 65

B.5 quicksort.c lähdekoodi: 65

C CIN pikalajittelun lähdekoodi **67**

D Pinoperustaisen pikalajittelun LabVIEW koodi **69**

E Korjattu pinoperustaisen pikalajittelun LabVIEW koodi **70**

1 JOHDANTO

LabVIEW on National Instruments yhtiön kehittämä visuaalinen ohjelmointiympäristö. Se käyttää kielenään graafista G-kieltä. Tässä tutkielmassa visuaalisella ohjelmoinnilla ei tarkoita kieliä, joiden ohjelmointikehitysvälineisiin on lisätty rutiineja helpottavia visuaalisia ominaisuuksia. Visual Basicilla ohjelmointi ei ole visuaalista ohjelmointia, koska toiminnallisuus graafisiin elementteihin on kirjoitettava tekstuaalisesti. Puhtaan visuaalisen ohjelmintikielen graafisiin elementteihin sisältyy sekä visuaalinen esitys elementistä että sen toiminnallisuus. Pääsääntöisesti mitään ohjelmointikoodin kirjoittamista ei siis tarvita.

Visuaalisessa ohjelmoinnissa lähdekoodi muodostuu visuaalisista elementeistä ja näiden välille luoduista yhteyksistä. LabVIEW-ympäristössä näitä elementtejä ovat kuvakkeet ja niiden väliset yhteydet hoidetaan lagoittamalla. Syntyy siis graafi, jossa kuvakkeet ovat solmuja ja langoitukset solmujen välisiä kaaria.

LabVIEW-ohjelmointiympäristö toteuttaa hyvin pitkälle tietovuota ohjelmointiparadigmana. LabVIEW-ohjelmissa tieto virtaa lagoituksia pitkin kuvakkeesta toiseen. Kuvakkeen toiminta suoritetaan vasta kun se on saanut kaiken siihen liitetyn tiedon ja tieto lähtee kuvakkeesta vasta kun sen kaikki osat on suoritettu. Tieto siis ohjaa ohjelman suoritusta.

Rekursio on yksi algoritmisen ongelmanratkaisun mallesta silloin, kun samaa operaatiosarjaa käytetään uudelleen. Toistoperiaatteesta poiketen funktio kutsuu rekursiossa itseää. Voitaisiin sanoa että funktio on itsensä aliohjelmarutiini. Vaikka LabVIEW-ohjelmointiympäristö on modulaarinen ja sallii aliohjelmarutiinit on rekursion toteuttaminen siinä ongelmallista. LabVIEW ei nimittäin salli lagoituksia itseensä, ei edes epäsuorasti. Kuvake ei siten myöskään voi kutsua itseään. Tässä tutkielmassa pyritään etsimään ratkaisumalleja siihen, miten rekursiivisluonteiset ratkaisut olisivat mahdollisia.

Kappaleessa 2 käydään lyhyesti läpi visuaalisen ohjelmoinnin historiaa ja kerrotaan minkälaisen visuaalisen kielen on oltava. Kappaleen loppussa esitellään LabVIEW-ohjelmointiympäristö pääpiirteissään.

Kappaleessa 3 käydään läpi rekursion ja iteraation perusperiaatteet. Sen jälkeen perehdytään rekursion teoriaan ja lopuksi siihen miten LabVIEW-ympäristö suh-

tautuu siihen.

Kappale 4 keskittyy esittelenään LabVIEW-ympäristössä toteutettuja rekursiivisia pikalajittelu ratkaisuja ja sitä miten ne on rakennettu. Kappaleen lopussa esitellään myös tutkielman aikana tehty pienimuotoinen nopeustesti eri ratkaisujen välillä.

Viimeisessä kappaleessa 5 tämän tutkielman tekijä tuo esille muutamia omia ajatuksia ja mietelmiä, joita hänellä on syntynyt tätä tutkielmaa tehdessä.

Liiteosassa on rekursiivisten pikalajittelu ratkaisuiden lähdekoodit. Lisäksi tämän tutkielman mukana toimitetaan CD-ROM levyke, jossa on valtaosa tässä tutkielmassa esitetyistä LabVIEW esimerkeistä lähdekoodeineen. Kaikki tässä tutkielmassa esitetyt LabVIEW esimerkit on ohjelmoitu kyseisen ohjelmointiympäristön versiolla 7.1.

2 VISUAALINEN OHJELMOINTI JA LabVIEW

LabVIEW on National Instrumentin kehittämä visuaalinen tietovuo-ohjelmointiympäristö, joka käyttää ohjelmointikielenään graafista G-kieltä. Margaret M. Burnettin [Bur99] mukaan ohjelmointikieli on visuaalinen silloin, kun sen syntaksi sisältää visuaalisia ilmaisuja *(visual expression)*. Lausekkeen notaatio on ilmaistu kaksi- tai useampiulotteisina objekteina. Tietovuo *(data flow)* on ohjelmointiparadigma, jota käytetään varsinkin ohjelmointikielissä, jotka on alun perin suunniteltu teollisuuden tarpeisiin. Tässä luvussa esitellään tutkielman kannalta olennaisia piirteitä edellä mainituista seikoista ja käsitellään sitä, miten ne vaikuttavat LabVIEW-ohjelmointiin. Tämän luvun tärkeimmät lähteet visuaalisen ohjelmoinnin osalta ovat Margaret M. Burnettin *Visual Programming* [Bur99] ja Marat Boshernitsan ja Michael Downesin *Visual Programming Languages: A Survey* [BD04], LabVIEW ohjelmoinnin osalta Robert H. Bishopin *Learning with LabVIEW 7 Express* [Bis04] sekä National Instrumentsin *LabVIEW 7 Express, User Manual* [Nat03a].

2.1 Historiaa

Visuaalinen ohjelmointi on kehittynyt yhdessä tietokonegrafiikan ja tietokoneiden käytettävyyden suunnittelun kanssa. Muun kuin pelkän näppäimistön käyttö tietokoneen tiedon syöttövälineenä mahdollisti myös ohjelmoinnin graafisella tavalla. Ivan Sutherlandin *Sketchpad* vuodelta 1963 on ensimmäinen graafinen tietokoneohjelma. Siinä valokynän avulla pystyttiin piirtämään kaksiulotteista grafiikkaa luomalla yksinkertaisia muotoja kuten viivoja ja ympyröitä, joita voitiin kopioida ja siten tuotettiin monimutkaisempia geometrisiä kuvioita. Ohjelmassa oli graafinen käyttöliittymä ja se tuki käyttäjän tekemiä määrityksiä. *Sketchpadia* pidetään nykyään visuaalisen ohjelmoinnin tärkeimpänä läpimurtona. Vuonna 1965 Sutherlandin veli William kehitti yksinkertaisen visuaalisen tietovuo-ohjelmointikielen, jolla pystyttiin luomaan, korjaamaan ja ajamaan tietovuokaavioita *(data flow diagrams)*.

Varsinaisena lähtökohtana laajamittaiseen visuaalisen ohjelmoinnin tutkimukseen ja kehitämiseen pidetään kuitenkin David Canfield Smithin *Pygmalionia* vuodelta 1975. Hän esitteli ensimmäistä kertaa kuvakepohjaisen ohjelmointiparadigman, jossa käyttäjä loi, muokkasi ja linkitti yhteen pieniä kuvaobjekteja. Objektit sisälsivät

sen toiminnallisuuden, jota ohjelman suorituksessa tarvittiin. Hyvin monet nykyisistä visuaalisista ohjelmointikielistä pohjautuvat tähän kuvakepohjaiseen lähestymistapaan, niin myös LabVIEW.

Visuaalisen ohjelmoinnin historiaa voidaan lähestyä myös toisenlaisesta näkökulmasta. Visuaalisen ohjelmoinnin voidaan ajatella kehittyneen kahta tietä. Ensinnäkin visuaalisuutta on lähdetty kehittänään perinteisten ohjelmointikielten näkökulmasta. Tekstuaalisiin kieliin on lisätty visuaalisia elementtejä helpottamaan ohjelmointia. Tavoitteena on ohjelmissa usein tarvittavien, toistuvien osien ja rutiinien uudelleenkirjoittamisen vähentäminen. Näitä osia ovat esimerkiksi käyttöliittymissä tarvittavat painonappulat ja vierityspalkit.

Toinen lähtökohta on ollut unohtaa tekstuaalisten kielten painolasti ja ottaa lähtökohdaksi kokonaan uusi visuaalinen näkökulma. Edellä mainittu *Pygmalion* on tästä hyvä esimerkki. Vaikka aiemmissa visuaalisissa ohjelmissa oli hyötyjä, jotka näyttivät jännittäviltä ja intuitiivisilta, niitä pidettiin lähinnä "leluohjelmina", joilla ei pystytä ratkaisemaan todellisia vaikeita reaalimaailman ongelmia. Niiden katsottiin olevan lähinnä akateemisia harjoituksia. Osaksi tästä syystä visuaalisten ohjelmointikielten tutkimus ja kehitys suuntautuikin niihin ongelmiin ja ohjelmistotuotannon osa-alueisiin, mihin sen katsottiin sopivan parhaiten. Vaikka monet nykyiset visuaaliset ohjelmointikielet sopisivat yleiseenkin käyttöön, ei yksikään niistä ole vielä pystynyt saavuttamaan tällaistä asemaa. Ne ovat kuitenkin hyvin tunnettuja ja käytettyjä niillä osa-alueilla, mihin ne alunperin kehitettiin.

2.2 Visuaalinen kieli

Puhtaista visuaalisista ohjelmointikielistä *(pure visual languages)* on pidettävä erillään visuaaliset ohjelmointiympäristöt *(visual programming environments)*, joita monet tekstuaalisten kielten kehittäjät ovat rakentaneet ohjelmistojensa ympärille. Delphi ja Visual Basic ovat esimerkkejä ohjelmointikielistä, jotka eivät ole visuaalisia ohjelmointikieliä graafisista ominaisuuksistaan huolimatta. Vaikka näillä kielillä pystytään toteuttamaan käyttöliittymät visuaalisesti, eivät graafiset elementit sisällä mitään toiminnallisuutta, vaan ne on kirjoitettava tekstuaalisesti kyseisen kielen syntaksia noudattaen.

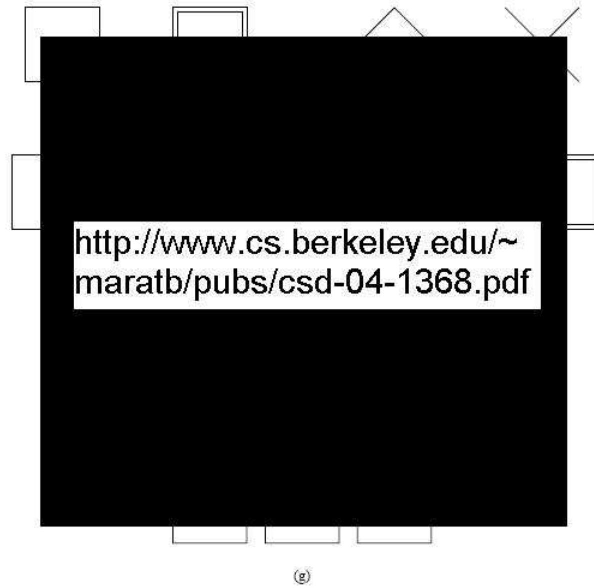
Puhtaissa visuaalisissa kielissä käytetään visuaalista tekniikkaa koko ohjelmointiprosessin ajan. Ohjelmoija käyttää hyväkseen kuvakkeita ja muita kielen graafisia elementtejä luodakseen ohjelman, joka jäsennetään, käännetään ja suoritetaan samassa visuaalisessa ympäristössä, missä se on kehitetty. Ohjelma siis käännetään suoraan sen visuaalisesta esityksestä ilman, että sitä ensin muunnetaan johonkin tekstuaaliseen ohjelmointikieleen.

Puhtaat visuaaliset kielet on rakennettu seuraavan teorian mukaan [BD04]:

- Kuvake *(icon)* on objekti, jolla on kaksitahoinen esitys. Sen looginen osa sisältää toiminnan tai tarkoituksen ja fyysinen osa itse kuvan esityksen.
- Kuvakejärjestelmä *(iconic system)* on joukko jäsennettyjä yhteen kuuluvia kuvakkeita.
- Visuaalinen lause *(visual sentence)* on kuvakejärjestelmän kuvakkeita, jotka on järjestetty sovitulla tavalla. Vastaa tekstuaalisen ohjelmointikielen lausetta.
- Visuaalinen kieli on joukko visuaalisia lauseita, joille on annettu tietty syntaksi ja semantiikka.

Lisäksi tarvitaan syntaksinen ja semanttinen analyysi, joka tuottaa esityksen rakenteen ja toiminnan.

Kuvassa 1 on osa Heidelbergin kuvakejoukosta *(icon set)* ja kuvakkeilla ilmaistu visuaalinen lause. Alkeiskuvakkeet *(elementary object icons)* kuvaavat kielen primitiivejä, jossa (a) on kirjain ja (b) on valittu kirjain. Prossessikuvakkeet *(process icons)* ovat myös alkeiskuvakkeita ja siten kielen primitiivejä, jotka kuvaavat toimintaa. Kuvakkeista (c) on lisäysoperaatio ja (d) on poisto-operaatio. Yhdistelmäkuvakkeet *(composite object icons)* ovat kielen syntaksin ja semantiikan mukaisesti järjestetty joukko alkeiskuvakkeita. Yhdistelmäkuvake (e) on merkkijono ja (f) on valikoitu merkkijono. Visuaalinen lause (g) ilmaisee merkkijonon kirjaimen vaihtamisen merkkijonoon.



Kuva 1: Heidelbergin kuvakejoukko.[BD04]

Visuaalinen ohjelmointikieli on määriteltävissä myös kolmikolla *(triple)* [BD04]:

$$(ID, G_0, B) \tag{1}$$

missä ID on kuvakehakemisto *(icon dictionary)*, G_0 on kielioppi *(grammar)* ja B sisältää tapauskohtaisen tietämyskannan *(knowledge base)*. Kuvakehakemisto on peruskuvakkeiden joukko, joita voidaan kuvata parilla (X_m, X_i), missä looginen osa X_m edustaa tarkoitusta tai toimintaa ja fyysinen osa X_i itse visuaalista esitystä. Kielioppi G_0 määrittää ne säännöt, joilla yhdistelmäkuvakkeita voidaan luoda peruskuvakkeista käyttämällä kielen prosessioperaatioita. Kieliopissa täytyy prosessioperaatiot myös määritellä tarkasti terminaaleina, koska ne eivät enää ole epäsuoria yleistyksiä kielen määrittelyssä. Esimerkiksi lisäys-operaation on saatava säännöt siitä, minkä tyyppistä dataa sen avulla voidaan lisätä. Tietämyskanta B sisältää kohdealueyksilöidyn tiedon siitä, miten luodaan tarkoitus tai toiminta annetulle visuaaliselle lausekkeelle. Se sisältää siis tiedon mm. tapahtumien *(event)* nimistä, käsiteyhteyksistä *(conceptual relations)* ja tulosobjektien *(resulting objects)* nimistä sekä niiden viittauksista.

Yleinen harhaluulo visuaalisten ohjelmointikielten kehittämisestä ja tutkimisesta

on se, että niiden tärkein tarkoitusperä olisi täydellinen tekstuaalisuudesta irtautuminen. Margaret M. Burnett pitää kuitenkin tärkeimpänä taivoitteena parantaa ohjelmointikielten suunnittelua. Muita tärkeitä tavoitteita ovat ohjelmoinnin tekeminen helpommin lähestyttäväksi suuremmalle joukolle ihmisiä, parantaa ohjelmien virheettömyyttä sekä nopeuttaa ohjelmointitehtävistä suoriutumista. Näihin tavoitteisiin pyritään alla olevien neljän strategian kautta [Bur99].

- Konkreettisuus *(concreteness)* on abstraktisuuden välttämistä. Esimerkiksi systeemi automaattisesti näyttää miten objektin tai arvon lisääminen vaikuttaa ohjelmaan ja mihin kohtaan sitä. Tekstuaalisessa ohjelmoinnissa tämän selville saamiseen tarvitaan vähintään ohjelman kääntäminen ja jos syntaktista virhettä ei ole, vaaditaan vielä ohjelman suoritus, jotta semanttinen merkitys tosiasiassa voi paljastua. Konkreettisuus antaa ohjelmoijalle mahdollisuuden puuttua jo aikaisessa vaiheessa semanttisiin virheisiin.
- Suoruus *(directness)* on suoraa vaikuttamista asiayhteyteen. Kongnitiivisessa mielessä kyse on päämäärän ja sen saavuttamiseksi tarvittavien toimenpiteiden välinen mahdollisimman pieni etäisyys. Ohjelmoija voi esimerkiksi muuttaa objektin tai arvon semantiikkaa suoraan ilman, että se ensin täytyy ilmaistaan tekstuaalisesti.
- Ymmärrettävyys *(explicitness)* on ohjelmistoympäristön semanttisten aspektien käsittäminen, kun ne on suoraan ilmaistu, ilman että ohjelmoijan täytyy tehdä niistä omia päätelmiään tai tulkintoja. Visuaalinen ohjelmointiympäristö voi esimerkiksi luoda automaattisesti kuvakkeiden väliset yhteydet piirtämällä tarvittavat relaatiot niiden välille.
- Välitön visuaalinen palaute *(immediate visual feedback)* on ohjemointiympäristön antamaa välitöntä palautetta siitä vaikutuksesta, mitä ohjelmoinnin edetessä mistäkin toimenpiteestä seuraa.

2.3 LabVIEW

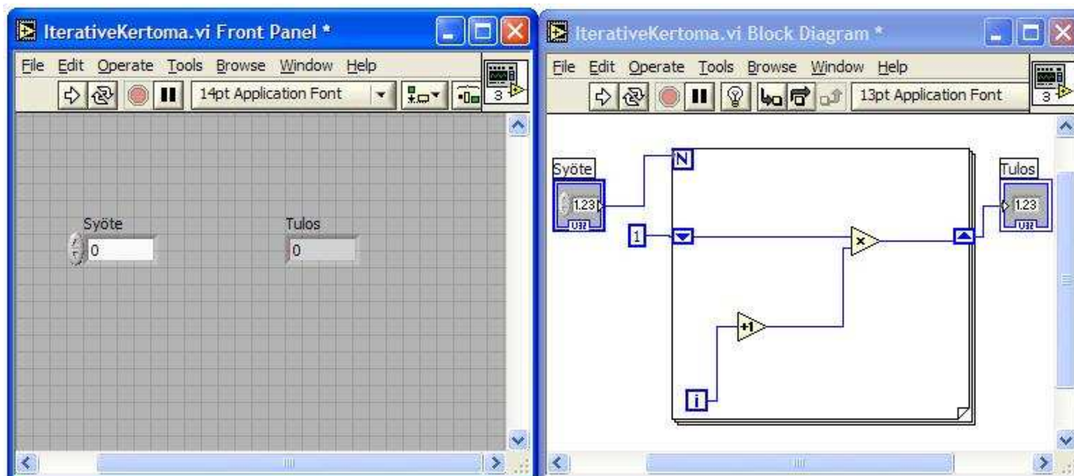
LabVIEW on National Instrumentin kehittämä visuaalinen tietovuo-ohjelmointiympäristö, joka käyttää ohjelmointikielenään graafista G-kieltä. Se on kehitetty lähinnä

tieteellisiin ja teknisiin tarpeisiin, mittaamaan ja analysoimaan erilaisista mittalaitteista tulevia signaaleja sekä näyttämään näiden mittausten tulokset. LabVIEW-ohjelmia kutsutaan virtuaali-instrumenteiksi *(virtual instruments)* ja se lyhennetään usein VI:ksi.

Ohjelmointi tapahtuu graafisen G-kielen peruskuvakkeita, ohjelmointiympäristössä olevia valmiita funktiota ja syöttö- ja tuloskuvakkeita käyttäen. Peruskuvakkeet ilmaisevat jonkin perusoperaation, kuten esimerkiksi yhteen- tai vähennyslaskun. Syöttö- tai tulostietojen kuvakkeet ovat tiedonvarastointipaikkoja, olematta kuitenkaan muuttujia traditionaalisen ohjelmoinnin näkökulmasta. Valmiit funktiot ovat ympäristöön tehtyjä aliohjelmia ja LabVIEW:n näkökulmasta ne ovat myös virtuaali-instrumentteja. Kuvakkeiden väliset yhteydet langoitetaan *(wires)* ja näin tieto virtaa tarvittavista operaatioista toiseen.

Kuvassa 2 on esitetty yksinkertainen iteratiivinen kertoma-ohjelma. Vasemmalla on ohjelman käyttöliittymä ja oikealla sen visuaalinen koodi. Ohjelma sisältää yhden syöttökontrollin *(Syöte)* ja yhden tulosindikaattorin *(Tulos)*. Kontrollin ja indikaattorin välissä on toistosilmukka, jonka vasemmassa ja oikeassa laidassa on alas- ja ylöspäin olevat nuolet. Ne esittävät siirtorekisteriä *(shift register)*, joka välittää arvon toistokierrokselta toiselle. Lisäksi silmukan sisässä on primitiivikuvakkeet kertolaskua ja yhdellä lisäämistä varten. Toistosilmukan ulkopuolella vasemmalla on vakiota ilmaiseva kuvake. Tässä yhteydessä ei ole tarpeen selittää kertoman laskua vaan kuvata yksinkertaisesti, kuinka LabVIEW-ohjelma toimii. Syöttökontrolliin annetaan luku, jonka kertoma halutaan laskea. Tämä luku on langoitettu toistosilmukkaan ja se kertoo, kuinka monta kertaa silmukka käydään läpi. Vakio (1) syötetään silmukan kertolaskuoperaatiolle toiseksi parametriksi ja toisen parametrin se saa silmukan informaatiopisteestä, joka antaa tiedon siitä, kuika monta kertaa silmukka on suoritettu. Informaatiopisteestä saatavaa lukua kasvatetetaan yhdellä ennen kertolaskuoperaatiota. Jokaisella silmukan läpikäyntikerralla kertolaskuoperaattori saa parametreikseen edellisen kertolaskun tuloksen ja informaatiopisteestä saadun luvun lisättynä yhdellä. Kun toistosilmukka pysähtyy, annetaan viimeisen käyntikerran kertolaskun tulos tulosindikaattoriin, joka näyttää laskennan tuloksen käyttöliittymässä.

