Strukturoitu ohjelmointi

- top-down -ohjelmointi
- asteittainen tarkentaminen (stepwise refinement)
- rajoitetut kontrollirakenteet
- bottom up -ohjelmointi (Naur: action clusters),

virtuaalikoneen idea: ohjelma jaetaan tasoihin, joista kukin toteutetaan alempien tasojen työkalujen avulla. (vrt. Unix)

Top down -menetelmä

1. Määrittele tarkasti, mitä tietoja ohjelma saa ja mitä sen täytyy niille tehdä.
2. Jos tehtävä on riittävän yksinkertainen, kirjoita sen suorittava ohjelmakoodi.
3. Muuten jaa tehtävä pienempiin osiin ja määrittele tarkasti kunkin osan tehtävä ja liitäntä muuhun ohjelmaan.
4. Suorita vaiheet 1–4 erikseen kullekin osatehtävälle.

Hyvän aliohjelman/funktion ominaisuuksia

Sisäinen lujuus:
Suorittaa yhden selvästi määritellyn tehtävän.
Tehtävä ei välttämättä yksinkertainen, mutta määriteltävissä yksinkertaisella tavalla.

Ohjelman paloittelu toiston välttämiseksi tai aliohjelmien koon pitämiseksi pienenä voi johtaa huonoon lujuuteen.

Ulkoiset kytkennät:
Mitä enemmän aliohjelma riippuu globaaleista muuttujista ja muista aliohjelmista sitä vaikeampi ohjelmaa on muuttaa.

Ideaalitilanne on, että kaikki tieto välitetään parametreina (vrt. sin, sqrt).

Tehokkuus

1) Oikean algoritmin valinta. Hyvänkin algoritmin voi pilata huonolla ohjelmoinnilla. Toisaalta kullakin menetelmällä on omat rajoituksensa, joita nerokaskaan ohjelmoija ei voi ylittää.

Menetelmä valittava niin, että se on tehokas tyypillisen kokoluokan tehtäville.

2) Laske vain tarpeellinen. Usein lineaarisia yhtälöryhmiä ratkaistaan laskemalla ensin kerroinmatriisin käänteismatriisi. Jos matriisi on suuri, tietokoneaikaa haaskaantuu huomattavia määriä aivan turhaan.

3) Älä keksi pyörää uudestaan. Aliohjelmakirjastoista löytyy valmiit rutiinit useimpiin numeerisiin tehtäviin. Ne ovat numeriikan ammattilaisten kehittämiä ja huolella testattuja. Mikäli käytettävissä on jokin hyvä kirjasto (Nag, IMSL), sitä kannattaa kokeilla ensimmäiseksi ajan ja vaivojen säästämiseksi.

4) Tehtävään liittyvät oletukset. Esimerkiksi lineaarisen yhtälöryhmän ratkaisemiseen on olemassa erilaisia menetelmiä eri tyyppisiä kerroinmatriiseja varten.

Muistiviittausten lokaalisuus

Virtuaalimuistikoneissa ohjelma paloitellaan tavallisesti kiinteän kokoisiksi sivuiksi, jotka talletetaan levylle. Keskusmuistiin ladataan sivuja sitä mukaa kuin niitä tarvitaan. Kun ohjelma viittaa sivulle, joka joudutaan lukemaan levyltä, tapahtuu sivunpuutoskeskeytys, jonka ajaksi ohjelman suoritus pysähtyy. Kun muisti täyttyy, poistetaan tavallisesti ne sivut, joihin viimeksi tehdystä viittauksesta on kulunut pisin aika.

```
do i=1,100000
  x=a(i)
   ...
end do
```

Useimmissa tapauksissa viitattu alkio on samalla sivulla kuin edellinenkin, joten tarvittava sivu on hyvin suurella todennäköisyydellä jo valmiiksi muistissa.