LabVIEW-ohjelmat rakennetaan hierarkisiksi ja niissä noudatetaan modulaarista ohjelmointia. Tehtävä jaetaan yksinkertaisempiin osatehtäviin ja jokainen osateh-



Kuva 2: Iteratiivinen kertoma.

tävä on myös virtuaali-instrumentti *(subVI)*. Analogisesti nämä osatehtävät ovat aliohjelmia tekstuaalisissa ohjelmointikielissä. Jokainen virtuaali-instrumentti koostuu kolmesta pääkomponentista: etupaneeli *(front panel)*, kaaviodiagrammi *(block diagram)* ja kuvake liitinpareineen *(connector pair)*. Etupaneeli sisältää syöttö- ja tuloskuvakkeet ja on pääohjelmassa ohjelman käyttöliittymä. Aliohjelmassa etupaneelia voidaan pitää rajapintana, jonka avulla siihen päästään käsiksi.

Kaaviodiagrammi on ohjelman kuvallinen esitys sen ohjelmointikoodista. Se koostuu suoritettavista kuvakkeista *(executable icon)*, joita LabVIEW:ssa kutsutaan solmuiksi *(nodes)* ja näitä yhdistävistä langoituksista. Solmut ovat analogisesti käskylauseita, funktioita, aliohjelmia ja tietorakenteita tekstuaalisen ohjelmoinnin näkökulmasta. Kuvakkeiden välille luotuja yhteyksiä eli langoituksia taas voidaan pitää muuttujina traditionaalisen ohjelmoinnin kannalta. Kaaviodiagrammissa on myös syöttö- ja tuloskuvakkeita, joiden vastin on etupaneelin syöttö- ja tuloskuvakkeet. Kun etupaneeliin luodaan esimerkiksi painonappi, LabVIEW automaattisesti luo kyseisen objektin myös kaaviodiagrammiin. Kyseisten kontrollien avulla luodaan yhteys etupaneelin ja kaaviodigrammin välille. Syöttö- ja tuloskuvakkeet ovat analogisessa mielessä parametrejä ja vakiota ja jokainen langoitettava kohta on terminaali, jos sen kautta voi kuljettaa tietoa.

Kolmas LabVIEW-ohjelman pääkomponentti on itse kuvake ja siinä oleva liitinpari. Jotta ohjelmakoodia voitaisiin käyttää muissa ohjelmissa on sille luotava kuvallinen esitys. Kuvakkeen on sisällettävä myös vähintään yksi liitinpari, jonka avulla

ohjelmakoodi voidaan langoittaa osaksi suurempaa kokonaisuutta.

LabVIEW-ohjelmia kutsutaan virtuaali-instrumenteiksi, koska ne näyttävät ja ovat olemukseltaan kuin fyysisiä mittareita tai laitteita. Ohjelmia suunnitellessa käytetään hyväksi erilaisia kaavioita kuvaamaan tulevaa ohjelmaa ja helpottamaan sen toteutusta. UML-standardin mukaiset suunnittelumenetelmät, kuten luokka- ja prosessikaaviot ovat käytössä laajalti. Eräs usein käytetty suunnittelumalli on tietovirta- eli tietovuokaavio, vaikka se ei kuulukkaan UML-standardiin. Sen avulla voidaan kuvata tiedon kulku, tiedon varastointi ja käsittely järjestelmässä [HM04]. Kun kyseinen malli otetaan käyttöön ohjelmointiparadigmana, puhutaan tietovuo-ohjelmoinnista *(data flow programming)*. Useimmissa tekstuaalisissa ohjelmointikielissä koodi käydään läpi rivi riviltä ja ohjelman suoritusjärjestys on helposti nähtävissä lähdekoodista. LabVIEW:n tietovuomallissa ohjelman suoritusjärjestystä ei välttämättä voida lukea suoraan ohjelmakoodista. Yksittäinen virtuaali-instrumentti suoritetaan, kun se on vastaanottanut kaikki sen tarvitsemat syöttötiedot. Tieto virtaa suoritettavasta elementistä ulos vasta, kun sen kaikki osat on suoritettu.

Tietovuomalli tukee mahdollisuutta rinnakkaisuuteen ja paradigma on läheisessä suhteessa funktionaaliseen ohjelmointiin [Aug97], sillä molemmat kuuluvat deklaratiivisiin ohjelmointikieliin [Sco00]. Deklaratiiviset kielet määrittelevät sen, mitä ollaan laskemassa, kun taas imperatiiviset kielet määrittelevät sen, kuinka laskenta suoritetaan. Funktionaaliset kielet esittelevät laskentamallin, joka perustuu funktioiden rekursiivisiin ominaisuuksiin. Tietovuoparadigma itsessään ei sulje rekursiota pois, mutta LabVIEW-ympäristössä rekursion käytömahdollisuudet ovat rajatut, koska virtuaali-instrumentit muodostavat suunnatun syklittömän graafin eikä takaisinkytkentä ole mahdollista. Virtuaali-instrumenttia tai sen solmua ei voi siis langoittaa itseensä ja siten funktion itsensä kutsu on poissuljettu. Tässä tutkielmassa asiaa käsitellään tarkemmin rekursiota kuvaavassa luvussa 3.

3 REKURSIO

Ohjelmointikielissä on kaksi tapaa toteuttaa ongelmanratkaisu tapauksissa, missä samaa operaatiosarjaa käytetään uudelleen. Imperatiivisissa kielissä käytetään toteutuksissa enimmäkseen iteraatiota eli toistosilmukoita, kun taas funktionaaliset kielet suosivat rekursiota. Rekursiossa ratkaisu saadaan aikaan funktion kutsuessa itseään. Yleensä rekursio on mahdollista kaikilla niillä ohjelmointikielillä, jotka tarjoavat käyttöön aliohjelmarutiinit. Tavat millä toisto toteutetaan, eivät sulje toisiaan pois. Yleensä iteratiivinen algoritmi voidaan uudelleenkirjoittaa rekursiivisesti ja päinvastoin. Tässä kappaleessa käsitellään ensin rekursiota ja iteraatiota yleisesti sekä rekursiota teorettisesta näkökulmasta. Lopuksi tarkastellaan rekursiota LabVIEW:n näkökulmasta. Kappaleen päälähteinä ovat rekursion osalta Michael L Scottin *Programming Language Pragmatics* [Sco00] ja Turun yliopiston täydennyskoulutuskeskuksen painama opintomoniste *Johdatus tietojenkäsittelytieteeseen* [BLP+00] sekä Cristopher Mooren *Recursion theory on the reals and continuous-time computation* [Moo95]. LabVIEW:n osalta Robert H Bishopin *Learning with LabVIEW 7 Express* [Bis04] ja National Instrumentsin *LabVIEW 7 Express, User Manual* [Nat03a].

3.1 Iteraatio ja rekursio

Iteraatio ja rekursio ovat algoritmisen ongelmanratkaisun perusrakenteita. Jokaisella suorituskerralla pyritään lähemmäksi ongelmanratkaisua oli kyse sitten kummasta ratkaisuperiaatteesta tahansa. Iteraatioperiaate toteutetaan useimmisssa ohjelmointikielissä toistosilmukoiden avulla. Jotta toisto joskus päättyisi, on silmukan rakenteessa oltava ehto, jonka toteutuessa suoritus päättyy. Tämän ehdon perusteella toistosilmukat voidaan jakaa kahteen perustapaukseen. Kun käytössä on mekanismi, joka antaa silmukalle sen suorituskertojen määrän, puhutaan laskentakontrolloidusta silmukasta *(enumeration-controlled loop)*. Esimerkiksi *for*-silmukka kuuluu tähän joukkoon. Koska suorituskertojen määrä on annettu, päättyy laskentakontrolloidun silmukan suoritus aina [Moo95]. Eri asia on se, onko suorituskertoja oikea määrä. Kun toiston lopetusehto on jokin totuusarvo, jonka täyttyessä iteraatio lopetetaan, puhutaan loogisesti kontrolloidusta silmukasta *(logically controlled loop)*. Esimerkiksi *while*-silmukka toteuttaa edellämainitun. Loogisesti kontrolloitu silmukka ei välttämättä pääty koskaan. Sen päättymisehtoa ei ole välttämättä määritelty

millään esiintulevilla tapauksilla ja siten suoritus voi jäädä niin sanottuun ikuiseen silmukkaan [Moo95]. Jälkimmäinen toistorakenne voidaan vielä jakaa esiehtoiseen *(precondition)* silmukkaan, jolloin ehto tarkistetaan joka kerta ennen toiston suorittamista ja jälkiehtoiseen *(postcondition)* silmukkaan, jolloin silmukka suoritetaan ensin ja lopetusehto tarkistetaan vasta sen jälkeen.

Iteraatiota käytetään yleensä sen antaman tärkeän sivuvaikutuksen vuoksi. Jokainen iteraatiokierros vaikuttaa paikallisiin muuttujiin *(local variables)*. Rekursiossa paikalliset muuttujat pysyvät koskemattomina. Jokaisella rekursiokierroksella luodaan paikallisista muuttujista uudet ilmentymät ja aikaisempien rekursiokutsujen muuttujat pysyvät muistissa. Tällä erolla muuttujien käsittelyssä voi olla vaikutusta koko ohjelman suoritukseen. Vaikka iteratiivinen ja rekursiivinen ratkaisu olisivat muuten yhtä tehokkaita, rekursion vaatima muistin määrä voi joissain tapauksissa olla liian suuri.

Esimerkiksi kertoman lasku määritellään usein rekursiivisen palautuskaavan avulla [Als99]:

$$0! = 1, \quad n! = n(n-1)! \quad jos\ n \geq 1 \tag{2}$$

Taulukoissa 1 ja 2 on lause 2 kirjoitettuna pseudokieliseksi rekursiiviseksi ja iteratiiviseksi ohjelmaksi. Nänä kaksi esimerkkiä osoittavat, että jos ratkaisu on luonteeltaan rekursiivinen, on sen rekursiivinen algoritmikin ymmärrettävämpi ja selkeämpi kuin iteratiivinen vastineensa. Tässä tapauksessa ei suoritustehokkuudessa algoritmien välillä ole juurikaan eroa.

```
MODULE kertoma(n) RETURNS n!
     IF n=0 THEN
          RETURN 1
     ELSE
          RETURN n*kertoma(n-1)
     ENDIF
ENDMODULE
```

Taulukko 1: Rekursiivinen kertoma [BLP+00].

```
MODULE kertoma(n) RETURNS n!
     k:=1
     WHILE n>1 DO
          k:=k*n
          n:=n-1
     ENDWHILE
     RETURN k
ENDMODULE
```

Taulukko 2: Iteratiivinen kertoma [BLP+00].

Taulukossa 3 käydään rekursiivinen kertoma läpi askel askeleelta. Siitä voidaan huomata, että jokaisen rekursiivisen kutsun suoritus päättyy vasta, kun kaikki sen aikaansaamat kutsut ovat päättyneet. Esimerkiksi kertoma(3):n lasku ainoastaan keskeytyy kertoma(2):n suorituksen ajaksi. Kun kertoma(2) on laskettu, jatketaan kertoma(3):n kesken jäänyttä suoritusta kertolaskulla 3*2=6 ja vasta sitten kutsu saa arvonsa.

```
kertoma(3):
IF 3=0 THEN
...
ELSE RETURN 3*kertoma(2)
  kertoma(2):
  IF 2=0 THEN
  ...
  ELSE RETURN 2*kertoma(1)
    kertoma(1):
    IF 1=0 THEN
    ...
    ELSE RETURN 1*kertoma(0)
      kertoma(0):
      IF 0=0 THEN RETURN 1  (0! valmis)
    RETURN 1*1=1          (1! valmis)
  RETURN 2*1=2          (2! valmis)
RETURN 3*2=6          (3! valmis)
```

Taulukko 3: Rekursiivisen kertoman läpikäynti [BLP+00].

Iteraatio ja rekursio ovat yleensä vaihtoehtoisia ongelmanratkaisutapoja. Se, kumpaa tapaa käytetään, riippuu usein itse algoritmin luonteesta. On olemassa luonteeltaan iteratiivisia algoritmeja, joille rekursiivisen ratkaisun löytäminen voi olla vaikeaa tai jopa mahdotonta. Koska tietokone itsessään toimii iteratiivisesti, on tä-

mä ratkaisumalli usein myös hyvin luonnollinen valinta. Jotkut ohjelmointiympäristöt jopa muuttavat rekursiiviset ratkaisut iteratiivisiksi ennen niiden kääntämistä. Tämän lisäksi on myös luonteeltaan rekursiivisia algoritmeja, joiden toteuttaminen iteratiivisesti voi olla vaikeaa. Rekursiivinen ratkaisu tuottaa usein lyhyemmän ja selkeämmän ratkaisumallin, kuin iteraatiolla olisi mahdollista toteuttaa [AHU74].

3.2 Rekursion teoriaa

Perinteinen rekursioteoria määrittelee joukon laskettavissa olevia tai rekursiivisia funktioita luonnollisille luvuille N. Ne voidaan luoda joukolla alkeisfuktioita, joilla on tiettyjä ymmärrettäviä sääntöjä. Nämä alkeisfunktiot ovat [Moo95]:

$$Z(n) = 0 \quad \text{(nollafunktio)} \tag{3}$$

$$S(n) = x + 1 \quad \text{(seuraajafunktio)} \tag{4}$$

$$P_i^n(x_1, \ldots, x_n) = x_i \quad (1 \leq i \leq n) \quad \text{(projektiofunktio)} \tag{5}$$

Projektiofunktio koostuu vektorista $\vec{x}$ komponentteineen. Alkeisfunktioita voidaan käyttää kolmen operaattorin avulla. Määrittelemällä alla olevat operaattorit termeillä f ja g saamme aikaan funktion h [Moo95]:

- Yhdistäminen *(composition)*: $h(\vec{x}) = f(g(\vec{x}))$.
- Primitiivirekursio *(primitive recursion)*:
  $h(\vec{x}, 0) = f(\vec{x}), h(\vec{x}, y + 1) = g(\vec{x}, y, h(\vec{x}, y))$.
- μ-Rekursio *(μ-recursion)*: $h(\vec{x}) = \mu_y f(\vec{x}, y) = \min\{y | f(\vec{x}, y) = 0\}$.

Funktiot, jotka voidaan generoida alkeisfunktioista Z, S ja P käyttämällä vain yhdistämistä ja primitiivirekursiota kutsutaan primitiivirekursiivisiksi *(primitive recursive)*. Useimmat yksinkertaiset aritmeettiset funktiot, kuten kertolasku ja eksponentti, ovat primitiivirekursiivisia. Iteratiivisesti ajatellen ne toteuttavat *for*-silmukan, jonka suoritus päättyy aina. μ-rekursion iteratiivinen vastine on *while*-silmukka. Siinä etsitään mielivaltaisen laajan y:n arvosta pienintä luonnollista lukua, jotka lopulta

toteuttaa $f(\vec{x}, y) = 0$. Tätä arvoa ei välttämättä koskaan löydy, joten rekursio ei silloin koskaan pääty. Jos funktio voidaan generoida kaikilla kolmella operaattorilla, mutta funktiota ei pystytä määrittelemään kaikilla vektorin $\vec{x}$ arvoilla on se osittainen funktio ja siten myös osittain rekursiivinen *(partial recursive)*. Jos funktio voidaan generoida kaikilla kolmella operaattorilla ja se voidaan määrittää kaikilla vektorin $\vec{x}$ arvoilla on se täydellinen funktio ja se on silloin rekursiivinen *(recursive)*. [Moo95]

Yksinkertaisempi tapa ilmentää rekursiota on rekursiivinen ohjelma P. Kuvataan se koosteena P, lauseiden joukkona S, joka ei sisälly koosteeseen P ja P:n itsensäviittauksena [Wir86]:

$$P \equiv P[S, P] \tag{6}$$

Jotta ohjelma voidaan ilmaista rekursiivisesti on siinä oltava proseduuri tai aliohjelmarutiini, joka pystytään nimeämään. Ohjelmalauseen on myös pystyttävä kutsumaan proseduuria tällä tietyllä nimellä. Kuten lauseesta 6 käy ilmi, proseduuri P sisältää suoran viittauksen itseensä. Rekursion sanotaan tällöin olevan suora *(directly recursive)*. Jos proseduuri P sisältää viittauksen toiseen proseduuriin Q ja tämä viittaa suorasti tai epäsuorasti proseduuriin P, sanotaan rekursiota epäsuoraksi *(inderectly recursive)*. Esimerkki tästä on lauseessa 7. Epäsuoran rekursion käyttäminen ohjelmakoodissa voi tämän vuoksi olla vaikeasti havaittavissa. Epäsuoraa rekursiota pitäisi mahdollisuuksien mukaan välttää, jotta ohjelmakoodi olisi luettavampaa ja helpommin ymmärrettävää.

$$\begin{aligned} P &\equiv P[S, Q] \\ Q &\equiv Q[S, P] \end{aligned} \tag{7}$$

Samoin kuin iteratiivinen toisto on myös rekursio saatava päättymään. Rekursiivisissa funktioissa ja proseduureissa on huomioitava kaksi asiaa. Ensiksi pitää kuvata ainakin yksi ongelman alkeistapaus, joka ratkeaa suoraan, ei-rekursiivisesti. Toiseksi, jokaisen rekursiivisen kutsun on lähestyttävä jotakin alkeistapausta. Edellämainitussa lauseessa 6 ei rekursion päättymistä ole otettu huomioon. Lisäämällä proseduuriin

P ehto B, joka saa jossain vaiheessa totuusarvon epätosi, voi rekursion päättyminen tapahtua. Rekursiivinen algoritmi voidaan esittää joko lauseen 8 tai lauseen 9 muodossa [Wir86]:

$$P \equiv \text{IF B THEN} \quad P[S, P] \quad \text{END} \tag{8}$$

$$P \equiv P[S, \quad \text{IF B THEN P END}] \tag{9}$$

Määrittelemällä funktio *f(x)*, jossa x on muuttujajoukko, voidaan rekursion päättyminen todistaa. Osoitetaan, että jokaisella proseduurin P kutsukerralla vähenee funktion *f(x)* muuttujajoukko ja lähennytään tilannetta, jossa *f(x)* $\leq$ *0*. Kun tämä on tosi, se samalla toteuttaa ehdon B totuusarvon muuttoksen epätodeksi. Päättyminen voidaan varmistaa ottamalla proseduuriin P parametri n. Lauseissa 10 ja 11 on tästä esimerkit [Wir86]:

$$P(n) \equiv \text{IF n>0 THEN P[S,P(n-1)] END} \tag{10}$$

$$P(n) \equiv \text{P[S,IF n>0 THEN P(n-1) END]} \tag{11}$$

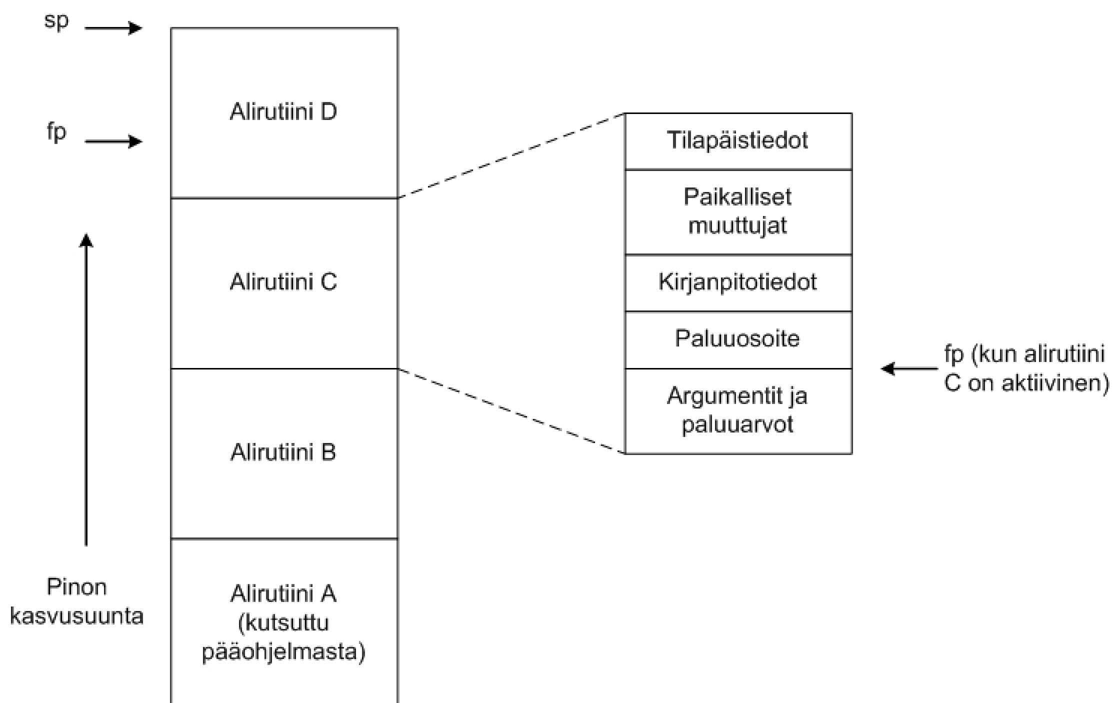
Vähentämällä proseduuri P parametriarvoa *n-1* jokaisella rekursiokierroksella ja vaihtamalla B lausekkeeksi $n > 0$ taataan rekursion päättyminen.

Tietokoneohjelmaa suoritettaessa varataan muistia ohjelman käyttöön jo ennen ohjelman varsinaista ajoa ja myös sen aikana. Kun tilaa varataan ennen ajoa, puhutaan staattisesta varauksesta *(static allocation)*. Ohjelman vakiot, globaalit muuttujat ja iteratiiviset paikalliset muuttujat ovat staattisia objekteja, joille annetaan absoluuttinen muistiosoite koko ohjelman suorituksen ajaksi.

Jos ohjelmointikieli tukee rekursiota, ei staattinen varaus ole enää käytössä paikallisilla muuttujilla. Koska samaan aikaan voidaan tarvita rekursiivisten kutsujen myötä käsitteellisessä mielessä ääretön määrä paikallisten muuttujien osoitteita, ei niitä voida varata etukäteen. Tällöin puhutaan ajonaikaisesta varauksesta *(run-time allocation)* ja rekursion yhteydessä tarkemmin pinoperusteisesta varauksesta *(stack-*

based allocation).

Jokainen rekursiivisen alirutiinin kutsu menee pinoon, jossa sillä on oma kehys *(frame).* Tätä kehystä kutsutaan aktivaatiotietueeksi *(activation record)* ja se sisältää mm. paluuarvot ja muut argumentit, paikalliset muuttujat sekä tilapäis- ja kirjanpitoinformaation. Aktivaatiotietueet ovat pinossa järjestäytyneet LIFO - periaatteen*(last-in, first-out)* mukaan. Normaalien pino-operaatioiden mukaan pinoon voidaan siis lisätä aktivaatiotietue vain pinon päällimmäiseksi ja pinosta voidaan ottaa pois vain pinon päällimmäinen aktivaatiotietue.



Kuva 3: Pinoperusteinen tilanvaraus.

Kuvassa 3 on esitetty pinoperusteinen tilanvaraus. Oletetaan, että aliohjelma A on pääohjelmasta kutsuttu rekursiivinen funktio. Kun aliohjelma A kutsuu itseään, pinon päällimmäiseksi tulee aliohjelma B, ja kun aliohjelma B kutsuu itseään pinon päällimmäiseksi tulee aliohjelma C jne. Pino-osoitin *(stack pointer)* (sp) osoittaa aina pinon ensimmäiseen käyttämättömään paikkaan. Kehysosoitin *(frame pointer)* (fp) osoittaa kyseisen aliohjelman aktivaatiotietueen paikkaan pinossa.

Jokainen rekursiivinen kutsu synnyttää uuden aktivaatiotietueen pinoon. Pino kasvaa jokaisella rekursiokierroksella kunnes rekursio päättyy. Paikallisista muuttujista

luodaan joka kerta uudet ilmentymät. Vaikka muuttujien nimet ovat samat ei sillä ole merkistystä, koska vain pinon päällimmäinen aktivaatiotietue voi olla aktiivinen ja kehysosoitin osoittaa siihen. Kun rekursio päättyy, alkaa pino pienentyä aktivaatiotietueen välittäessä paikallisten muuttujien arvot argumentteina sitä kutsuneelle alirutiinille.

Rekursiivisten algoritmien toteuttamisessa pinorakennetta voidaan käyttää hyväksi [AHU74] silloin, kun ohjelmointikieli ei tue rekursiota. Sen avulla voidaan toteuttaa rekursiiviset algoritmit, joiden iteratiivinen ratkaisu olisi muuten monimutkainen. Tästä kuitenkin enemmän tämän tutkielman luvuissa 3.3 ja 4, joissa käsitellään LabVIEW:n ja rekursion suhdetta sekä rekursion toteutusta.

Rekursion tehokkuutta arvioitaessa törmätään häntärekursion *(tail-recursive)* käsitteeseen. Kyse on rekursiivisista algoritmeista, joissa rekursiokutsu on viimeisenä eikä lisälaskentaa esiinny sen jälkeen. Paluuarvo on siis se, minkä rekursiokutsu ainoastaan palauttaa. Dynaaminen pinonvaraus ei ole häntärekursiivisten algoritmien osalta tarpeellista, koska kääntäjä voi rekursiivisen kutsun yhteydessä uudelleenkäyttää toistokerralle varatun tilan. Itseasiassa kyse on rekursiiviseksi naamioidusta iteraatiosta. Edellä esitelty kertoman rekursiivinen ratkaisu on häntärekursiivinen ja sen aikavaativuus on *O(n)*.

Tilanne muuttuu toiseksi, jos rekursiivisen kutsun jälkeen seuraa toinen rekursiivinen kutsu. Fibonaccin lukusarja voidaan määritellä rekursiivisesti lauseen 12 mukaisesti luonnollisille luvuille N [Als99]:

$$f(1) = f(2) = 1, \quad f(n) = f(n-1) + f(n-2) \quad jos\ n \geq 3 \tag{12}$$

Lauseesta 12 voidaan johtaa rekursiivisen ohjelman, joka on esitetty taulukossa 4.

Kuten ohjelmasta voidaan nähdä, aiheuttaa jokainen *n > 1* kutsu kaksi uutta kutsua. Fibonaccin lukujen rekursiivinen ratkaisu aiheuttaa kutsujen eksponentaalisen kasvun ja sen aikavaativuus on $O(2^n)$. Onneksi Fibonaccin luvuille on löydettävissä yksinkertainen iteratiivinen ratkaisu. [Wir86]

Käytännössä rekursion käyttäminen ratkaisuissa kannattaa tilanteissa, joissa rekur-

```
MODULE fib(n) RETURNS n
     IF n=0 RETURN 0
     ELSEIF n=1 RETURN 1
     ELSE RETURN fib(n-1)+fib(n-2)
     ENDIF
ENDMODULE
```

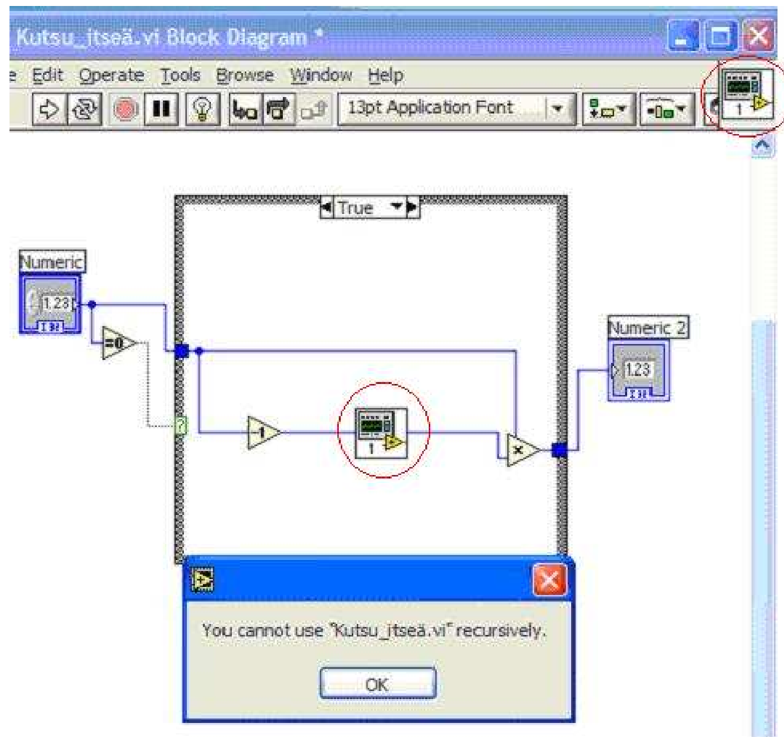
Taulukko 4: Rekursiivinen Fibonacci.

sion syvyys ei ole ääretön. Myös tapaukset, joissa dynaamisesti varattua muistitilaa voidaan uudelleenkäyttää, ovat rekursion käytön kannalta edullisia. Kyse on tällöin ns. Hajoita ja hallitse -algoritmeista *(divide and conquer)* [AHU74]. Niissä ongelman ratkaisua lähestytään jakamalla ongelma pienempiin osiin, etsitään osille ratkaisut ja sen jälkeen yhdistetään osaratkaisut yhdeksi kokonaisuudeksi. Esimerkiksi pikalajittelu *(quicksort)* on Hajoita ja hallitse -algoritmi ja sitä käsitellään tarkemmin tämän tutkielman rekursion toteutusta käsittelevässä luvussa 4.

3.3 LabVIEW ja rekursio

Lähtökohtaisesti LabVIEW ei tue rekursiota. Siihen ei ole rakennettu rekursion mahdollistavia valmiita mekanismeja eikä rekursiivisia tietotyyppejä [Kod]. Jos LabVIEW koodissa yrittää kutsua virtuaali-instrumenttia itseään, saadaan kuvassa 4 nähtävä lopputulos. LabVIEW ilmoittaa, ettei voi käyttää virtuaali-instrumenttia rekursiivisesti. Itse asiassa systeemi ei anna laittaa rekursiivista langoitusta, vaan kyseistä kuvaan on jouduttu manipuloimaan esimerkin aikaansaamiseksi.

Kuten kappaleessa 2.3 mainitaan, perustuu LabVIEW tietovuo- eli graafimalliin. Se on vaihtoehtoinen malli von Neumannin laskentamallille ja arkkitehtuurille [GMK06]. Imperatiivinen ohjelmointiparadigma yhdistetään yleensä lähes synonyyminä von Neumann -pohjaiselle mallille. Kun ohjelman suoritus perustuu peräkkäisyyteen puhutaan kontrollivuosta. Siinä ohjelmakoodin kirjoitusjärjestys ja siinä olevat kontrollirakenteet ohjaavat suoritusta. Tietovuomallissa itse tieto *(data)* on ohjaava tekijä. Siinä tieto käsitellään liikkuvana elementtinä, tietoalkiona *(data token)*, joka virtaa paikasta A paikkaan B. Näitä paikkoja A ja B voidaan pitää graafin solmuina ja näiden välisiä yhteyksiä solmujen kaarina. Tietovuo-ohjelmassa voi liikkua samanaikaisesti useita tietoalkioita, joiden kaikkien on oltava kaaritetun solmun käytössä ennen



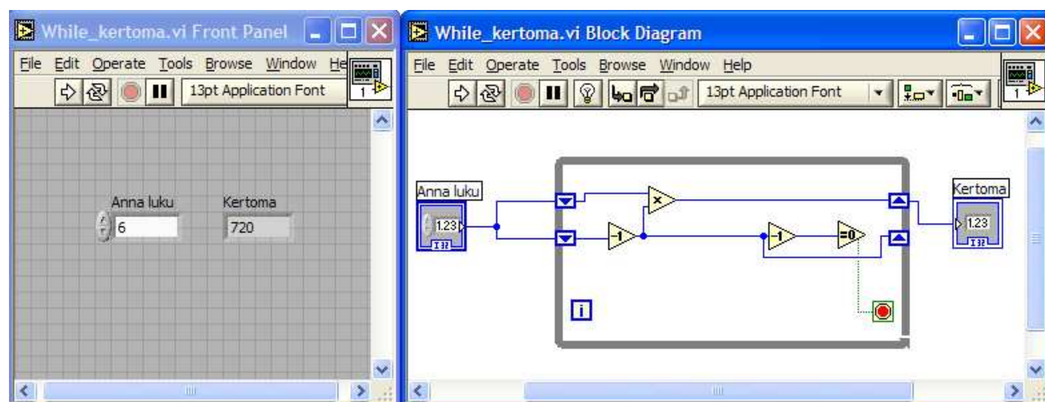
Kuva 4: LabVIEW:n virtuaali-instrumentti yrittää kutsua itseään.

kuin suoritus voi jatkua. Näin muodostuu tietovuograafi, joka on suunnattu graafi. Koska LabVIEW-ohjelmassa ei voida langoittaa virtuaali-intrumenttia itseensä, edes epäsuorasti, muodostuu syklitön suunnattu graafi.

Tietovuomalli ei sinänsä ole este rekursion toteuttamiselle. Rekursion toteuttaminen tietovuo-ohjelmointikielessä edellyttää kuitenkin dynaamista tietovuoarkkitehtuuria. Esimerkiksi Sisal-ohjelmointikielessä se on toteutettu [Fin96]. Myös LabVIEW:n G-kieli ei luontaisesti sulje rekursiota pois. Onhan iteratiiviset toisto- ja case-rakenteet toteutettu kielessä erillisin ratkaisuin. Niiden toteutus sallii syklisyyden, tosin vain näiden rakenteiden sisällä. Rekursiomekanismien puuttuminen johtuu G-kielen sisäisestä suoritusmallista, joka estää rekursion tehokkaan toteutuksen [Gup02]. Jostain syystä National Instruments ei ole halunnut näitä mekanismeja G-kieleen liittää. Kuvaavaa on LabVIEW:n kehittäjän Jeff Kodoskyn artikkelissaan *Is LabVIEW a general purpose programming language?* esiintuoma kanta, ettei rekursion puuttuminen ole vakava ongelma yleiskäyttöisten ohjelmien rakentamisessa [Kod].

Kuinka rekursiiviset algoritmit voidaan toteuttaan LabVIEW-ohjelmointiympäristös-

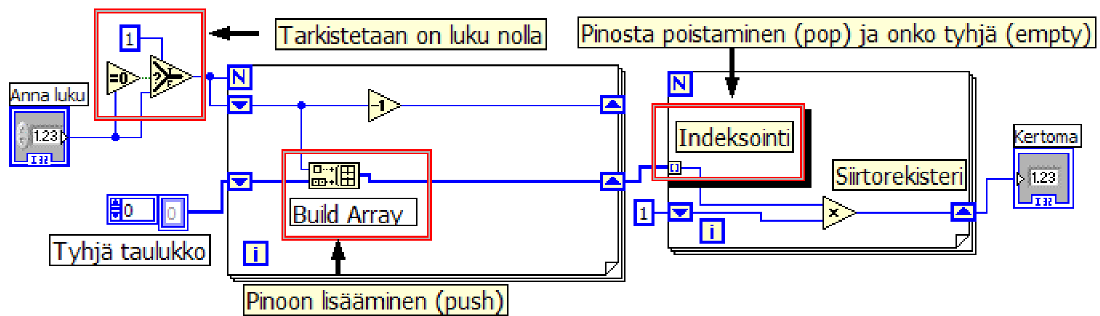
sä? Häntärekursiivisten algoritmien osalta se ei ole ongelma, koska ne voidaan aina helposti toteuttaa iteratiivisesti toistosilmukan avulla. Kappaleessa 2.3 kuvassa 2 on esitetty kertoman iteratiivinen muoto *for*-silmukan avulla. Kuvassa 5 on kertoma toteutettu *while*-silmukan avulla.



Kuva 5: Kertoma while silmukan avulla.

Kuvista huomataan, että toistorakenteiden sivuilla on ylös/alas -nuolia. Kyseiset kuvalliset esitykset ovat siirtorekistereitä *(shift register)*. Kysymyksessä on LabVIEW:n tapa siirtää historiatietoa seuraavalle toistokerralle. Tässä mielessä niitä voidaan pitää muuttujina, mutta mitään todellisia tekstuaalisiin ohjelmointikieliin rinnastettavia muuttujia ne eivät ole, koska niihin ei voi viitata. Erikoista siirtorekistereissä on se, että samaa rekisteriä voidaan monistaa siten, että se muistaa erillisten kierrosten historiatiedon. Esimerkiksi, jos toistokierroksia on kolme voidaan viimeiselle toistokerralle tuoda myös ensimmäisen toistokerran tulos käytettäväksi.

Muiden kuin häntärekursiivisten algoritmien osalta on LabVIEW-ympäristössä käytettävä toisia menetelmiä, jos algoritmi halutaan toteuttaa rekursioluonteisesti. Rekursiivista rakennetta voidaan simuloida pinon avulla. LabVIEW ei sisällä valmista pinorakennetta, mutta se voidaan toteuttaa helposti esimerkiksi yksi- tai useampiulotteisen taulukkorakenteen avulla. Kun taulukkoa käytetään pinona, operoidaan vain indeksin nolla (0) kanssa, joka samalla abstrahoi pino-osoittimen. Pino-operaatioista on pystyttävä toteuttamaan pinoon lisääminen *(push)*, joka lisää uuden alkion taulukon ensimmäiseksi (index 0) ja kasvattaa taulukon pituutta yhdellä. Pinosta poistaminen *(pop)* lukee taulukon ensimmäisen alkion (index 0), poistaa sen ja lyhentää taulukkoa yhdellä. Lisäksi tarvitaan myös operaatio, millä todetaan pinon olevan tyhjä *(empty)* eli sen, ettei taulukossa ole yhtään alkiota.

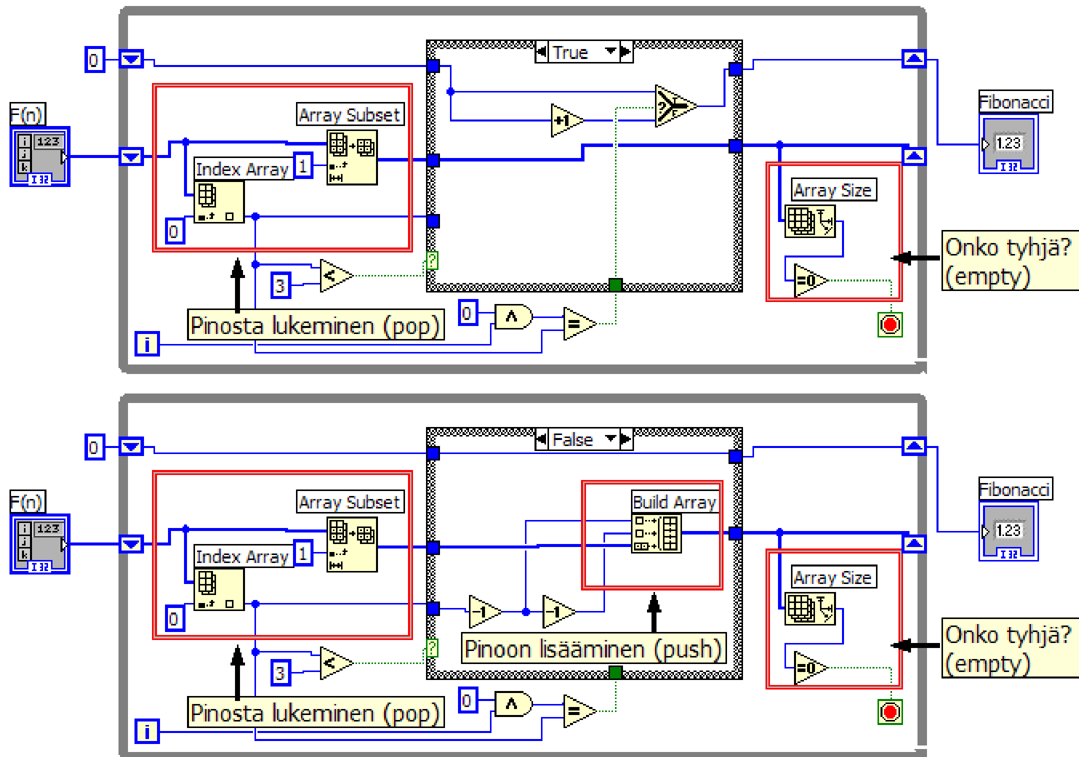


Kuva 6: Kertoma toteutettuna pinon avulla.

Kuvassa 6 on kertoma toteutettu pinon avulla. Sen ohjelmoiminen pinoratkaisuna ei kylläkään ole mielekästä, koska se on iteratiivisesti paljon helpommin toteutettavissa. Kyseinen esimerkki tuo kuitenkin esille erään LabVIEW:n mielenkiintoisen ominaisuuden, indeksoinnin.

Kuvan 6 esimerkissä tarkistetaan ensin, onko kyseessä alkeistapaus eli onko syötetty luku nolla. Silloin *for*-silmukat käydään läpi vain kerran. Jos annettu luku on jokin muu, käydään silmukkakierroksia annetun luvun verran. Ensimmäisessä *for*-silmukassa annettua lukua vähennetään yhdellä jokaisella kierroksella ja siirtorekisteri välittää kyseisen tiedon seuraavalle toistokierrokselle. *Build Array* on ohjelman alkaessa alustettu tyhjällä taulukolla. Jokaisella toistokerralla pinoon (taulukkoon) viedään luku päällimmäiseksi ja siirtorekisteri välittää syntyneen pinon seuraavan toistokierroksen pohjaksi. Kun ensimmäisen *for*-silmukan suoritus päättyy, muodostuneen pinon päällimmäisenä alkiona on luku yksi ja pohjimmaisena annettu luku.

Esimerkin toinen *for*-silmukka saa näin muodostuneen pinon (taulukko) indeksointitunnelin *(loop tunnel)* kautta. Sen kuvaajana on pienet hakasulut toisen *for*-silmukan vasemmassa laidassa. Kun indeksointi otetaan käyttöön toistosilmukassa, taulukko käydään läpi alkio kerrallaan siten, että joka kierrokselle otetaan käsittelyyn seuraava alkio. Tätä jatketaan kunnes taulukko on läpikäyty. Indeksoinnin etuna on, että taulukon koko määrää toistokerrat eikä mitään erillistä lopetusehtoa tai toistokierrosten lukumäärää tarvitse asettaa. Mikään todellinen pino-operaatio indeksointi ei kuitenkaan ole. Siinä ei käsitellä vain pinon päällimmäistä alkiota (index 0) eikä poisteta pinosta yhtään alkiota. Indeksoinnin käyttö rekursiivisissa pinoratkaisuissa on mahdollista vain silloin, kun missään vaiheessa ohjelman suoritusta pinoon ei enää lisätä eikä siitä poisteta yhtään alkiota.



Kuva 7: Fibonaccin lukujonon rekursiivinen pinoratkaisu.