Jos viittaukset kohdistuvat kaukana toisistaan oleviin alkioihin, ne osuvat usein eri sivuille. Mikäli kyseessä on suuri taulukko, aikaisemmin käytettyjä sivuja ehditään ehkä palauttaa levylle ennen kuin niihen viitataan uudestaan. Sivunpuutoskeskeytysten määrä voi kasvaa huomattavasti.

Muistiviittausten lokaalisuudella tarkoitetaan, että peräkkäiset viittaukset kohdistuvat aina lähellä toisiaan oleviin osoitteisiin. Lokaalisuus parantaa virtuaalimuistin käytön tehokkuutta ja siitä on etua myös välimuistin (cache) käytön optimoinnissa.

Reaalimuistikoneilla (Cray) sivunpuutoskeskeytyksiä ei voi tapahtua. Niilläkin muistiarkkitehtuuri saattaa aiheuttaa omia vaatimuksiaan.

Muistiviittausten lokaalisuudesta on aina etua!

Useampiulotteisten taulukoiden käsittely voi aiheuttaa ongelmia. Fortranissa taulukot talletetaan muista ohjelmointikielistä poiketen sarakkeittain eikä riveittäin.

```
do i=1,1000
  do j=1,1000
    x=a(i,j)
    ...
  end do
end do
```

Sisemmässä silmukassa haetaan taulukosta `a` joka tuhannes alkio, jotka ovat kaikki eri sivuilla.

Ohjelma toimii paljon tehokkaammin, jos silmukoiden järjestys voidaan vaihtaa:

```
do j=1,1000
  do i=1,1000
    x=a(i,j)
    ...
  end do
end do
```

Haja-ajatuksia ohjelmointityylistä

Ohjelman on parempi tehdä yksi asia kunnolla kuin jotenkuten vähän sitä sun tätä. Ennen ohjelmointia on määriteltävä täsmällisesti, mitä ohjelmalta vaaditaan.

Käytetystä tekniikasta riippumatta strukturoitu ohjelmointi pyrkii ohjelman pilkkomiseen pieniin, helposti hallittaviin ja riittävän yksinkertaisiin moduuleihin.

Aliohjelman pitäisi olla aina ymmärrettävissä kerralla yhtenä kokonaisuutena. Mitä monimutkaisempia kontrollirakenteita aliohjelma sisältää, sitä lyhyempi sen on oltava.

Kukin aliohjelma tekee yhden täsmällisesti määritellyn toimenpiteen. Tämä ei tarkoita, että toiminnon pitäisi olla yksinkertainen, pääasia on että se voidaan määritellä yksinkertaisesti, esim. ”lajittele tiedosto”, ”etsi osittaisdifferentiaaliyhtälön ratkaisu, joka toteuttaa annetut reunaehdot”, ”todista Goldbachin konjektuuri”. Samaan aliohjelmaan ei pidä kasata sekalaisia toimenpiteitä, jotka eivät kuulu loogisesti yhteen.

Aliohjelmien tulee olla toisistaan mahdollisimman riippumattomia. Globaalien muuttujien avulla tulisi välittää mahdollisimman vähän tietoa. Jos aliohjelmalla on hyvin paljon parametreja, jotakin voi olla vialla.

Hyvä ohjelma

Hyvän ohjelman ominaisuuksia (Yourdonin mukaan)

1. Toimivuus (vaatimusten mukaan)
2. Testauskustannusten minimointi
3. Ylläpitokustannusten minimointi
4. Joustavuus (”Nykyiseen ympäristöön parhaiten soveltuva järjestelmä soveltuu mahdollisimman huonosti mihin tahansa muuhun ympäristöön.”)
5. Laatimiskustannusten minimointi
6. Yksinkertaisuus
7. Tehokkuus

Hyvän ohjelman ominaisuuksia (Boehm et al.)