Kuvassa 7 esitellään pinoon perustuva rekursiivinen Fibonaccin lukusarjan läpikäyntisovellus [GMK06]. Kappaleessa 3.2 on lauseessa 12 esitetty Fibonaccin lukusarjan määrittely ja samassa kappaleessa on taulukossa 4 esitetty sen pseudokielinen ohjelmakoodi. Pinoon perustuva, rekursiota simuloiva LabVIEW-ohjelma on toteutettu *while*-silmukan ja *case*-rakenteen avulla. Ohjelmassa on toteutettu pino-operaatiot *push, pop* ja *empty.*

Pinon lukemisoperaatio *(pop)* on toteutettu LabVIEW:n valmiiden taulukkofunktioiden *Index Array* ja *Array Subset* avulla. *Index Array* palauttaa annetun indeksin arvon, joka on pinon tapauksessa päällimmäinen eli index 0. *Array Subset* palauttaa taulukon (pinon) annetusta indeksistä lähtien. Kun funktiolle annetaan indeksin arvoksi 1 saadaan pino, josta on poistettu päällimmäinen alkio.

Pinoon lisäämisoperaatio *(push)* on toteutettu taulukkofunktion *Buil Array* avulla. Siinä pinon alimmaiseksi asetetaan jo olemassaoleva pino ja sen päälle lisätään alkiot, jotka ovat *Fibonaccin* lukusarjan kohdalla $F_{(n-1)}$ ja $F_{(n-2)}$. Sen toteamiseen onko pino tyhjä *(empty)* käytetään taulukkofunktiota *Array Size*. Kun taulukon ko-

ko on 0, on myös pino tyhjä.

$F_{(n)}$:n arvoilla 0, 1 ja 2 ovat ohjelmassa alkeistapauksia, joissa *while*-silmukka ja *case*-rakenteen *true*-tapaus suoritetaan vain kerran. Jos $F_{(n)} = 0$ siirtorekisterin arvoa ei kasvateta yhdellä, vaan vastaus on 0. Jos $F_{(n)} = 1$ tai $F_{(n)} = 2$ niin siirtorekisterin arvoa kasvatetaan yhdellä ja kummassakin tapauksessa vastaus on 1.

Kahta suuremmilla $F_{(n)}$ arvoilla joudutaan *while*-silmukka suorittamaan useammin kuin kerran. Pinosta luetaan päällimmäinen arvo, jos se on suurempi kuin 2, pinoa kasvatetaan *case*-rakenteen *false*-tapauksen $F_{(n-1)}$ ja $F_{(n-2)}$ arvoilla. Tätä jatketaan niin kauan kunnes saavutetaan jokin alkeistapaus, jolloin kasvatetaan siirtorekisterin arvoa yhdellä. Näin jatketaan niin kauan kunnes pino on tyhjä. Itseasiassa kyseinen ohjelma on yksinkertainen silmukkalaskuri, jossa *true*-tapausten määrä on sama kun kyseessäoleva $F_{(n)}$ *Fibonaccin* luku.

Ohjelma tuottaa *Fibonaccin* lukujonon *(0,1,1,2,3,5,8,13...)* mukaisen tuloksen eri $F_{(n)}$ arvoilla. Se on kuitenkin varsin tehoton ja hidas. Kun *while*-silmukka triviaalitapauksissa käydään vain kerran ja $F_{(4)}$ arvolla vielä siedettävät viisi kierrosta, käydään *while*-silmukka $F_{(8)}$ arvolla jo 41 kierrosta.

Häntärekursion toteuttaminen toistorakenteen ja siirtorekisterin avulla sekä muiden rekursiotyyppien toteuttaminen pinon avulla ovat tietovuomallin mukaisia ratkaisuja. LabVIEW tarjoaa kuitenkin myös muita ratkaisumenetelmiä rekursiivisten algoritmien toteuttamiseen. Näitä vaihtoehtoisia malleja sekä pinon hyväksikäyttöä pikalajittelun *(quicksort)* yhteydessä käsitellään rekursion toteutusta käsittelevässä luvussa 4.

4 REKURSION TOTEUTUS JA TESTAUS

Vaikka LabVIEW:n G-ohjelmointikieli ja sen tietovuomalli eivät tue rekursiota, on ohjelmistokehitysvälineeseen lisätty ominaisuuksia, joiden avulla rekursio voidaan toteuttaa. Ohjelmistokehitysvälineeseen on rakennettu mekanismi, jolla voidaan rakentaa visuaalisia elementtejä yleisimmillä tekstuaalisilla ohjelmontikielillä ja liittää ne käytettäväksi LabVIEW-ympäristössä. Von Neumannilaisesta ohjelmointimaailmasta tutut rekursiiviset algoritmit voidaan täten toteuttaa tällä menetelmällä. LabVIEW-ympäristössä on myös valmiit virtuaali-instrumentit palvelinohjelmistojen *(VI Server)* rakentamiseen. Vaikka näitä instrumenttejä ei alunperin olekaan tarkoitettu rekursion toteuttamiseen, suovat ne tämän mahdollisuuden luomalla samasta virtuaali-instrumentista useita ilmentymiä palvelinkäytön mahdollistamiseksi. Tässä kappaleessa toteutetaan edellämainituilla tekniikoilla pikalajittelu *(quicksort)* esimerkit. Mukaan on otettu myös pinoperusteinen pikalajittelu. Kappaleen lopuksi suoritetaan pienimuotonen suorituskykymittaus esimerkkiratkaisuiden välillä. Niitä myös verrataan LabVIEW:n valmiiseen yleiskäyttöiseen lajitteluinstrumenttiin. Tämän kappaleen päälähteinä ovat LabVIEW manuaalit *Using External Code in LabVIEW* [Nat03b] ja *LabVIEW 7 Express, User Manual* [Nat03a] sekä *National Instrumentsin LabVIEW internetsivustot* [Nat07].

4.1 Pikalajittelu

Lajittelualgoritmit voidaan luokitella niiden käyttämän pääperiaatteen mukaan lisäys *(insertion)*, valinta *(selection)* tai vaihto *(exchange)* lajitteluiksi [Wir86]. Pikalajittelu kuuluu viimeksimainituun ryhmään. Sen kehitti vuonna 1962 *C.A.R. Hoare.* Hän antoi sen tuoman valtavan suorituskyvyn takia sille nimeksi *quicksort.* Pikalajittelu perustuu siihen tosiasiaan, että vaihto on tehokkain menetelmä silloin, kun lajiteltavien alkoiden määrä on suuri [Wir86].

Kuten kappaleessa 3.2 mainittiin kuuluu pikalajittelu myös Hajoita ja hallitse -algoritmeihin. Näiden algoritmien suoritus voidaan jakaa kolmeen vaiheeseen [Als99]:

- Hajoitavaihe *(divide step)*, jossa lajiteltava syöte*(n)* jaetaan $p \geq 1$ osaan, joista jokaisen osan laajuuden täytyy olla pienenpi kuin n.

- Hallitsevaihe *(conquer step)* sisältää osien*(p)* rekursiiviset kutsut.
- Yhdistämisvaihe *(combine step)*, jossa osien*(p)* rekursiivisten kutsujen tulokset yhdistetään, jotta saavutetaan haluttu ratkaisu.

Toisin sanoen Hajoita ja hallitse -algoritmin kulku on seuraavanlainen. Jos tapaus I on "pieni", ratkaistaan ongelma suoraan ja palautetaan ratkaisu. Kyse on tällöin triviaalista tapauksesta. Muuten jaetaan tapaus I alitapauksisksi $I_1, I_2, \ldots, I_p$. Alitapausten laajuuden täytyy olla suurinpiirtein samanlainen. Rekursiivisesti kutsutaan algoritmia jokaisen alitapauksen $I_j, 1 \leq j \leq p$ kohdalla, jotta saadaan alitapausten osittaisratkaisut. Lopuksi yhdistetään alitapausten osaratkaisut alkuperäiseksi tapaukseksi I ja palautetaan tapauksen I ratkaisu. [Als99]

```
10   PROCEDURE Quicksort;

30     PROCEDURE sort(L,R:index);
40       VAR i, j: index; w, x: item;
50     BEGIN i:=L; j:=R;
60       x:= a[(L+R) DIV 2];
70       REPEAT
80         WHILE a[i] < x DO i:= i+1 END;
90         WHILE x < a[j] DO j:= j-1 END;
100        IF i <= j THEN
110          w:= a[i]; a[i]:= a[j]; a[j]:= w;
120          i:= i+1; j:= j-1;
130        END
140      UNTIL i > j;
150      IF L < j THEN sort(L, j) END;
160      IF i < R THEN sort(i, R) END;
170    END sort;

190  BEGIN sort(1, n)
200  END Quicksort
```

Taulukko 5: Quicksort algoritmin tekstuaalinen rekursiivinen esitys [Wir86].

Taulukossa 5 on pikalajittelu algoritmin tekstuaalinen esitys. Sen aikavaativuus on keskimäärin *O(n log n)*, mutta huonoimmassa tapauksssa aikavaativuus kasvaa $O(n^2)$ asti. Parhaimmassa tapauksessa pikalajittelun aikavaativuus on vain *O(n/2)*, tällöin lajiteltavien alkioiden on kuitenkin oltava käännetyssä järjestyksessä. [Wir86]

Pikalajittelun etuna moniin muihin lajittelualgoritmeihin verrattuna voidaan pitää sitä, ettei se tarvitse lajitteluun mitään aputaulukoita tai muuta ylimääräistä tiedon varastointitilaa. Algoritmi käyttää lajittelussa suoraan lajiteltavaa tietovarastoa ja lajittelu tapahtuu siinä. Itse asiassa pikalajittelusta puuttuu kokonaan Hajoita ja hallitse -algoritmien yhdistämisvaihe [Als99]. Rekursion yhteydessä tästä on suurta hyötyä, koska lajiteltavasta tietovarastosta ei tarvitse joka rekursiokierroksella tehdä uutta ilmentymää. Tietokoneen muistitilaa säästyy, koska lajiteltavan tietovaraston muistiavaruus voidaan varata staattisesti eikä dynaamista varausta tarvita. Kuten luvusssa 3.2 mainitaan, paikallisille muuttujille tarvitaan dynaaminen muistinvaraus myös pikalajittelun yhteydessä, joten siltä osin tarvittavan muistin määrä riippuu rekursiokierrosten lukumäärästä.

Pikalajittelualgoritmin idea on yksinkertainen. Sen ytimenä on ositusalgoritmi *(partitioning algorithm)*, jonka avulla lajiteltavasta taulukosta ensin etsitään alkio, jota kutsutaan keskuselementiksi *(pivot element)* tai jakoelementiksi *(splitting element)*. Se mikä taulukon alkioista valitaan jakoelementiksi voidaan suorittaa useammalla tavalla. Se voi olla taulukon ensimmäinen alkio, se voidaan valita satunnaisesti tai se voidaan valita algoritmille annettujen indeksiparametrien keskiarvona, kuten taulukossa 5 on tehty rivillä 60. Taulukon muita alkioita verrataan tähän jakoelementtiin. Jos taulukossa on ennen jakoelementtiä sitä suurempi alkio, se vaihtaa paikkaa keskenään sellaisen alkion kanssa, joka on jakoelementin jälkeen taulukossa ja sitä pienempi. Näin jatketaan kunnes taulukon läpikäynnit sekä sen alusta että lopusta saavuttavat jakoelementin indeksin. Tätä esittävä ohjelmakoodi on taulukon 5 riveillä 70 - 140. Ositusalgoritmin lisäksi pikalajittelualgoritmiin kuuluu kaksi rekursiivista kutsua, joista ensimmäinen kohdistuu taulukon alkupuoliskolle ja toinen taulukon loppupuolelle. Tämä pätee kylläkin vain lajittelualgoritmin ensimmäisellä suorituskerralla. Rekursion myötä muuttuvat myös algoritmin saamat indeksiparametrit.

Oletetaan, että on taulukko *a[5,3,9,2,7,1,8]*, joka täytyy lajitella suuruusjärjestykseen pienimmästä suurimpaan. Taulukossa on seitsemän alkiota, joten *sort(L,R)* saa parametreikseen L:=1 ja R:= 7. Parametri L on siis taulukon ensimmäisen alkion indeksi ja parametri R on taulukon viimeisen alkion indeksi. Seuraavaksi alustetaan paikalliset apumuuttujat i:=L ja j:=R. Tämän jälkeen etsitään jakoelementin indeksi taulukon keskivaiheelta kaavalla *a[(L+R) DIV 2]*, jonka jälkeen sen osoittaman tau-

lukon indeksin alkion arvo asetetaan jakoelementti *x:n* arvoksi. Tässä tapauksessa indeksin arvoksi tulee 4 ja kyseisen indeksin alkion arvo on 2, joka tulee jakoelementti x:n arvoksi. On huomattava, ettei jakoelementin arvo muutu ositusalgoritmin suorituksen aikana, vaikka sen alkuperäisen indeksin osoittama arvo muuttuisi. Jakoelementin arvo voi vaihtua vain uuden rekursiokutsun takia.

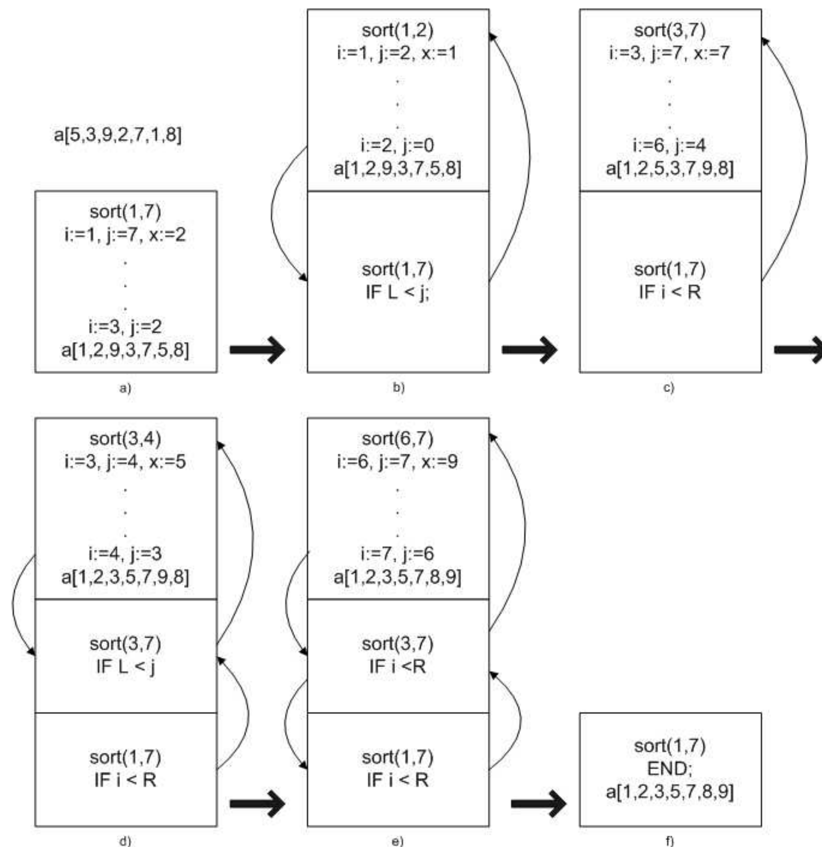
Seuraavaksi siirrytään *REPEAT-UNTIL* toistosilmukkaan. Ensimmäisessä *WHILE* silmukassa taulukon ensimmäistä alkiota verrataan jakoelementtiin *(WHILE a[i]<x DO i:= i+1 END;)*. Koska tässä tapauksessa lause on epätosi (5<2) ei i:n arvoa kasvateta yhdellä, vaan siirryrään toiseen *WHILE x<a[j] DO j:=j-1 END* silmukkaan. Siinä verrataan jakoelementtiä taulukon viimeiseen alkioon ja koska lause on tosi (2<8), vähennetään j:n arvoa yhdellä. Toinen *WHILE* silmukka suoritetaan uudelleen j:n arvolla 6 ja koska nyt lause on epätosi (2<1) ei j:n arvoa enää vähennetä vaan siirrytään lauseeseen *IF i<=j THEN w:=a[i]; a[i]:=a[j]; a[j]:=w; i+1; j-1; END*. Koska ehto on tosi *(i:=1<j:=6)* vaihdetaan indeksien osoittamien alkioiden paikkaa keskenään ja kasvatetaan i:n arvoa yhdellä sekä vähennetään j:n arvoa yhdellä. Taulukko a on nyt muodossa *[1,3,9,2,7,5,8]*. Koska lopetusehto *UNTIL i>j* on epätosi suoritetaan *REPEAT-UNTIL* silmukka uudelleen.

Toisen suorituskerran jälkeen taulukko a on muodossa *[1,2,9,3,7,5,8]* ja i:=3 ja j:=3. Koska lopetusehto ei vieläkään ole tosi suoritetaan *REPEAT-UNTIL* silmukka kolmannen kerran. Tällä suorituskerralla *WHILE* silmukoissa ainoastaan j:n arvoa vähennetään yhdellä. Koska i:=3 ja j:=2 ei *IF* lauseen *THEN* osaa suoriteta lainkaan ja lopetusehtokin täyttyy lopettaen *REPEAT-UNTIL* silmukan suorituksen. Taulukko a pysyy muuttumattomana.

Pikalajittelun seuraavassa vaiheessa siirrytään rekursiokutsuihin ja niitä edeltäviin ehtoihin. Jos lause *IF L < j* on tosi suoritetaan rekursiivinenkutsu *sort(L, j)*. Koska L:=1 ja j:=2 niin suoritetaan *sort(1,2)*. Apumuuttujat ovat i:=1 ja j:=2 sekä jakoelementti on x:=1. Tässä tapauksessa on huomioitavaa se, että lajittelu kohdistuu ainoastaan taulukon kahteen ensimmäiseen alkioon. Koska tiedämme niiden olevan jo oikeassa järjestuksessä on suoritus lajittelun kannalta turha. Rekursion päättymisen kannalta kutsu ei kuitenkaan ole tarpeeton, koska se vaikutaa apumuuttujiin ja siten myös rekursiokierroksen päättymiseen. Lopetusehdon täyttyessä ovat i:=2, j:=0 ja taulukko a on pysynyt muuttumattomana. Koska myös ehdot *IF L<j* ja *IF i<R* ovat epätoseja päättyy rekursio ja palataan alkuperäiseen suoritukseen.

Seuraavaksi testataan toisen rekursiokutsun ehto *IF* $i<R$ ja jos se on tosi suoritetaan rekursio *sort(i, R)*. Koska palasimme alkuperäiseen suoritukseen ovat myös muuttujien ja parametrien arvot ne, mitkä ne olivat ennen ensimmäistä rekursiokutsua. Apumuuttuja i:=3 ja parametri R:=7. Rekursion suoritusehto ($3<7$) on tosi, joten suoritetaan rekursiivinenkutsu *sort(3,7)*. Ositusalgoritmin suorituksen jälkeen taulukko a on muodossa *[1,2,5,3,7,9,8]*. Jakolementti x:=7 ja apumuuttujat ovat i:=6 ja j:=4.

Koska *IF* $L<j$ on tosi suoritetaan rekursiivinen kutsu *sort(3,4)*, jolloin jakoelementti x:=5. Ositusalgoritmin suorituksen jälkeen taulukko a on muodossa *[1,2,3,5,7,9,8]* ja apumuuttujien arvot ovat 1:=4 ja j:=3. Rekursiokutsujen ehdot eivät täyty, joten palataan edelliseen suoritukseen.



Kuva 8: Kaavio pikalajittelun rekursiokierroksista.

Vuorossa on rekursioehdon *IF* $i<R$ testaaminen arvoilla i:=6 ja R:=7. Koska lause on tosi suoritetaan rekursiivinenkutsu *sort(6,7)*. Jakoelementti x:=9, apumuuttujat i:=6 ja j:=7. Ositusalgoritmin suorituksen päätyttyä taulukko a on muodossa

[1,2,3,5,7,8,9] sekä apumuuttujat i:= 7 ja j:=6. Kummankaan rekursiivisenkutsun ehdot eivät täyty, joten palataan *sort(3,7)* suoritukseen. Myös sen suoritus päättyy ja palataa takaisin alkuperäiseen suoritukseen *sort(1,7)*. Pikalajittelualgoritmin suorittamisessa tulemme taulukon 5 ohjelmakoodin riville 170 *(END sort;)*, joka samalla päättää pikalajittelualgoritmin suorituksen ja tuloksena olemme saaneet lajitellun taulukon *a[1,2,3,5,7,8,9]*.

Kuvassa 8 on edellä mainitun esimerkin rekursiokierrokset kuvattuina pinon avulla. Ainostaan pinon päällimmäinen suoritus on aktiivinen ja muut ovat odotustilassa. Ne aktivoituvat vasta, kun niiden yläpuolella oleva rekursio on päättynyt.

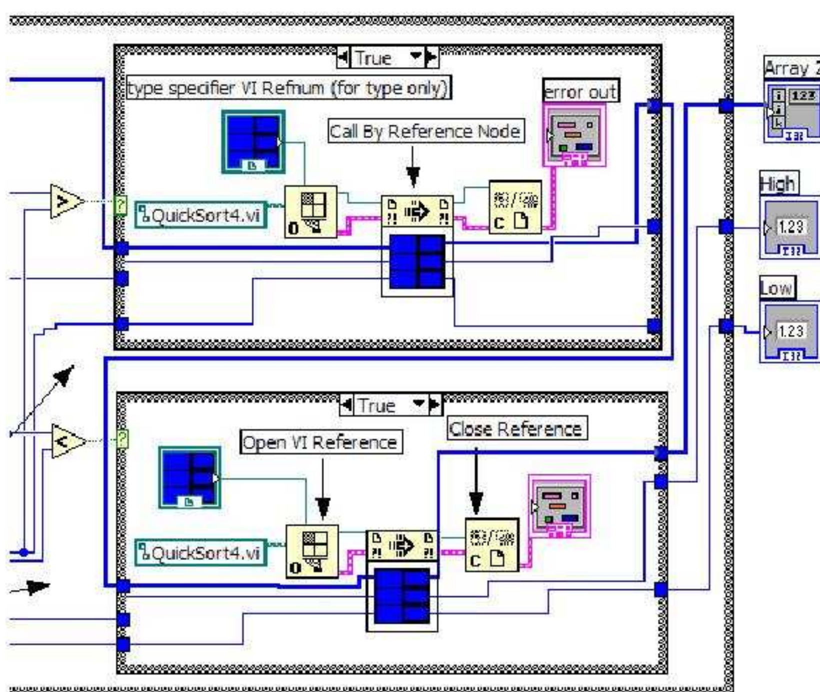
4.2 Pikalajittelu VI Serverin avulla

VI Server on joukko funktioita, jotka ovat olleet mukana LabVIEW -ohjelmointipärisössä versiosta 5.0 alkaen. Niiden avulla päästään ohjelmallisesti käsiksi LabVIEW:n virtuaali-instrumentteihin ja voidaan näin puuttua niiden toimintaan. VI Server -funktioiden arkkitehtuuri perustuu viitenumerontiin *(reference number, refnum)* [Nat03c]. Se on yksilöllinen tunnistin, joka luodaan viitattavaan objektiin. Tälläisiä objekteja voivat olla tiedostot, laitteet tai verkkoyhteydet. Koska viitenumero on vain väliaikainen osoitin *(pointer)* avoimena olevaan objektiin, on se voimassa vain sen ajan kun objekti on avattuna [Nat03a]. Jos esimerkiksi tiedosto suljetaan ei sama osoitin enää toimi, vaikka sama tiedosto avattaisiin uudelleen. Sille on luotava tällöin uusi yksilöllinen osoitin. Jos sama tiedosto avataan useamman kerran ilman, että se suljetaan välillä, luodaan joka avauksen yhteydessä uusi yksilöllinen tunnistin kyseiseen tiedostoon. Rekursion toteuttaminen VI Server -funktioiden avulla perustuu juuri tähän ominaisuuteen.

Viitteen lisäksi tarvitaan funktiot, jotka luovat nämä viitteet dynaamisesti. Kuten kappaleessa 3.2 mainitaan, staattinen varaus ei ole rekursion yhteydessä mahdollista, koska rekursiokierrosten lukumäärä ei voi etukäteen määritellä. Normaalisti dynaamisesti kutsuttavilla aliohjelmilla on se hyvä puoli, etteivät ne kuluta turhaan muistia. Niitä kutsutaan tarvittaessa, varataan muistia ajonaikana ja vapautetaan muisti ajon päätyttyä. Rekursion yhteydessä tästä ominaisuudesta ei ole hyötyä, sillä rekursiivisia kutsuja voi olla varattuna niin monta, ettei muistitila riitä.

Aiemmin on mainittu, ettei LabVIEW salli ohjelman tai funktion kutsuvan itseään eli kuvakkeet eivät voi olla langoitettu siten, että ne muodostavat syklin. Kuitenkin VI Serverin staattiset ja dynaamiset funktiot sallivat myös kutsut ohjelmaan, jossa ne itse sijaitsevat. Tämä johtuu funktioiden alkuperäisestä tarkoituksesta. Palvelinsovellusten on pystyttävä luomaan useampia yhtäaikaisia yhteyksiä samaan laitteeseen, verkkoon tai tiedostoon. Itsensä kutsumista ei ole rajattu tämän ulkopuolelle.

Rekursion toteuttamiseen tarvitaan siis yksilöllinen viitenumero, menetelmä luoda tämä viite ajonaikaisesti sekä mahdollisuuden viitata virtuaali-instrumenttiin, joka sisältää kutsuvan kuvakkeen. Tämän lisäksi tarvitaan menetelmä, jolla voidaan välittää parametreja suoritusten välillä. LabVIEW:n VI Serverin dynaamisella funktiolla *Call by Reference Node* ja siihen liitetyllä vahvasti tyypitetyllä virtuaali-insrumentin viitenumerolla *(Strictly Typed VI Refnum)* se voidaan tehdä.

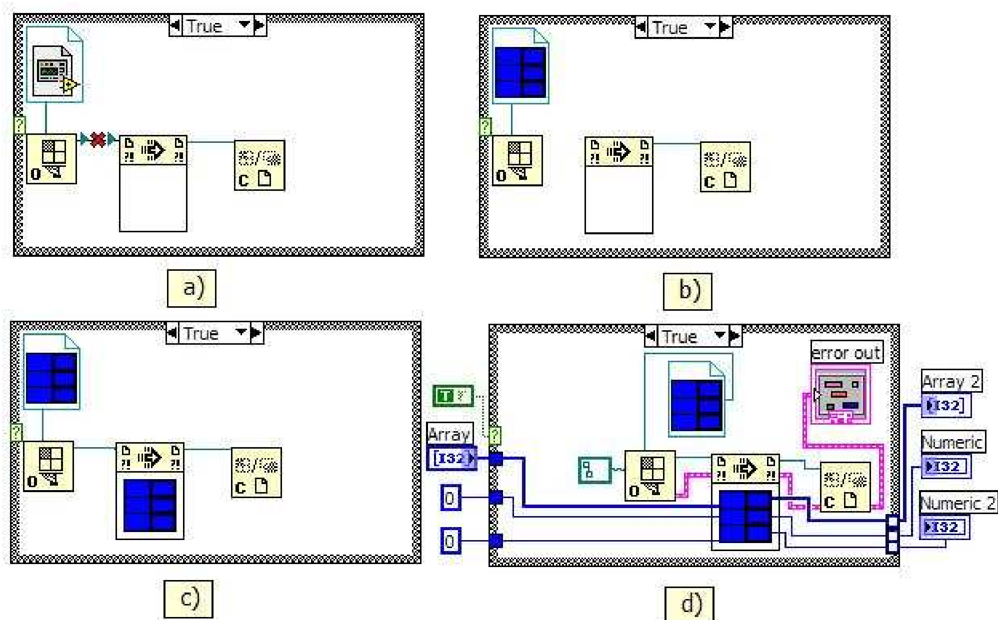


Kuva 9: Vi Server pikalajittelun rekursiiviset osat.

Liitteessä A on pikalajittelun toteutus edellämainituilla elementeillä. Ohjelman rakenne on samanlainen kun taulukossa 5 kuvattu tekstuaalisen pikalajittelualgoritmin. Ohjelma saa syötteenä yksiulotteisen taulukon ja sen alku- ja loppuindeksit. Se valitsee jakoelementin ja suorittaa ositusalgoritmin samoin kun on edellä esitetty. Rekursiiviset kutsut sen sijaan tehdään *Call by Reference Node* -funktion avulla.

Kuvassa 9 on osasuurennos pikalajittelualgoritmin rekursiivisesta osasta. Case -rakenteiden sisälle on koottu kaikki ne elementit, mitä rekursion suoritukseen vaaditaan. Ensin avataan viite tiedostoon *(Open VI Reference)*, sen jälkeen suoritetaan viitattu ohjelma annetuilla parametreilla *(Call by Reference Node)* ja lopuksi suljetaan viite ja ohjelman suoritus *(Close Reference)*.

Open VI Reference solmun vaatima ainoa pakollinen tieto on polku avattavaan virtuaali-instrumenttiin, joka tässä tapauksessa on polku itseensä. Solmu tuottaa tuloksena viitteen polussa mainittuun virtuaali-instrumenttiin. Tämä pelkkä viite ei *Call by Reference Node* -solmun yhteydessä riitä. Kuten yllä mainittiin, tarvitaan funktion käyttämiseksi siihen liitetty vahvasti tyypitetty viitenumero. Tämän takia on *Open VI Reference* -solmuun liitettävä vielä tämä tyypitys *type specifier VI Refnum* -määrityksen avulla.



Kuva 10: Vahvan tyypityksen luominen

Viittausnumeroa joudutaan vielä säätämään, koska se on oletusarvoisesti virtuaali-instrumentti -tyyppinen. Valitaan kuvakkeen valikosta *SelectVI Server Class* ja sen *Strictly Typed VIs* vaihtoehto. Kuvassa 10 on kuvasarjalla osoitettu vahvan tyypityksen luominen. Kuvan *a)* kohdasta huomataan, ettei pelkän *type specifier VI Refnum* -määrityksen luominen riitä. Jos yritetään langoittaa viitteen *Call by Reference Node* -solmuun, ei se onnistu ja langoitus jää rikkinäiseksi. Muuttamalla

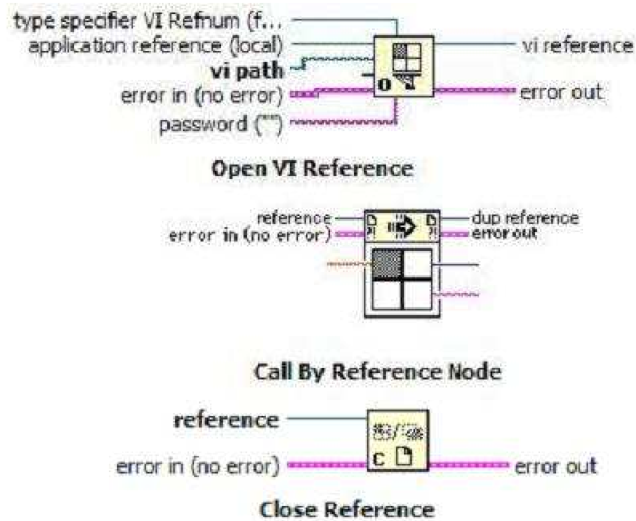
viitteen vahvasti tyypitetyksi, kuten kuvasarjan kohdassa *b)* on tehty, muuttaa kuvake muotoaan. Kun nyt langoitamme *Open VI Reference* -solmun *Call by Reference Node* -solmuun, ilmestyy sen alle liitinpareja, joiden avulla parametrien välitys onnistuu, kuten kuvasarjan kohdista *c)* ja *d)* voidaan todeta.

Normaalisti LabVIEW -ohjelma ajetaan yhdessä ja samassa säikeessä *(thread)*, jossa käyttöliittymä ja suoritettavat tehtävät vuorottelevat. Kun rekursio toteutetaan *Call by Reference Node* -solmun avulla tulee säikeen hallinta kyseiselle solmulle, joten LabVIEW ei voi käyttää samaa suoritussäiettä *(execution thread)* muihin tehtäviin. Tämä johtuu siitä, että säikeen koko kontrolli siirtyy ajettavan solmun haltuun. Koska kontrolli on ajettavalla solmulla, ei LabVIEW voi keskeyttää ohjelman suoritusta. Sen on odotettava suorituksen päättymistä, ennen kuin se saa säikeen taas haltuunsa. Rekursion yhteydessä kyseinen toimintamalli ei toimi, sillä siinä on pystyttävä aloittamaan funktion tai proseduurin suoritus ennen kuin edellisen suoritus on päättynyt. Jotta rekursio olisi mahdollinen, on oltava jokin keino puuttua suoritussäikeen toimintaan tai kiertää se.

Ratkaisu on LabVIEW -ohjelman rakentaminen monisäikeiseksi *(multithread)*. Luomalla rekursion sisältävät ohjelmanosat ns. vapaakäyntisiksi ohjelmiksi *(Reentrant Program)* mahdollistetaan useamman säikeen käyttö. Ohjelmat, jotka ovat vapaakäyntisiä, suorittavat tehtäviä limittäin siten, että tietyn suorituksen keskeytyessä toisen suoritus voi heti alkaa tai jatkua. *Call by Reference Node* -solmua käytettäessä vapaakäyntisyys saadaan aikaan helposti kuvakkeen ominaisuuksia säätämällä *(propeties)*. *VI properties* -valikon *Execution* -kohdasta valitaan *Reentrant execution.*

Tietovuoparadigman mukaisissa ohjelmissa ei aina voi tietää ohjelman todellista suoritusjärjestystä. Pikalajittelussa on kuitenkin tärkeää, että ensimmäinen rekursiokutsu on päättynyt ennen toisen kutsun suorittamista. Esimerkkitapauksessa suoritusjärjestys on määritelty siten, ettei toinen rekursiokutsu voi alkaa ennen ensimmäisen kutsun päättymistä. LabVIEW -solmun suoritus voi alkaa vasta, kun sen kaikki sisään tuleva tieto on saatu. Langoittamalla ensimmäisen rekursiokutsun tulos toisen rekursiokutsun lähtöparametriksi, voidaan suoritusjärjestys määrätä.

Vi Server -funktioiden käyttäminen rekursion toteuttamiseen LabVIEW -ohjelmissa on raskas ja resursseja vievä tapahtuma. Joka kerta kun kutsutaan kyseisen menetelmän avulla virtuaali-instrumenttia, siihen ei vain luoda viitettä vaan se myös la-



Kuva 11: VI Server pikalajittelun kuvakkeiden esitykset

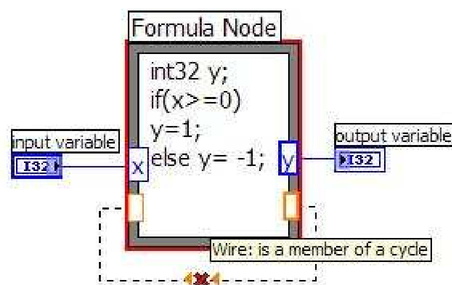
dataan kokonaisuudessaan muistiin. Ohjelmointikielet, jotka tukevat rekursiota, tarvitsevat parhaimmillaan vain muistitilan tarvittaville paikallisille muuttujille. Tämä johtuu siitä, ettei itse funktiota tai proseduuria uudestaan avata viemään muistitilaa, vaan siihen luodaan pelkkä osoitin.

Kuvassa 11 on vielä esitetty pikalajittelu esimerkissä tarvittavat VI Server solmut ja niihin tuotava ja niistä lähtevä tieto

4.3 LabVIEW ja tekstuaalinen ohjelmakoodi

LabVIEW -ohjelmointiympäristöön on liitetty mekanismeja, joilla pystytään liittänään ulkopuolista koodia sen tuottamiin virtuaali-instrumentteihin. Ulkopuolisella koodilla tarkoitetaan lähinnä ei-visuaalisilla menetelmillä eli tekstuaalisilla ohjelmointikielillä tuotettua toiminnallisuutta. LabVIEW -ympäristössä on myös kuvakesolmuja, joihin voidaan kirjoittaa tekstuaalisesti toiminnallisuutta. Esimerkiksi kaavainsolmuun *(Formula Node)* voidaan kirjoittaa C-kielen kaltaisella syntaksilla ohjelmakoodia. Se on tarkoitettu lähinnä matemaattisten laskutoimitusten ja kaavojen tuottamiseen, mutta sen avulla voidaa saada aikaan myös muuta toiminnallisuutta. Rekursion toteuttamiseen kaavainsolmu ei kuitenkaan käy. Vaikka siihen on sisäänrakennettu joukko funktioita, joita voidaan kutsua, ei sillä pystytä rakenta-

maan uusia nimettyjä funktioita. Näin kaavainsolmun sisään kirjoitettua toimintaa ei voida kutsua rekursiivisesti. Koska kaavainsolmu on myös LabVIEW -kuvake, ei se salli langoittamista itseensä. Se antaa virheilmoituksena tiedon siitä, että langoitus muodostaa syklin *(Wire: is a member of a cycle)* kuten kuvasta 12 voidaan todeta.



Kuva 12: Kaavainkuvake

Rekursion toteuttamiseen LabVIEW -ympäristössä tekstuaalisin menetelmin on siis käytettävä muita menetelmiä. Käytettävissä on kaksi hyvinkin saman kaltaista tapaa liittää ulkopuolista ohjelmakoodia virtuaali-instrumentteihin. Helpommin näistä on *Call Library Function Node* -solmun avulla tehtävä linkitys tekstuaaliseen ohjelmakoodiin. Toinen tapa on *Code Interface Node (CIN)* -solmun käyttäminen. Se on LabVIEW -ohjekirjojen mukaan vaikeampi menetelmä ja vaatii kokemusta C-kielen ohjelmistokehityksestä. Molemmat menetelmät tuottavat yleensä yhtä nopean lopputuloksen.

Call Library Function Node ja *CIN* ovat molemmat kaaviodiagrammin objekteja *(block diagram object)*, joiden avulla voidaan tekstuaalisella ohjelmointikielellä tuotettu lähdekoodi linkittää käytettäväksi LabVIEW -ympäristössä. Ne ovat siis kaaviodiagrammin kuvakkeita, joilla on sisään- ja ulostuloterminaaleja. Näiden menetelmien toteuttamistapa on erilainen riippuen käytettävästä käyttöjärjestelmästä. Tässä tutkielmassa keskitymme kuvaamaan ainostaan Windows -ympäristössä tapahtuvaa toimintaa. *Call Library Fuction Node* käyttää hyväkseen joko olemassaolevia tai ohjelmoinnin yhteydessä luotavaa yhteistä kirjastoa *(shared libraries)*. Windows käyttää niistä nimeä *Dynamic Link Libraries (DLL)*. *CIN* -solmuun ulkopuolinen ohjelmakoodi liitetään huomattavasti monimutkaisemmalla tavalla. Tuloksena on kuitenkin täysin LabVIEW -yhteensopiva kuvake, joka voidaan liittää suoraan käytettäväksi virtuaali-instrumentissa ilman tarvittavan yhteisen kirjaston

läsnäoloa. Sen siirrettävyys on siis parempi kuin *Call Library Function Node* -solmun avulla luodun kuvakkeen.

Edellä mainittujen menetelmien käyttäminen LabVIEW:n virtuaali-instrumenteissa sisältää seuraavat neljä askelta [Nat03b]:

- Tekstuaalinen lähdekoodi käännetään, jonka jälkeen sen ajettava muoto on LabVIEW-ympäristön käytössä.
- Kun *Call Library Function Node* tai *CIN* aktivoituvat, LabVIEW kutsuu ajettavaa ohjelmaa.
- LabVIEW välittää syötetiedot kaaviodiagrammin kautta ajettavalle ohjelmalle.
- Ajettava ohjelma palauttaa kaaviodiagrammin kautta paluuarvot LabVIEW virtuaali-instrumenttiin.

Useimmissa ohjelmointitapauksissa LabVIEW:n oma kääntäjä pystyy muodostamaan koodia tarpeeksi nopeasti ilman, että esiintyy ongelmia edes aikakriittisissä tehtävissä. Niin kuin aiemmin on mainittu, normaalisti LabVIEW -ohjelma ajetaan yhdessä ja samassa säikeessä, jossa käyttöliittymä ja suoritettavat tehtävät vuorottelevat. *CIN* ja yhteisten kijastojen käyttö suoritetaan synkronoidusti *(execute synchronously)*, joten LabVIEW ei voi käyttää samaa suoritussäiettä muihin tehtäviin. Rekursion kannalta tällä ei ole *Call Library Function Node-* ja *CIN* -elementtien yhteydessä merkitystä. Rekursiivinen toiminnallisuus rakennetaan niihin sisäisesti, joten rekursiokutsut ovat kokoajan niiden halussa, eikä LabVIEW:n tarvitse puuttua rekursion suoritukseen. Kyseiset elementit voidaan myös monisäikeistää. Tämä on tarpeen varsinkin silloin, kun halutaan LabVIEW -ohjelman tekevän myös muita tehtäviä. *Call Library Fynction Node* menetelmää käytettäessä vapaakäyntisyys saadaan aikaan helposti kuvakkeen ominaisuuksia säätämällä *(configure)*. *CIN* menetelmässä ominaisuuden päällekytkeminen täytyy kirjoittaa itse lähdekoodiin.

Seuraavissa alaluvuissa käsitellään ensin *Call Library Function Node* menetelmää ja sen avulla luotua pikalajittelua. Tämän jälkeen esitellään *Code Interface Node* menetelmä pikalajitteluineen.

4.3.1 Call Library Function Node -pikalajittelu

Call Library Function Node -kuvakkeen avulla voidaan käyttää ja kutsua melkein kaikkia standardeja *DDL* -kirjastoja. Kuvake sisältää laajan joukon tietotyyppejä ja kutsumenettelyjä *(calling conventions)*. Se voi kutsua funktioita myös itsetehdyistä *DLL* -kirjastoista. Yhteisten kirjastojen käytöstä on seuraavia etuja [Nat03b]:

- DDL -kirjasto voidaan vaihtaa toiseen ilman, että sillä on vaikutusta virtuaali-instrumenttiin, joka on siihen liitetty. Toki funktion prototyypin *(function prototype)* täytyy pysyä muuttumattomana.
- Käytännössä kaikki uudet ohjelmistokehitysvälineet tarjoavat tuen DDL -kirjastojen luomiseen. Sen käyttäminen on siis kehitysympäristöstä riippumatonta.

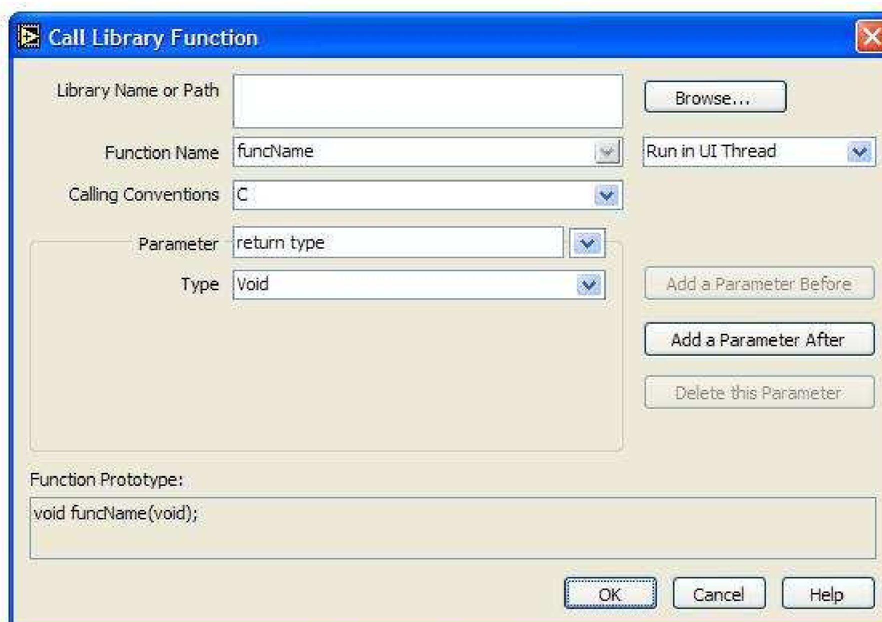
Valmiin LabVIEW -ohjelman siirrettävyys sitävastoin ei ole paras mahdollinen, jos käytetään muita kuin Windowsin standardikirjastoja. Itse ohjelman lisäksi on siirrettävä myös siihen liitetty *DLL*.