Kolme tarkastelukulmaa:

- Käytettävyys
- Ylläpito
- Siirrettävyys

1. Laiteriippumattomuus
    - muuttujien esitystapa
    - koneenläheinen I/O
2. Laskentatarkkuus
3. Täydellisyys
    - ei puuttuvia osia
    - syöttötietojen tarkistus (muoto, järkevyys)
    - taulukkoindeksien tarkistus
    - toimivuus kaikissa tilanteissa kaikilla syöttöaineistoilla
4. Yhdenmukaisuus
    - ulkoinen (käyttö, rajoitukset, ...)
    - sisäinen (muuttujien nimet, tyypit, vakiot, ...)
5. Tehokkuus
6. Saavutettavuus
    - ohjelman ominaisuuksien muutettavuus (syöttömuoto, tarkkuus, vakiot, ...)

7. Kommunikoivuus
    - käyttöliittymän helppous
    - syöttötietojen muoto
    - tulostuksen selkeys
    - virheilmoitukset
8. Strukturointi
    - modulaarisuus
    - moduulien määrittely
9. Itsensäselittävyys
    - esim. kuvaukset moduulien tarkoituksesta ja liitännästä
    - muuttujien kuvaavat nimet
10. Suppeus
    - ei tarpeettomia muuttujia
    - ei saavuttamatonta koodia
    - ei turhia osoitteita
    - toistuvat osat aliohjelmiksi
    - vakiokoodi silmukoiden ulkopuolelle
11. Luettavuus
    - ulkoasu
    - muuttujien määrittely
    - sopiva kommentointi
12. Laajennettavuus

Ohjelman ulkoasu

Seuraava on täysin sääntöjen mukainen osa Fortran-ohjelmaa:

```
      DO100I=1,10
      IF(X(I).GT.0.0.AND.X(I).LT.1.0)THEN
      Y(I)=SQRT(X(I))+123.4*SIN(X(I))+(A-1.0)*SQRT(
     11.0-X(I))
      ELSE
      Y(I)=0.0
      ENDIF
100   CONTINUE
```

Luettavampi versio:

```
      do 100 i=1,10
         if(x(i) .gt. 0.0 .and. x(i) .lt. 1.0) then
            y(i)=sqrt(x(i))+123.4*sin(x(i))+
     .           (a-1.0)*sqrt(1.0-x(i))
         else
            y(i)=0.0
         end if
  100 continue
```

F90-ratkaisu voisi olla

```
      y(1:10)=0
      do i=1,10
         if(x(i) > 0.0 .and. x(i) < 1.0) then
            y(i)=sqrt(x(i))+123.4*sin(x(i))+ &
                 (a-1.0)*sqrt(1.0-x(i))
      end do
```

tai jopa

```
     y=0
     where (x > 0.0 .and. x < 1.0) &
       y=sqrt(x)+123.4*sin(x)+(a-1.0)*sqrt(1.0-x)
```

Jo pelkästään typografiset muutokset parantavat ohjelmaa:

- Kontrollirakenteiden logiikkaa kannattaa havainnollistaa sisennyksillä, jolloin yhteen kuuluvien lauseiden muodostamat lohkot erottuvat välittömästi.
- Fortran 77:ssa välilyönneillä ole merkitystä. Ne voi jättää kokonaan pois, tai niitä voi sirotella minne mieli tekee. Luettavin tulos saadaan, kun niitä käytetään suunnilleen kuin luonnollisessa tekstissäkin.
- Kokonaan isoilla kirjaimilla kirjoitettu teksti on raskasta luettavaa. Pienet kirjaimet tekevät tekstin ilmavammaksi ja helpommin luettavaksi.
- Pitkät lausekkeet on syytä jakaa eri riveille mahdollisimman loogisista katkopaikoista. Enää ei tarvitse säästää reikäkortteja.
- Jatkorivin merkki kannattaa valita niin, ettei se sekaannu itse ohjelmaan. Fortran 90 on tässä suhteessa järkevämpi.