Tekstuaalisen ohjelmakoodin liittäminen LabVIEW -ohjelmaan *Call Library Function Node* -kuvakkeen avulla sisältää seuraavat päävaiheet [Nat03b]:

- Vaihe 1: Rakennetaan funktion prototyyppi ja luodaan sen pohjalta .c -tiedosto LabVIEW -ohjelmointiympäristössä.
- Vaihe 2: Lisätään tarvittava toiminnallisuus .c -tiedostoon.
- Vaihe 3: Rakennetaan C-kielen tunnistavalla ohjelmistokehitysvälineellä kirjastoprojekti *DLL* -kirjaston luomiseksi.
- Vaihe 4: Liitetään luotu *DLL* -kirjasto *Call Library Function Node* -kuvakkeen käytettäväksi.

Kun *Call Library Function Node* -kuvake luodaan kaaviodiagrammiin, ei se sisällä mitään toiminnallisuutta eikä siinä ole tarvittavia liitinpareja. Avaamalla kuvakkeen valikko ja valitsemalla konfiguroi *(Configure...)* -vaihtoehto, aukeaa kuvan 13 mukainen ikkuna. Sen avulla voidaan rakentaa funktion prototyyppi. *Function Name* -kohtaan kirjoitetaan luotavan funktion nimi ja samalla huomataan, kuinka ikkunan

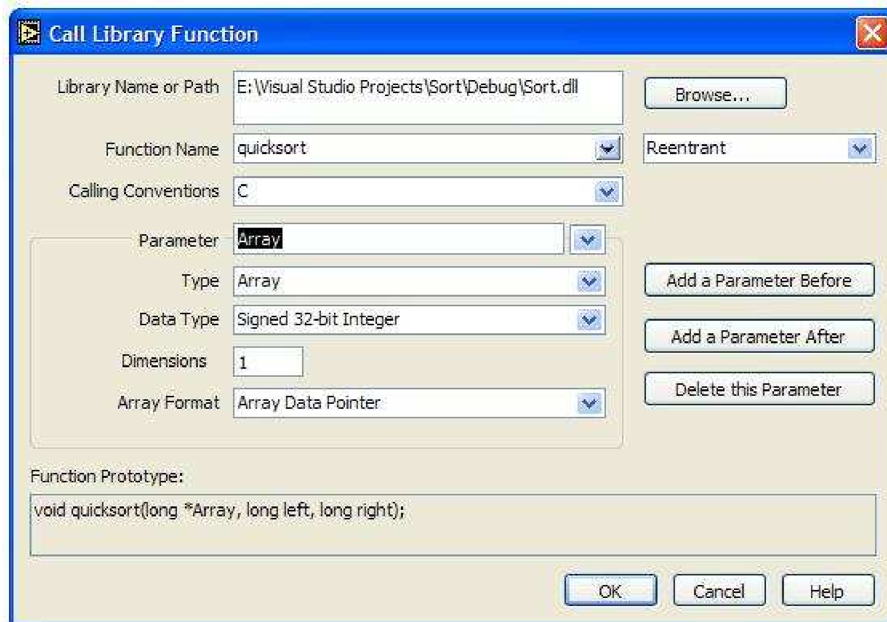
alalaidassa olevassa *Function Prototype* -laatikossa funktion nimi vaihtuu. Koska lähdekoodi on C-kielellä jätetään *Calling Conventions* -kohta oletusarvoonsa. Tämän jälkeen lisätään funktioon sen tarvitsemat parametrit *Add a Parameter After* -painikken avulla. Jokaiselle parametrille annetaan vähintään nimi ja tyyppi *(Type)*. Valitusta tyypistä riippuen annetaan vielä lisämääreitä, jotka ilmaantuvat valittavaksi sen jälkeen, kun tyyppi on määritelty. Esimerkiksi, jos valitaan parametrin tyypiksi numeerinen *(Numeric)*, pitää sille antaa varsinainen tietotyyppi *(Data Type)*, kuten 32-bit Integer, 8-byte Double jne. Numeeriselle tyypille valitaan myös se tapa, miten se välitetään*(Pass)* eli välitetäänkö itse arvo vai osoitin siihen.



Kuva 13: Funktion prototyypin luominen

Funktion prototyypin määrittelyllä luodaan ajettavan ohjelman päämetodi, joten se on aina void -tyyppinen. Kuvassa 14 on valmis pikalajittelualgoritmin prototyyppi. Siinä on vielä vaihdettu funktion suoritus vapaakäyntiseksi käyttöliittymäsäikeen *(Run in UI Thread)* sijasta. Siinä on myös nähtävissä taulukkoparametrin tarvitsemat tiedot. Tässä vaiheessa jätetään kirjaston nimi tai polku *(Library Name or Path)* -kohta tyhjäksi, koska sitä ei ole vielä luotu. Hyväksymällä luodun funktion prototyypin ilmestyy *Call Library Funktion Code* -kuvakkeeseen prototyypissä määritellyt parametrit liitinpareina.

Seuraavaksi avataan uudestaan kuvakkeen valikko ja valitaan *Create .c File...* -

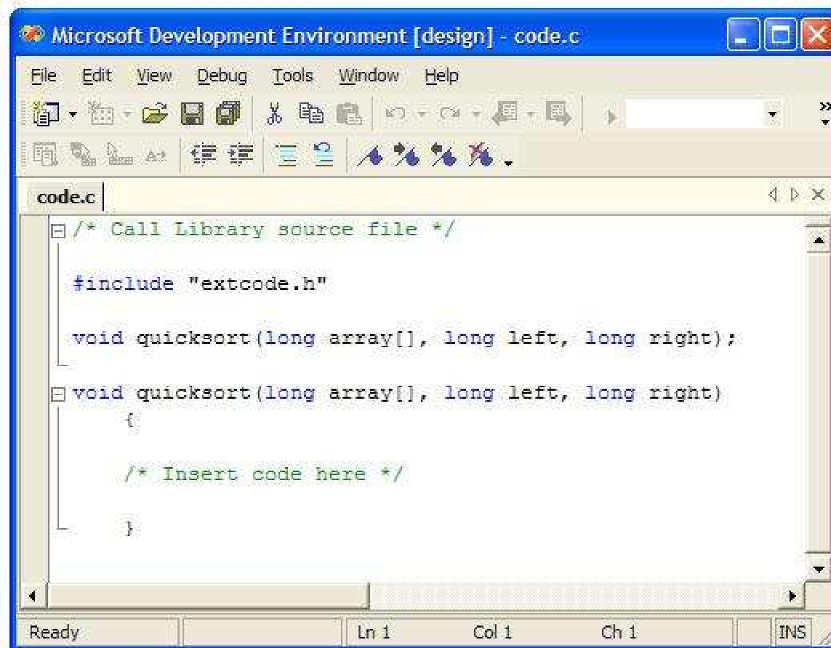


Kuva 14: Pikalajittelun prototyyppi

vaihtoehto ja talletetaan luotu lähdekoodipohja. LabVIEW luo funktion prototyypin perusteella kuvan 15 mukaisen C-kielisen pohjan, johon toiminnallisuus kirjoitetaan. Koodiin liitetty *extcode.h* -otsikkotiedosto *(Header File)* on tarpeellinen silloin, kun halutaan tai joudutaan käyttämään LabVIEW -ympäristön omia hallintafunktioita *(Manager Functions)*. Näillä funktioilla voidaan mm. muuttaa matalantason tavumuunnokset rutiineiksi, joilla lajitellaan tietoalkoita, hallinnoidaan muistin käyttöä jne.

Jos käytetään hallintafunktioita, on *DLL* -kirjastoprojektin luomisen yhteydessä muistettava liittää yhteinen kirjasto mukaan. Käytettävä kirjasto riippuu ohjelmistokehitysvälineestä. Microsoftin kehitysvälineitä käytettäessä se on *labview.lib* ja se löytyy LabVIEW *cintools* -hakemistosta.

Kuvassa 16 on *DLL* -projektikelpoinen ohjelmakoodi pikalajitelualgoritmista C-kielellä. Kun tarkastellaan lähdekoodia huomataan, että jakoalkion määrääminen *(etsipivot)* ja ositusalgoritmi *(partition)* ovat omina funktioinaan ja sijaitsevat omissa tiedostoissaan. *Call Library Function Node* -kuvakkeen avulla voidaan luoda vain päämetodin prototyyppi. Jos ohjelmassa on muita funktioita, on myös niistä luotava prototyypit. Tämä tapahtuu otsikkotiedoston avulla. Jokaisesta ohjelmassa käytettävästä aliohjelmasta luodaan otsikkotiedosto, jossa kyseinen funktio esitellään. Nä-



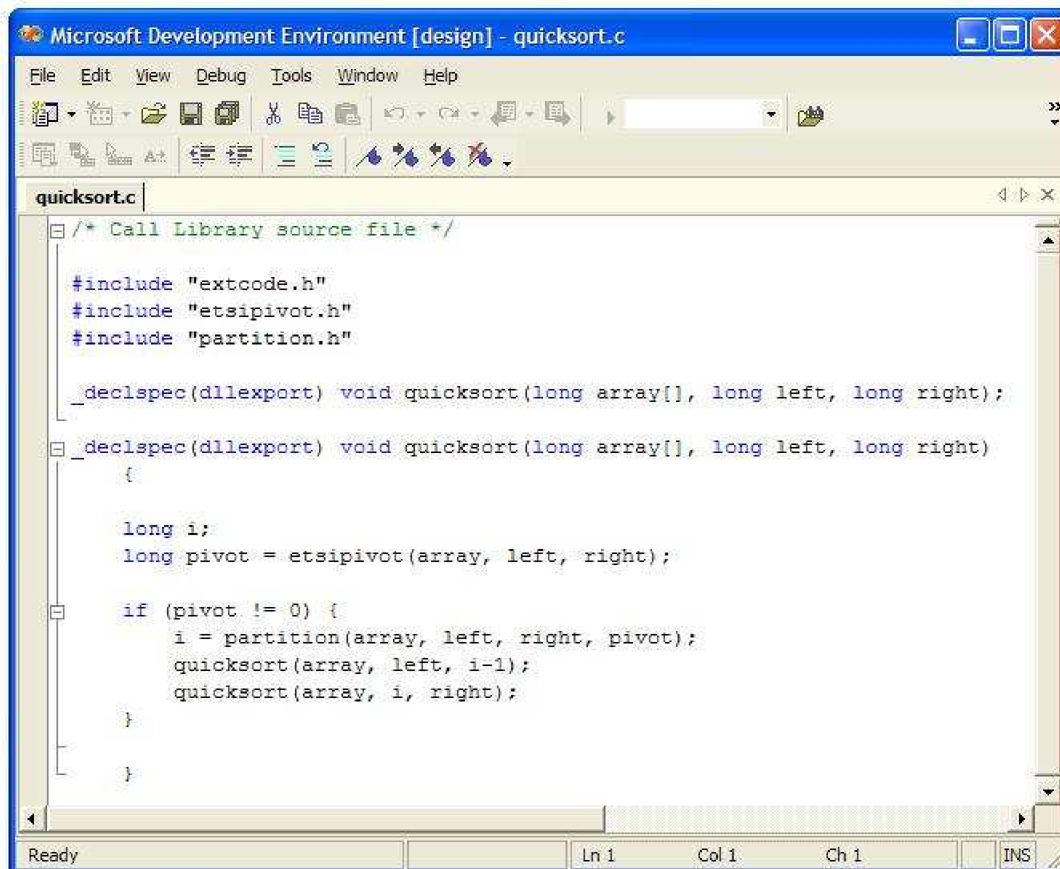
Kuva 15: LabVIEW luoma lähdekoodipohja

mä tiedostot liitetään *include* -määreellä pääohjelmaan. Tämän lisäksi aliohjelmista tehdään pääohjelman kaltainen .c -tiedosto, jossa sen toiminnallisuus määritellään.

Rakennettaessa *DDL* -kirjastoa on jokainen käytettävä funktio esiteltävä _ *decls-pec(dllexport)* -avainsanan *(keyword)* avulla. Avainsanaa käytetään lähdetiedostoissa. Otsikkotiedostoissa sitä ei vielä tarvitse olla, koska niissä esitellään vasta funktion prototyyppi.

Tässä yhteydessä ei käydä läpi *DLL* -kirjaston varsinaista luomista, koska jokaisella ohjelmistokehitysvälineellä on siihen oma tapansa. Todetaan vain, että LabVIEW -kehitysvälineen mukana tulee sähköisessä muodossa *Using External Code in LabVIEW* -ohjekirja. Siinä on hyvät ohjeet kyseisen projektin luomiseen *Microsoft Visual Studio C++ 6.0* -ympäristössä. Ohjeita voidaan soveltaen käyttää myös Microsoftin uudempien sekä muiden valmistajien ohjelmistokehitysvälineiden yhteydessä.

Kun on saatu aikaan tarvittava *DLL* -kirjasto, on se vielä liitettävä *Call Library Function Node* -kuvakkeen käytettäväksi. Tämä tapahtuu avaamalla kuvakkeen valikosta konfigurointi ikkuna ja lisäämällä *DLL* -kirjaston sijaintipolku ja nimi sille varattuun tilaan. Kuvake on nyt valmis käytettäväksi LabVIEW -ohjelmissa normaalin kuvakkeen tavoin ja se sisältää sen toiminnallisuuden, joka siihen ohjelmoitiin.

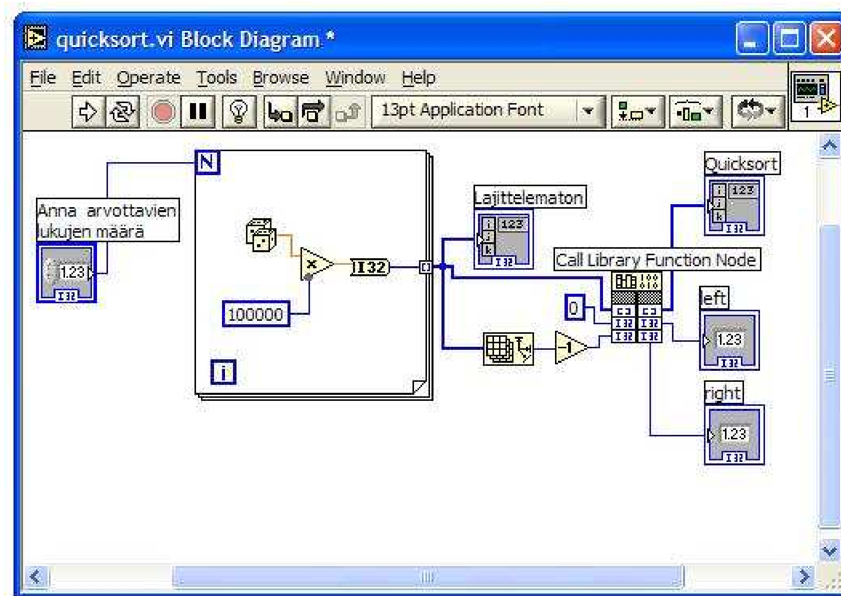


Kuva 16: DLL projektikelpoinen pikalajittelualgoritmi

Koska ohjelmoimme rekursiivisen pikalajittelualgoritmin, on itse kuvakkeen sisäinen toiminta rekursiivista. Ulkoisesti se käyttäytyy samojen sääntöjen mukaan kuin muutkin LabVIEW -kuvakkeet. Sitä ei voi langoittaa itseensä, eikä se voi olla graafin jäsen, joka muodostaa syklin.

Kuvassa 17 on rekursiivisen pikalajittelualgoritmin sisältävä *Call Library Function Node* käytössä. Ohjelmalle annetaan ensin alkioiden määrä, jotka halutaan lajitella. Sen jälkeen ohjelma arpoo annetun määrän alkioita arvoväliltä 0 - 100.000 ja tulostaa nämä alkiot sisältävän taulukon. *Call Library Function Node* saa syöteparametreinaan lajittelemattoman taulukon, vakion nolla (index 0) ja taulukon viimeisen indeksin arvon. Tuloksena on lajiteltu taulukko, joka myös tulostetaan näytölle.

Tässä aliluvussa esitellyn pikalajittelualgoritmin *DLL* -projektikelpoiset C-kieliset lähdekoodit löytyvät liitteestä B. Liitteessä on myös aliohjelmien otsikkotiedostojen lähdekoodit.



Kuva 17: Call Library Function Node pikalajitteluohjelmassa

4.3.2 Code Interface Node (CIN) -pikalajittelu

Code Interface Node (CIN) on kaaviodiagrammin kuvake, joka liittää C/C++ -lähdekoodin LabVIEW -ohjelmaan. Teknisesti on mahdollista kirjoittaa liitettävä lähdekoodi myös muilla ohjelmointikiellillä, jos *CIN* -ohjelman aloituskohdat *(entry points)*, kuten *CINRun, CINLoad* jne. on mahdollista esittää "C-kielimäisesti". Kuitenkin *National Instruments* suosittelee tällöin käytettäväksi mieluummin edellä mainittua *DLL* -kirjastojen käyttöön perustuvaa menetelmää.

LabVIEW ei tue sitä, että ulkopuolisesta tekstuaalisesta koodista luodaan erillisiä aliohjelmia ilman yhteisten kirjastojen apua. Jos aliohjelmia halutaan käyttää, on käytettävä edellisessä kappaleessa käytettyä menetelmää. Kun käytetään *CIN* -kuvaketta, koko lähekoodi on kirjoitettava samaan tiedostoon. Ulkopuolinen koodi on käännettävä sellaiseen ajettavaan muotoon, joka soveltuu käytettäväksi juuri sille määritetyllä alustalla. Koska LabVIEW lataa ulkopuolisen koodin samaan muistiavaruuteen kuin pääohjelman, on koodin oltava myös vapaasti sijoitettavissa *(relocatable)*.

Seuraavat neljä yleistä vaihetta tulee suorittaa luotaessa *CIN* -kuvakkeeseen toiminnallisuutta [Nat03b]:

- Vaihe 1: Kuvataan LabVIEW:n avulla se tieto, joka halutaan välitää *CIN* -elementin kautta.
- Vaihe 2: Kirjoitetaan ja käännetään lähdekoodi jollain tuetulla ohjelmointikielellä.
- Vaihe 3: Ajetaan ohjelma, joka muuntaa käännetyn lähdekoodin LabVIEW:n ymmärtämään muotoon.
- Vaihe 4: Ladataan *CIN* LabVIEW:n käyttöön.

Kun virtuaali-instrumentti ajetaan edellä mainittujen vaiheiden jälkeen ja *CIN* suoritetaan, LabVIEW kutsuu *CIN* -elementin kohdekoodia *(object code)* ja välittää siihen langoitetut tiedot. Jos virtuaali-instrumentti tallennetaan sen jälkeen kun koodi on ladattu, LabVIEW tallentaa koodin yhdessä virtuaali-instrumentin kanssa. LabVIEW ei tämän jälkeen enää tarvitse alkuperäistä koodia ajaakseen *CIN* -elementin. Tämä ominaisuus tekee ohjelmasta helposti siirrettävän, koska mitään lähdekoodia tai *DLL* -kirjastoa ei tarvitse erikseen siirtää. *CIN* -elementin kohdekoodi on myös helposti päivitettävissä tai vaihdettavissa.

Seuraavissa luvuissa käydään lyhyesti läpi *CIN* -objektin rakentamisen erityiset vaiheet kuten [Nat03b]:

- Vaihe 1: Luodaan *CIN* -elementtiin syöttö- ja tulosterminaalit.
- Vaihe 2: Langoitetaan syöttö- ja tulosterminaalit.
- Vaihe 3: Luodaan .c -tiedosto
- Vaihe 4: Käännetään *CIN* -elementin lähdekoodi.
- Vaihe 5: Ladataan *CIN* -elementtiin kohdekoodi.

Toisin kuin *Call Library Function Node* -menetelmässä, luodaan *CIN* -elementtiin ensin kuvakkeen tarvitsemat terminaaliparit ja langoitetaan ne syöttökontrolleihin ja tulosindikaattoreihin. Tämä vastaa funktion prototyypin luomista, koska niiden perusteella *CIN* luo .c -tiedoston mallin.

Kun *CIN* tuodaan kaaviodiagrammiin, siinä on valmiina yksi liitinpari. Lisäliittimiä voidaan lisätä vetämällä hiirellä kuvakkeen alareunasta alaspäin. Liitinparit voivat olla joko syöte-tulostepareja tai pelkästään tulosteen tuottavia. Kun LabVIEW

kutsuu *CIN* -elementtiä, se välittää liitetyt parametrit osoittimina *CIN* -elementin kohdekoodille. *CIN* -elementin suorituksen jälkeen LabVIEW välittää arvoviitteen *(value reference)* osoittimena tulosindikaattorille. Kun liitinparit on kytketty LabVIEW olettaa, että *CIN* pystyy käsittelemään dataa ja se voi kopioida syötetiedon jonkin muun solmun käytettäväksi. Jos kaikkia liitinpareja ei langoiteta syöttö- ja tuloskuvakkeilla, LabVIEW taas olettaa ettei *CIN* pysty käsittelemään dataa ja silloin syötetietoa ei voida käyttää muissa ohjelman solmuissa, vaikka *CIN* sinällään toimisi oikein ja antaisi oikean tulosteen. Tämän takia *CIN* -kuvakkeeseen ei saa jättää ylimääräisiä liitinpareja.

Terminaaliparien järjestys kuvakkeessa vastaa samaa parametrien järjestystä kuin kirjoitetussa koodissa. Syöte- ja tuloskuvakkeissa voidaan käyttää kaikkia LabVIEW-ympäristön sallimia tietotyyppejä. Myös monimutkaiset ja hierarkkiset tietorakenteet ovat mahdollisia, kuten taulukko, joka sisältää klusterin, joka taas voi sisältää muita taulukoita tai klustereita.