Älä käytä tabulaattorimerkkejä, vaan korvaa ne välilyönneillä. Tabulaattorimerkit saattavat tulostua eri tavoin eri laitteilla, jolloin suurella vaivalla tehty asettelu tuottaakin lähinnä sekasotkua.

Kommentit

Kommenttien avulla voit helpottaa niiden henkilöiden työtä, jotka joskus joutuvat muuttamaan tai tutkimaan ohjelmaasi. Tähän joukkoon kuulut myös itse.

Jokaisen ohjelmatiedoston alussa pitäisi olla selostus siitä, mihin ohjelmaa tai tässä tiedostossa olevaa ohjelmanosaa käytetään. Ohjelmaa kehitellessään tulee helposti tehneeksi tiedostoja, joilla on sellaisia havainnollisia nimiä kuin `a`, `test`, `koe3`, `prog` tai `xx`. Muutamaa päivää myöhemmin ei enää itsekään muista, mitä mikin niistä tekee.

Mikäli ohjelmaan on linkitettävä muissakin tiedostoissa olevia moduleita, alkukommenteissa olisi hyvä kertoa myös käännös- ja linkitysohjeet. Hyödyksi on myös tieto, mitä syöttö- ja tulostustiedostoja ohjelma mahdollisesti tarvitsee.

Älä selittele sotkuista ohjelmaa, vaan kirjoita uusiksi.

Kommenttien on vastattava ohjelman toimintaa:

```
if (x*y .le. 0.0) ... ! tulo negatiivinen
```

Tällaisia virheitä syntyy, kun muutetaan ohjelmakoodia, mutta unohdetaan korjata kommentit.

Kommenttien kirjoittamisessa kannattaa keskittyä oleelliseen:

```
i=i+1  ! lisataan i:ta yhdella
```

Tämä ei lisää ohjelman ymmärrettävyyttä. Seuraavassa muodossa kommentti sen sijaan antaa tietoa, josta saattaa olla hyötyä:

```
  i=i+1  ! siirrytaan matriisin seuraavalle
         ! vaakariville
```

Yleensä ei ole tarpeen selitellä yksittäisiä lauseita. Parempi on keskittyä laajempiin kokonaisuuksiin, kuten aliohjelmiin ja kokonaisiin kontrollirakenteisiin.

Aliohjelman alussa tulisi olla lyhyt selostus aliohjelman toiminnasta, sen parametreista, mahdollisista globaaleista muuttujista, sivuvaikutuksista ja funktion tapauksessa sen palauttamasta arvosta.

Hyvä tapa on kommentoida muuttujien määrittelyt. Tärkeintä on keskittyä oleellisimpiin muuttujiin; silmukan toistomuuttujia ja lausekkeissa esiintyviä tilapäisiä apumuuttujia ei kannata selittää.

```
c-------------------------------------------------
c ratkaistaan n. asteen yhtalo x**n-1=0
c juuret palautetaan vektorissa z(1), ..., z(n)
c-------------------------------------------------
subroutine solven(n,z)
  implicit none
  integer, intent(in):: n        ! yhtalon asteluku
  complex, intent(out):: z(:)    ! ratkaisuvektori
  real, parameter:: pi=3.141592654
  real phi
  integer k
  phi=2*pi/n
  do k=0,n-1
    z(k+1)=cmplx(cos(k*phi),sin(k*phi))
  enddo
  return
  end
```

Muuttujien määrittelyt

Implisiittisten määrittelyjen tarkoituksena on vähentää ohjelman kirjoittamiseen tarvittavaa työmäärää (ja säästää reikäkortteja). Suhteellisen pienestä säästöstä maksetaan kohtuuton hinta, kun jäljitetään syitä ohjelman kummalliseen toimintaan.

Mitä seuraava ohjelma tekee?

```
      NZERO=0
      DO 10 I=1.5
        IF (NUM(I).EQ.0) NZERO=NZERO+1
10    CONTINUE
```

Ohjelmassa on kaksi virhettä, joilta välttyy, jos kieltää implisiittiset määrittelyt.