Kun kaikki liitinparit on langoitettu ja niihin on muutettu sopivat tietorakenteet ja tietotyypit, voidaan .c -tiedosto luoda avaamalla *CIN* -kuvakkeen valikko. Valitaan *Create .c File* -vaihtoehto ja LabVIEW luo valmiin lähdekoodipohjan, josta on esimerkki taulukossa 6.

CIN -objektin lähdekoodiin voidaan liittää kahdeksan eri rutiinia *(routine)*, joista *CINRun* on ainut pakollinen ja jota LabVIEW kutsuu, kun se ajaa *CIN* -objektin. Muut ovat ylläpitoon *(housekeeping)* liittyviä rutiineja. Kuten luvussa 4.3 mainittiin, on *CIN* -objektin lähdekoodiin kirjoitettava vapaakäyntisyys, jos se halutaan ottaa käyttöön, rekursion toiminnan kannalta sitä ei välttämättä tarvita. Saadaksemme vapaakäyntisyyden aikaan ja samalla monisäikeistyksen toimintaan, tarvitaan *CINProperties* rutiinia. Lähdekoodiin on silloin liitettävä taulukossa 7 oleva moduli [Nat03b].

Liittessä C on pikalajittelualgoritmin lähdekoodi *CIN* -elementin vaatimassa muodossa. Kun verrataan sitä liitteessä B oleviin *Call Library Function Node* -objektin lähdekoodeihin on niissä iso ero, vaikka kyseessä on sama algoritmi. *CIN* -objektin lähdekoodi on kirjoitettava suoraan LabVIEW -ympäristön ymmärtämässä muodossa. Siinä on käytettävä osoittimia ja viittauksia, jotka *Call Library Function Node* -menetelmää käytettäessä on automatisoitu ja muutettu yleisempään muotoon yh-

```
/* CIN source file */

#include "extcode.h"

/* Typedefs */

typedef struct {
int32 dimSize;
int32 Numeric[1];
} TD1;
typedef TD1 **TD1Hdl;

MgErr CINRun(TD1Hdl Array, int32 *Numeric, int32 *Numeric2);

MgErr CINRun(TD1Hdl Array, int32 *Numeric, int32 *Numeric2)
{

/* Insert code here */

return noErr;
}
```

Taulukko 6: CIN kuvakkeen rakentama lähdekoodipohja

teisten kirjastojen käytön takia.

Seuraavaksi *CIN* -objektin lähdekoodi on käännettävä LabVIEW:n alirutiini (.lsb) -tiedostoksi *(subroutine)*. Tämä voidaan tehdä ohjelmistokehitysvälineillä tai ajaa joissain LabVIEW -versioissa jopa suoraan komentoriviltä. Ohjeet tähän löytyvät *Using External Code in LabVIEW* -ohjekirjasta. Ohjelmistokehitysvälineitä käytettäessä luodaan niillä samankaltainen *DDL* -projekti kuin *Call Library Funktion Node* -menetelmän yhteydessä. Projektiin on liitettävä LabVIEW:n *cintools* -hakemistosta *labview.lib* -tiedoston lisäksi myös *cin.obj, lvsb.lib* ja *lvsbmain.def* -tiedostot. Oli menetelmä mikä tahansa pääasia on, että prosessin seurauksena muodostuu *.lsb* -muotoinen tiedosto. Luotu alirutiini tuodaan *CIN* -objektin käyttöön valitsemalla sen vetovalikosta *Load Code Resource* -vaihtoehto. Kun virtuaali-instrumentti ajetaan ja sen jälkeen tallennetaan, on *CIN* -kuvake samoilla ehdoilla käytettävissä kuin *Call Library Function Node* -kuvake, se vain on siirrettävämpi, koska *DLL* -kirjastoa ei tarvitse liittää sen mukaan.

```
CIN MgErr CINProperties(int32 prop, void *data)
{
switch (prop) {
case kCINIsReentrant:
*(Bool32 *)data = TRUE;
return noErr;
}
return mgNotSupported;
}
```

Taulukko 7: Vapaakäyntisyyden päällekytkeminen

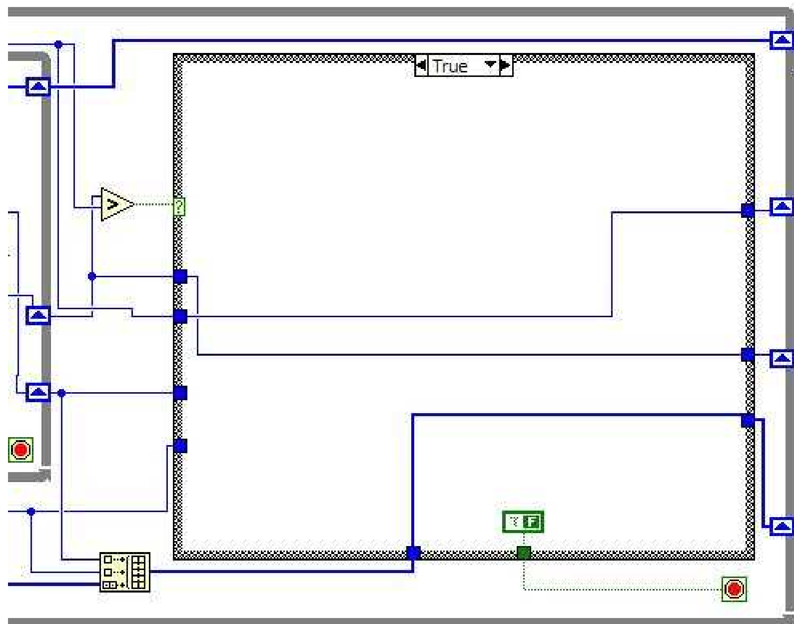
4.4 Pinoperustainen pikalajittelu

Kuten luvuissa 3.2 ja 3.3 on mainittu, voidaan pinorakennetta käyttää hyväksi mallinnettaessa rekursiivisia algoritmeja. LabVIEW ei sisällä valmista pinorakennetta, mutta sitä pystytään mallintamaan joko taulukko- tai jonorakennetta hyväksikäyttäen.

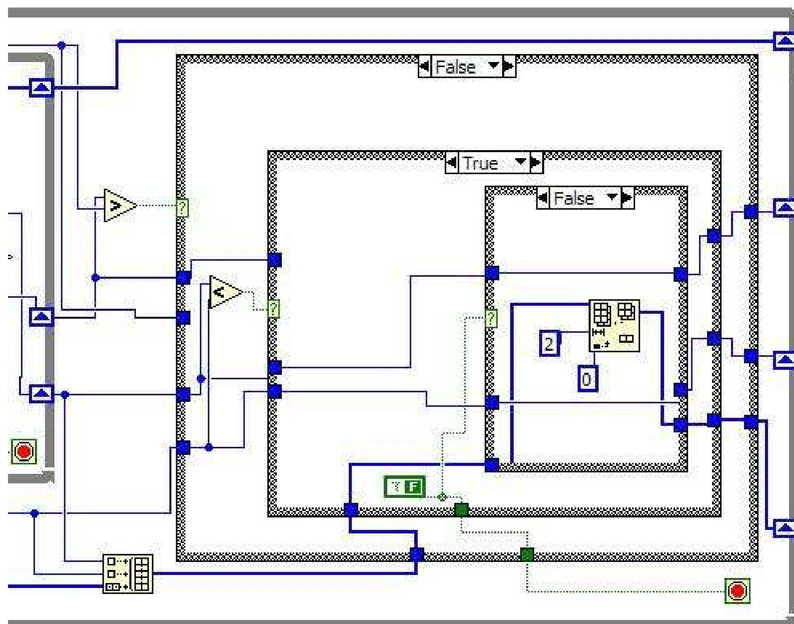
Liitteessä D on pinoon perustuvan pikalajitttelun LabVIEW -ohjelmakoodi. Ratkaisussa on jakoalkion valinta ja ositusalgoritmi toteutettu täsmälleen samalla tavalla, kuin liitteessä A ja luvussa 4.2 esitetyn *VI Server* pikalajittelun ratkaisussakin. Ero on rekursiivisten kutsujen toteuttamisssa ja mallintamisessa.

Kyseisessä ratkaisussa käytetään taulukkoa pinomaisesti tallentamaan rekursion mallintamisessa tarvittavat tiedot. Taulukkoon tuodaan uusi tieto päällimmäiseksi ja siitä poistetaan päällimmäisin tieto. Taulukossa 5 luvussa 4.1 olevan pikalajittelualgoritmin mukaisesti pinoperustainen pikalajittelu tallentaa jokaisella uloimman *while* -silmukan kierroksella parametriä R ja apumuuttujaa i vastaavat tiedot. Apumuuttuja i tallentetaan indeksiin 0 ja parametri R indeksiin 1. Aiemmilla silmukan kierroksilla talletetut arvot siirtyvät pinossa alaspäin.

Kuvissa 18, 19, 20 ja 21 on kuvasarjassa esitetty rekursion mallinnukseen tarvittavat osat. Kuvassa 18 on tila a, joka vastaa rekursiivisen pikalajittelualgoritmin ensimmäistä rekursiokutsua *sort(L, j)*. Siihen mennään, jos lause $L < j$ on tosi. Apumuuttujaa j vastaavan tiedon saamme suoraan ositusalgoritmin tuloksesta ja parametria L vastaavat tiedot syöttökontrolliin *Left* liitetystä siirtorekisteristä. Pinoa mallintava taulukko välitetään myös seuraavalle kierrokselle.

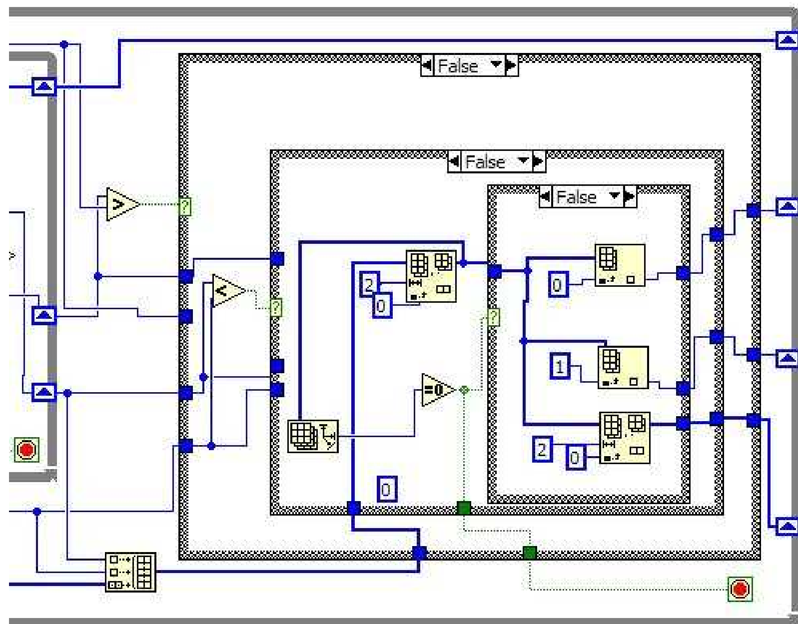


Kuva 18: Pinoperusteisen pikalajittelun rekursion toteutus, tila a

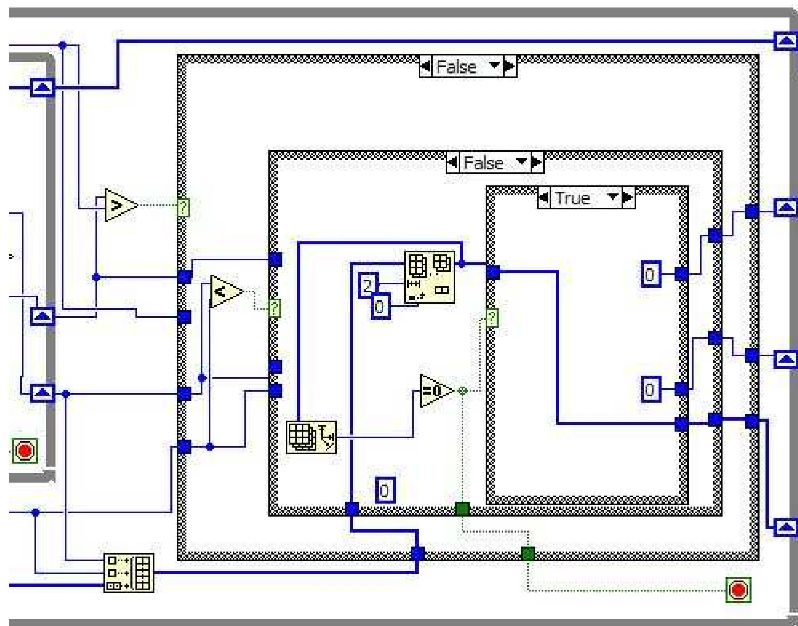


Kuva 19: Pinoperusteisen pikalajittelun rekursion toteutus, tila b

Kuvassa 19 on tila b, jossa $L < j$ on epätosi ja $i < R$ on tosi, jolloin suoritetaan *sort(i, R)*. Tämän toisen rekursiokutsun parametrit saamme myös suoraan kyseisestä silmmukkakierroksesta. Apumuuttujaa i vastaavan tiedon saamme suoraan



Kuva 20: Pinoperusteisen pikalajittelun rekursion toteutus, tila c



Kuva 21: Pinoperusteisen pikalajittelun rekursion toteutus, tila d

ositusalgoritmin tuloksesta ja parametria R vastaavat tiedot syöttökontrolliin *Right* liitetystä siirtorekisteristä. Toki voimme ottaa ne myos taulukon indeksin arvoista 0 ja 1, jonne ne myös on tallennettu. Tärkeää on kutenkin poistaa kyseiset arvot

taulukosta, joka tehdäänkin sisimmässä *case* -rakenteessa.

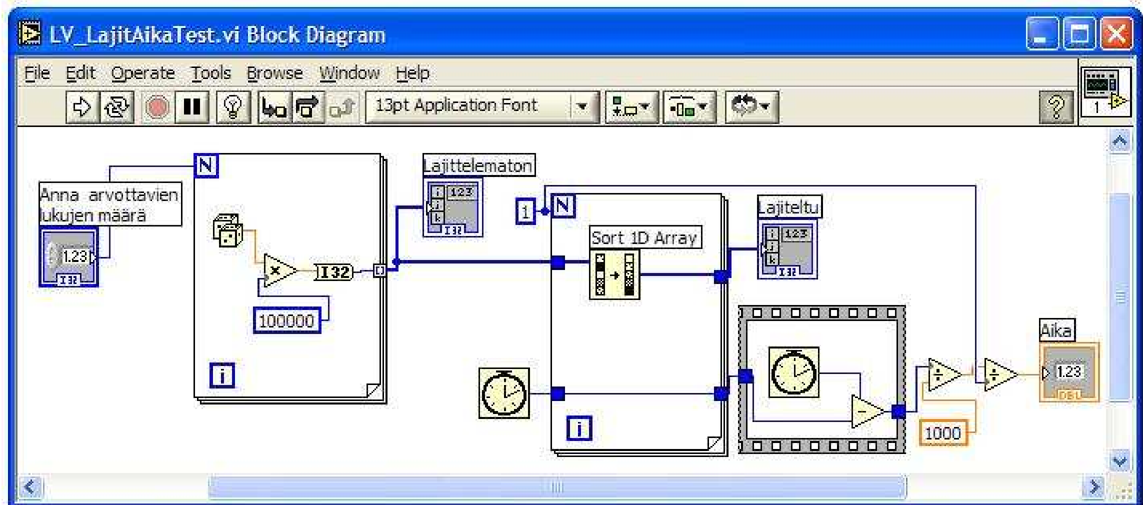
Kuvassa 20 on tila c, jolloin molemmat $L< j$ ja $i < R$ ovat epätosia. Se mallintaa tilannetta, jolloin rekursiossa lähdetään palaamaan takaisin edellisiin rekursiokierroksiin. Keskimmäisessä *case* -rakenteessa ensin poistetaan pinosta sinne tällä *while* -kierroksella sinällään turhaan tallennetut tiedot. Sen jälkeen tutkitaan, onko pino tyhjä. Jos pino ei ole tyhjä, suoritetaan sisimmässä *case* -rakenteessa oleva toiminta. Siinä pinon (taulukon) indeksin 0 arvo välitetään *Left* -siirtorekisteriin ja indeksin 1 arvo *Right* -siirtorekisteriin. Eli palaamme edelliseen suoritukseen ja kyseinen toiminta vastaa *sort(i, R)* suoritusta. Tässä yhteydessä emme tarkista ehtoa $i < R$. Tämä ei vaikuta lajittelun lopputulokseen, sen nopeuteen kylläkin, koska tämän vuoksi suoritetaan turhia *while* -silmukan kierroksia. Lisäksi poistamme pinosta (taulukosta) kaksi päällimmäistä alkiota, jotka juuri välitimme parametreinä uloimmalle *while* -silmukalle.

Viimeisessä kuvassa 21 on tila d, joka kuvaa tilannetta rekursion päättyessä. Siinä lopetetaan ohjelman suoritus ja tulostetaan lajiteltu taulukko. Kyseessä on muutoin edellisen kappaleen mukainen tilanne paitsi pinosta (taulukosta) poistamisen jälkeen pino tulee tyhjäksi, jolloin totuusarvo katkaiseen ohjelman suorituksen.

4.5 Rekursiivisten ratkaisujen nopeustesti

Edellämainituille rekursiivisille pikalajitteluratkaisuille suoritettiin nopeustesti. Verrokki -lajitelualgoritmina käytettiin LabVIEW -ympäristön omaa yksiuloitteisen taulukon lajittelevaa kuvaketta *(Sort 1D Array)*. Kuvassa 22 olevassa LabVIEW -ohjelmassa kyseinen lajittelu on käytössä.

Ohjelmalle annetaan lajiteltavien alkioden määrä, jonka jälkeen se arpoo alkiot arvovälille 0 - 100.000 taulukkoon. Taulukko lajitellaan ja sen lisäksi tulostetaan ohjelman kokonaissuoritusaika. Jokainen edellämainitu pikalajitteluratkaisu sijoitettiin samanlaiseen arvonta- ja aikasilmukkaan ja näin on mitattu lajittelunopeus. Vaikka ohjelma ei mittaa pelkästään lajittelun nopeutta, vaan mukana on myös alkioiden arvontaan kuluva aika, ei sillä ole tässä tapauksessa merkitystä. Koska alkioiden arpomiseen kuluvaa aikaa voidaan pitää vakiona testitapauksissa, joissa niitä arvotaan sama määrä, ei tällä ajalla ole vaikutusta vertailtaessa ratkaisujen lajittelunopeuksia



Kuva 22: Arvotun taulukon lajittelu LabVIEW -ympäristön omalla lajittelualgoritmilla

toisiinsa.

Testialustana toimi Hp Compaq nx7010 kannettava tietokone Intel Centrino 1.6 GHz -prosessorilla ja 512 megatavun RAM -muistilla. Käyttöjärjestelmänä oli Windows XP Professional ver. 2002 Service Pack 2:lla täydennettynä. Pikalajitteluratkaisut ohjelmoitiin LabVIEW ohjelmistokehitysvälineen versiolla 7.1.

Jokainen rekursiivinen pikalajitteluratkaisu testattiin 10.000, 100.000 ja 1.000.000 alkiolla sekä 10.000.000 alkiolla niissä tapauksissa, joissa se oli mahdollista. Kuvan 23 taulukossa esitetään jokaisen ratkaisun keskiarvolajittelunopeus sekunneissa kullakin alkiomäärällä. Keskiarvot on laskettu kunkin osalta kymmenen testiajon perusteella, paitsi Vi Server -lajittelunopeus on 1.000.000 alkion tapauksessa kolmen testiajokerran keskiarvo.

Lukumäärä \ lajittelu	LV -lajittelu	VI Server -lajittelu	DLL -lajittelu	CIN -lajittelu	Pinolajittelu 1	Pinolajittelu 2
10.000	0,029 s	20,29 s	0,041 s	0,045 s	0,17 s	0,16 s
100.000	0,25 s	400,78 s	0,27 s	0,24 s	0,93 s	0,83 s
1.000.000	1,33 s	26371,4 s	0,89 s	0,94 s	8,90 s	8,45 s
10.000.000	12,8 s		6,85 s	7,16 s	104,24 s	100,76 s

Kuva 23: Rekursiivistenratkaisujen nopeudet eri alkiomäärillä

Pinoon perustuvasta pikalajittelusta oli testisssä mukana kaksi eri versiota. Edellä luvussa 4.4 kuvatun ratkaisun *(pinolajittelu 1)* lisäksi siitä tehtiin toinen versio *(pi-*

nolajittelu 2). Siinä pinoon tallennetaan parametriä R ja apumuuttujaa i vastaavat arvot vain siinä tapauksessa kun $i < R$. Tämä vastaa toisen rekursiokutsun *sort(i, R)* esiehdon tarkistusta ja vähentää toistosilmukan suoritusmääriä. Liiteessä E on tämän rekursiivisen pinoon perustuvan pikalajittelun LabVIEW -koodi.

Testin tuloksista voidaan todeta se, että eri menetelmien erot samoilla alkiomäärillä ovat huomattavat. Kun 1.000.000 alkion lajittelu pahimmillaan kestää yli 26.371 sekuntia, on lajittelun nopeus samalla alkiomäärällä parhaimmillaan vain 0,89 sekúntia.

Toisena nousee esille se huomio, että *VI Server* -lajittelu käyttäytyy eri tavoin kuin muut menetelmät. Kun muiden osalta voidaan huomata lajitteluajan lineaarinenkasvu alkiomäärien suhteen, kasvaa *VI Server* -lajittelun aika lähes ekspotentiaalisesti. Kuten kuvasta 23 voidaan huomata ei se sisällä *VI Server* -lajittelun tuloksia suurimmalla alkiomäärällä. Kun kyseistä menetelmää testattiin miljoonalla alkiolla jätettiin testikone työskentelemään yön yli. Vasta yli seitsemän ja puolen tunnin jälkeen lajittelu oli saatu päätökseen. Koska testikonetta ei voinut jättää pelkästään testauskäyttöön ei voitu edes ajatella kyseisen menetelmän testaamista 10.000.000 alkiolla. Kahdella pienimmällä alkiomäärällä suoritetun testin perusteella voidaan huomata lajitteluajan ekspotentiaalinen kasvu. Miljoonalla alkiolla lajitteluaika ei kuitenkaan kasva enää ekstonentiaalisesti, koska 7,5 tunnin sijasta lajitteluajan olisi pitänyt olla yli 44 tuntia. Joka tapauksessa *VI Server* -lajittelun aika ei kasva lineaarisesti alkiomäärien suhteen ja lopullista aikaa 10.000.000 alkion osalta voidaan vain arvailla.

Aktivaatiotietuetta mallintavista pinoon perustuvista lajitteluista haluttiin myös testata se, miten paljon lajittelunopeuteen vaikuttaa toisen rekursiivisen kutsun esiehdon poisjättäminen. Kokonaisuudessaan esiehdon poisjättämisen vaikutus on pinolajittelun yhteydessä yllättävän pieni verrattuna sen kokonaissuoritusaikaan. Kun rekursiivisia toistoja 1000 alkiolla suoritetaan pinolajittelu 1:n (ei tarkistusta) tapauksessa keskimäärin 1006 kertaa, niin pinolajittelu 2:n (tarkistus) kohdalla toistoja on keskimäärin 887 kertaa. Vaikka toistokierrosten määrä ratkaisujen välillä on suuri, ei tällä ole kuitenkaan suurta merkitystä lajittelujen kokonaisaikaan. Tästä voidaan tehdä myös se johtopäätös, että pinolajittelun hitaus verrattuna verrokkilajitteluun *(LV -lajittelu)* ja ulkopuolista koodia käyttäviin lajitteluihin *(DLL -lajittelu, CIN -lajittelu)* täytyy johtua jostain muusta syystä kuin toiston hitaudes-

ta.

Call Library Function Code (DLL -lajittelu) ja *Code Interface Node (CIN -lajittelu)* olivat testin selvästi nopeimmat ratkaisut, vaikka verrokkilajittelu ei niistä paljon jälkeen jäänytkään. Lajittelunopeus näillä kahdella menetelmällä on käytännössä yhtä suuri. Jostain syystä *DLL* -lajittelu oli kuitenkin joka suorituskerralla aavistuksen verran nopeampi kuin *CIN -lajittelu*. Olisi voinut luulla asian olleen toisinpäin, koska *CIN -lajittelu* on integroidumpi ratkaisu kuin *DLL -lajittelu*.

Kovin pitkällemeneviä johtopäätöksiä ei tämän pienen testin johdosta voi tietenkään tehdä. Etenkin kun pikalajitteluagoritmin rakenne poikkeaa jakoalkion ja ositusalgoritmin osalta DLL- ja CIN -lajitteluissa muista ratkaisuista. Yhteenvetona voidaan kuitenkin todeta, että LabVIEW-ympäristön oma lajitteluinstrumentti on hyvin käyttökelpoinen useimmissa lajittelutapauksissa. Se lajittelee kuitenkin vain yksiulotteisia taulukoita, eikä LabVIEW:n perusasennuspaketista löydy instrumenttia muiden tietorakenteiden lajitteluun. Rekursiivisten ratkaisuiden osalta tuli varsin selväksi, ettei *VI Server* -elementtejä kannata käyttää rekursion toteuttamiseen. Pinoon perustuvaa rekursiota voidaan käyttää silloin, kun rekursiokutsujenmäärä ei ole liian suuri. Jos haluamme optimoida koodia eikä LabVIEW-ympäristöstä löydy tai siinä ei ole tarpeeksi tehokasta instrumenttia, kannattaa ulkopuolisen koodin liittäminen virtuaali-instrumentteihin.

5 POHDINTA

Kun lähdin etsimään lähdemateriaalia tätä tutkielmaa varten, selvisi varsin pian, että visuaalisiin kieliin suuntautunut yleinen tutkimus on vähentynyt 2000- luvulle tultaessa. 1980- ja 1990 -luvuilta löytyy paljonkin tähän liittyvää tutkimusta, mutta tänä päivänä se on suuntautunut yleisestä kohti kielen tai ohjelmointiympäristön jonkin erityisalueen tutkimukseen. Tästä huolimatta rekursiota ja visuaalista ohjelmointia käsitellään vain harvassa tutkimuksessa. LabVIEW:n osalta en löytänyt ainuttakaan kohdealueelleni tehtyä tieteellistä tutkimusta, ainostaa sivulauseenomaisia mainintoja asiayhteyteen.

LabVIEW on mielestäni onnistunut ainakin yhdessä visuaalisen kielen käytön tarkoitusperässä. Se on helppo oppia ja jo vähäisillä ohjelmointitaidoilla voi insinööri rakentaa sillä tarvitsemiaan virtuaalimittareita. Toki LabVIEW:n kaikkien mahdollisuuksien ja tietovuo-ohjelmoinnin sisäistämiseen tarvitaan perehtyneisyyttä. LabVIEW:n visuaalinen lähdekoodi on myös tiiviimpää kuin tekstuaalinen lähdekoodi. Usein koko ohjelman koodin voi nähdä samaan aikaan näyttöruudussa, kun saman esittäminen tekstuaalisesti voi vaatia pitkänkin listauksen. LabVIEW-ohjelman kaikki tilat eivät sen sijaan näy samaan aikaan. Esimerkiksi case-rakenteesta on nähtävissä vain yksi vaihtoehto kerrallaan. Jos halutaan julkaista lähdekoodi kokonaisuudessaan painettuna, on sellainen tehtävä LabVIEW-ohjelman jokaisesta eri tilasta. Tämän tutkielman liitteenä olevat painetut LabVIEW-ohjelmakoodit ovat vain sen yhdestä tilasta. Täydelliset lähdekoodit löytyvät tutkielman liitteenä toimitetusta CD-ROM levykkeestä.

Kun lähdin tekemään tätä tutkielmaa, ei minulla ollut aikaisempaa kokemusta LabVIEW-ohjelmoinnista ja tietovuo-ohjelmointiparadigmasta tiesin vain pääperiaatteen. Tästä johtuen on tutkielman LabVIEW-esimerkit ohjelmoitu hyvin pitkälti traditionaalisen ohjelmoinnin ajattelumallin mukaisesti. Tosin pikalajittelu ratkaisuissa halusin tietoisesti niiden muistuttavan vastaavaa tekstuaalista ohjelmointikoodia. Tämä sen takia, että visuaalista koodia voitaisiin helpommin verrata ja seurata yhdessä tekstuaalisen koodin kanssa.

Valtaosa tutkielmassa olevista ohjelmointiesimerkeistä ohjelmoitiin heti tutkielman teon alussa. Näin niissä näkyy kokematttomuuteni tietovuo-ohjelmoinnista. Uskonkin, että ratkaisuja voidaan parantaa lähestymällä niitä toisesta näkökulmasta. Var-

sinkin pinoperustainen pikalajittelu nopeutuisi varmasti koodin optimoimisella enemmän tietovuoparadigman mukaiseksi.

Monissa lähteissä mainitaan tietovuoparadigman läheinen suhde funktionaalisiin kieliin. Ne puolestaan perustuvat laskentamalliin, jossa funktioiden rekursiivisilla ominaisuuksilla on suuri merkitys. Tästä syystä on vähän hämmentävää huomata ettei LabVIEW tue rekursiota.

Ohjelmointikielet ovat aina kompromisseja siitä, millä paradigmalla ne on toteutettu. LabVIEW-ympäristökään ei ole sataprosenttisesti tietovuoperiaatteen mukainen eikä se ole täysin visuaalinen kieli, koska siihen voidaan linkittää tekstuaalista ohjelmakoodia. Tämän vuoksi en oikein voi ymmärtää, miksi rekursiiviset ominaisuudet on siitä jätetty pois. Vaikka LabVIEW-ohjelmat pääasiassa toimivat hyvinkin staattisesti, ei dynaamisia ominaisuuksia ole unohdettu silloin kun niihin on katsottu tarvetta. Tullakseen yleiskäyttöiseksi, oikeasti modulaariseksi, ohjelmointikieleksi LabVIEW-ympäristöön olisi mielestäni liitettävä rekursion mahdollistavat ominaisuudet Jeff Kodoskyn vastakkaisista argumenteista huolimatta. Toki yleiskäyttöisyyden saavuttamiseksi tarvitaan muutakin, mutta ohjelmointikielessä on oltava ne perusrakenteet, joita nykyaikaiselta ohjelmointikieleltä vaaditaan.

Millainen voisi rekursiivinen ratkaisu LabVIEW:ssä olla? Visuaaliselta kannalta se voisi olla toistosilmukoiden kaltainen rakenne, jonka sisällä syklisyys on sallittu. Se voisi olla yksinketaisimmillaan myös langoitukseen liitettävä ominaisuus, joka kertoo LabVIEW:lle rekursiosta ja sallii takaisinkytkennän jättäen vastuun ohjelmoijalle.

Lähtökohtaisesti LabVIEW-ympäristössä kannattaa käyttää sen valmiita virtuaali-instrumentteja, kuvakkeita ja rakenteita iteratiivisesti aina kun se vain on mahdollista. Rekursiiviset ratkaisut, ainakin tässä tutkielmassa esiintuodut, ovat joko hitaita tai työläitä ohjelmoida. On kuitenkin tilanteita, jolloin rekursion toteuttamismahdollisuus ja rekursion tuoma hyöty ovat suuremmat. Esimerkiksi tässä tutkielmassa esitellyn pikalajittelun iteratiivinen ratkaisu on monimutkainen ja työläs ohjelmoida LabVIEW-ympäristössä. Myös valmiin lajitteluinstrumentin puuttuminen asennusversiosta muilta tietorakenteilta kuin yksiulotteiselta taulukolta voidaan pitää omituisena ratkaisuna.

CIN- ja DLL -pikalajittelujen nopeus osoittaa mielestäni sen, että kyseisillä menetelmillä päästään kiinni LabVIEW:n G-kielen sisäiseen rakenteeseen. *Call Library*

Function Node ja *Call Interface Node* ovat kielen kehittäjien luomia ja optimoimia rakenteita. Jos niiden avulla on mahdollista ohjelmoida tehokkaita rekursiivisia virtuaali-instrumentteja, miksei niitä voisi olla LabVIEW-ympäristössä valmiina?

Niin hyvät ja laajat kuin National Instrumetsin LabVIEW:tä koskevat internet sivustot ovatkin, en löytänyt sieltä, enkä muualtakaan kunnon vastausta kahteen tämän tutkielman kannalta tärkeään kysymykseen. Ensiksikin kunnon perusteluja sille, miksi rekursiivisia ominaisuuksia ei ole liitetty LabVIEW-ohjelmointiympäristöön? Olisin halunnut myös tietää, mikä lajittelualgoritmi on LabVIEW:n oman lajitteluinstrumentin pohjana ja miten se on toteutettu? Ainakaan minä en etsiskelyistä huolimatta löytänyt näihin kysymyksiin vastauksia.

Tämä tutkielma sai alkunsa tietojenkäsittelytieteen laitoksen kahvihuoneessa käydystä keskustelusta. Kun tiedustelin tulevalta ohjaajaltani mahdollisia tutkielman aiheita, hän sanoi "LabVIEW:ssa rekursio ei ole mahdollista, ota selvää, miten se siinä voidaan toteuttaa". Yhdellä pienellä lauseella on suuri merkitys ohjelmoinnissa, ja näin se voi olla myös meidän ihmisten osalta...

Viitteet

[AHU74] Alfred M. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[Als99] M. H. Alsuwaiyel. *Algorithms: Design Techniques and Analysis*. World Scientific Publishing Co. Pte. Ltd., 1999.

[Aug97] Michael Auguston. Visual data flow language based on iterative constructs. In *Proceedings of the 8 Israeli Confrence on Computer-Based Systems and Software Engineering (ICCSSE'97)*, 1997.

[BD04] Marat Boshernitsan and Michael Downes. Visual programming languages: A survey. Technical report, Computer Science Division,University of California, 2004. http://www.cs.berkeley.edu/~maratb/pubs/csd-04-1368.pdf.

[Bis04] Robert H. Bishop. *Learning with LabVIEW 7 Express*. Pearson Education, Inc., 2004.

[BLP+00] Jorma Boberg, Seppo Lammi, Martti Penttonen, Tapio Salakoski, Tuula Strömberg, and Jukka Teuhola. Johdatus tietojenkäsittelytieteeseen. Turun yliopisto, täydennyskoulutuskeskus, 2000.

[Bur99] Margaret M. Burnett. Visual programming. In John G. Webster, editor, *Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons Inc., 1999. http:/ cespc1.kumoh.ac.kr/~ygkim/vb2/WhatisVP.pdf.

[Fin96] Raphael A. Finkel. *Advanced Programming Language Design*. Addison-Wesley Publishing Company, 1996. ftp://ftp.aw.com/cseng/authors/finkel/apld/ , viitattu 13.02.2007.

[GMK06] Tapio Grönfors and Maija Marttila-Kontio. *Visuaalinen tietovuo-ohjelmointi (VTO)*. Kuopion yliopisto, Tietojenkäsittelytieteenlaitos, luentomoniste, syksy 2006.

[Gup02] Kyle P. Gupton. Implementing recursive algorithms in g. In *LabVIEW Technical Resource*, volume 6, number 2. LTR Publishing, Inc, 2002. http://www.ltrpub.com, viitattu 02.03.2005.

[HM04] Ilkka Haikala and Jukka Märijärvi. *Ohjelmistotuotanto.* Talentum, 2004.

[Kod] Jeff Kodosky. Is labview a general purpose programming language? In *National Instruments Developer Zone.* National Instruments. http://zone.ni.com/devzone/cda/tut/p/id/5313, viitattu 13.02.2007.

[Moo95] Cristopher Moore. Recursion theory on the reals and continuous-time computation. In *Theorical Computer Science. Vol. 162, no. 1, pp. 23-44*, 1995. http://www.santafe.edu/research/publications/workingpapers/95-09-079.ps.

[Nat03a] National Instruments. *LabVIEW 7 Express, User Manual*, 2003.

[Nat03b] National Instruments. *Using External Code in LabVIEW*, 2003.

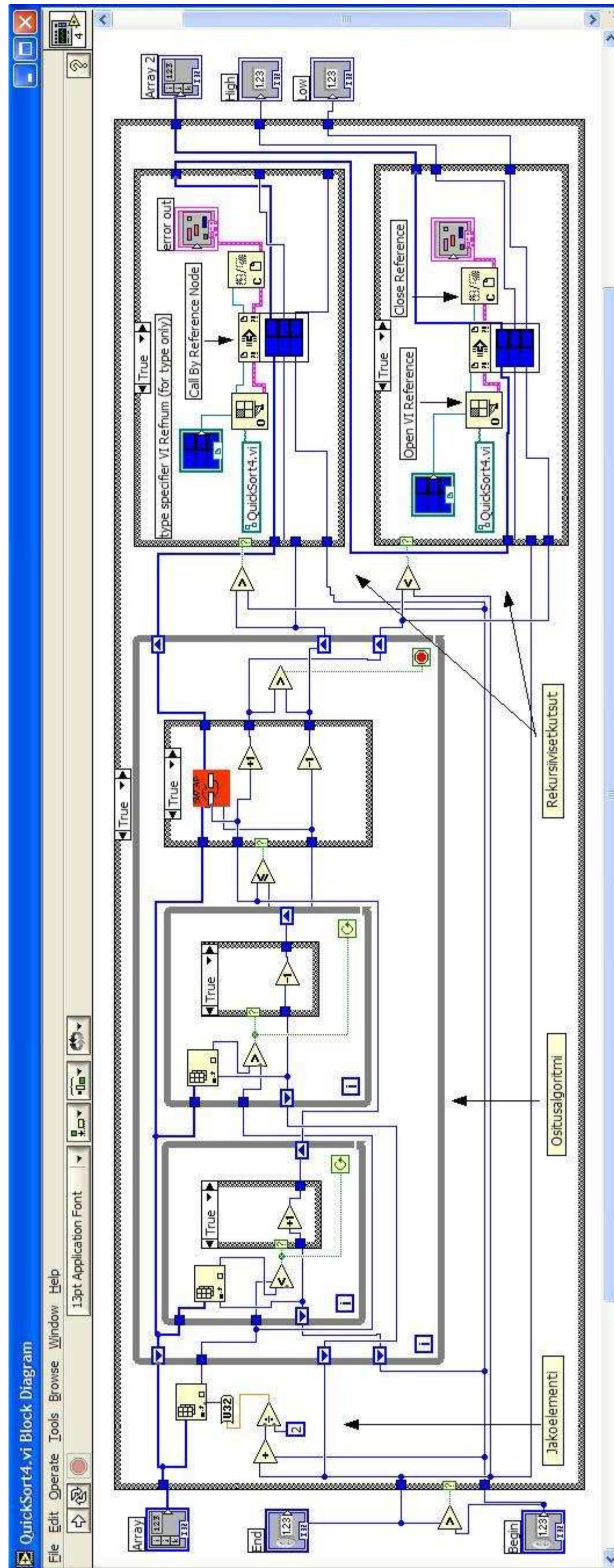
[Nat03c] National Instruments Developer Zone. *Serving Up Powerful Solutions: LabVIEW VI Server Examples*, 2003. http://zone.ni.com/devzone/cda/tut/p/id/4062, viitattu 14.11.2007.

[Nat07] National Instruments. *National Instruments: LabVIEW internet sivut*, 2007. http://www.ni.com/labview/.

[Sco00] Michael L. Scott. *Programming Language Pragmatics.* Morgan Kaufmann Publishers, 2000.

[Wir86] Niklaus Wirth. *Algorithms & Data Structures.* Prentice-Hall, Inc, 1986.

A VI Server pikalajittelun LabVIEW koodi



B Call Library Function Code pikalajittelun lähdekoodit

B.1 etsipivot.h osoitetiedosto:

```
long etsipivot(long array[], long left, long right);
```

B.2 partion.h osoitetiedosto:

```
long partition(long array[], long left, long right, long pivot);
```

B.3 etsipivot.c lähdekoodi:

```
#include "extcode.h"

_declspec (dllexport) long etsipivot(long array[], long left,
                                      long right);

_declspec (dllexport) long etsipivot(long array[], long left,
                                      long right){

        long i;
long eka = array[left];
for (i = left + 1; i <= right; i++) {
if (array[i] > eka) return array[i];
else if (array[i] < eka) return eka;
}
return (0);
        }
```

B.4 partition.c lähdekoodi:

```
#include "extcode.h"

_declspec (dllexport) long partition(long array[], long left,
                                      long right, long pivot);

_declspec (dllexport) long partition(long array[], long left,
                                      long right, long pivot){


long temp;
do {
temp = array[left];
array[left] = array[right];
array[right] = temp;
while (array[left] < pivot)
left++;
while (array[right] >= pivot)
right--;
}
while (left <= right);
return left;

}
```

B.5 quicksort.c lähdekoodi:

```
/* Call Library source file */

#include "extcode.h"
#include "etsipivot.h"
#include "partition.h"
```

```
_declspec(dllexport) void quicksort(long array[], long left,
    long right);

_declspec(dllexport) void quicksort(long array[], long left,
    long right)
{

long i;
long pivot = etsipivot(array, left, right);

if (pivot != 0) {
i = partition(array, left, right, pivot);
quicksort(array, left, i-1);
quicksort(array, i, right);
}
}
```

C CIN pikalajittelun lähdekoodi

```
/* CIN source file */

#include "extcode.h"
#include "hosttype.h"

/* Typedefs */

typedef struct {
int32 dimSize;
int32 Numeric[1];
} TD1;
typedef TD1 **TD1Hdl;

MgErr CINRun(TD1Hdl Array, int32 *Numeric, int32 *Numeric2);
CIN MgErr CINProperties(int32 prop, void *data);
void quicksort(TD1Hdl array, int32 left, int32 right);
int32  etsipivot(TD1Hdl array, int32 left, int32 right);
int32  partition(TD1Hdl array, int32 left, int32 right, int32 pivot);

MgErr CINRun(TD1Hdl Array, int32 *Numeric, int32 *Numeric2){

/* Insert code here */
quicksort(Array, *Numeric, *Numeric2);
return noErr;
}
CIN MgErr CINProperties(int32 prop, void *data){

switch (prop) {
case kCINIsReentrant:
*(Bool32 *)data = TRUE;
return noErr;
}
```

```
return mgNotSupported;
}
void quicksort(TD1Hdl array, int32 left, int32 right){

int32 i= 0;
int32 pivot = etsipivot(array, left, right);

if (pivot != 0) {
i = partition(array, left, right, pivot);
quicksort(array, left, i-1);
quicksort(array, i, right);
}}
int32 etsipivot(TD1Hdl array, int32 left, int32 right){

int32 i = 0;
int32 eka = (*array)->Numeric[left];
for (i = left + 1; i <= right; i++) {
if ((*array)->Numeric[i] > eka) return (*array)->Numeric[i];
else if ((*array)->Numeric[i] < eka) return eka;
}
return (0);
}
int32 partition(TD1Hdl array, int32 left, int32 right , int32 pivot){

int32 temp = 0;
do {
temp = (*array)->Numeric[left];
(*array)->Numeric[left] = (*array)->Numeric[right];
(*array)->Numeric[right] = temp;
while ((*array)->Numeric[left] < pivot)
left++;
while ((*array)->Numeric[right] >= pivot)
right--;}
while (left <= right);
return left;}
```

D Pinoperustaisen pikalajittelun LabVIEW koodi

E Korjattu pinoperustaisen pikalajittelun LabVIEW koodi