`implicit none` ei ole F77-standardin mukainen. Lähes yhtä tehokas on `implicit logical A-Z`.

Muuttujille on syytä käyttää selkeitä ja merkitystä kuvaavia nimiä. Joissakin ympäristöissä globaalien tunnusten (aliohjelmien ja `common`-alueiden) nimien täytyy olla paikallisten muuttujien nimiä lyhempiä. Rajoitus johtuu linkittäjästä, jonka pitää pystyä linkittämään yhteen erikielisiä moduuleita.

Fortranin `dimension`-määrittely on tarpeeton. On turhaa kirjoittaa

```
dimension x(10)
```

kun saman asian voi sanoa lyhyemmin ja selvemmin:

```
real x(10)
```

Fortran 90:ssä kannattaa johdonmukaisuuden vuoksi käyttää määrittelyä

```
real, dimension(10) :: x
```

Kontrollirakenteet

Sisennysten vuoksi rivien alkupäät siirtyvät yhä kauemmas oikealle. Jos sisennyksiä on runsaasti, tulee kuvaruudun oikea reuna pian vastaan ja samalla ohjelmatekstin vastinkohtien löytäminen käy vaikeaksi. Tämä on merkki siitä, että moduulissa on jotakin vialla. Ehkä koko sen toimintaperiaate kannattaisi ajatella uudestaan eri tavalla, tai ehkä olisi paikallaan jakaa se useammaksi aliohjelmaksi.

Toistorakenteet ovat erikoistapauksia $n + \frac{1}{2}$ kierroksen silmukoista. Hyvin usein vastaan tulee tilanteita, joissa lopetusehtoa ei voi kovin luontevasti testata silmukan alussa tai lopussa.

```
      do i=0,n
        ..
        if (x(i) > 0) goto 100
          ...
      end do
  100 continue

      do i=0,n
        ..
        if (x(i) > 0) then
          ...
        end if
      end do
```

F90:ssä tämä voidaan kirjoittaa luettavampaan muotoon

```
      do i=0,n
        ..
        if (x(i) <= 0) exit
          ...
      end do
```

Goto-lauseet

Assembler-kielissä ainoa keinoa ohjata käskyjen suoritusjärjestystä on yleensä hyppykäsky. Myös ensimmäisissä Fortranin versioissa hyppykäsky oli lähes ainoa kontrollirakenne.

Jos ohjelmat suunnitellaan alusta alkaen huolella, päädytään yleensä täysin luontevasti ratkaisuihin, joissa hyppykäskyjä ei tarvita.

`goto` voi siirtää kontrollin minne tahansa: ohjelmalistauksen lukijalle ei ole itsestään selvää, mistä suoritus jatkuu, vaan hänen on etsittävä myös vastaava osoite.

Jokainen `goto` vaatii myös "vastakappaleen" eli osoitteen jonnekin. Osoite on vielä hankalampi olio kuin itse `goto`: siitä ei näy millään tavoin, mistä osoitteeseen hypätään.

```
      if (x .gt. 0.0) goto 200
      goto 100
200   y=sqrt(x)
100   ...
```

Tämä voidaan toteuttaa yksinkertaisemmin:

```
      if (x .gt. 0.0) y=sqrt(x)
```

Mitä seuraava ohjelma tekee?

```
      if (a .gt. b) then
      big=a
      goto 100
      end if
      big=b
100   if (big .gt. c) goto 200
      big=c
200   ...
```

Ohjelma voitaisiin kirjoittaa luettavampaan muotoon

```
      big=a
      if (b .gt. big) big=b
      if (c .gt. big) big=c
```

mutta itse asiassa samasta asiasta selvitään vielä yksinkertaisemmin

```
      big=max(a,b,c)
```

Seuraava pätkä on todellisesta ohjelmasta:

```
681  FORMAT( .. )
     DO 7813 LL=1,NTOT
     IF(LL.LE.N)WRITE(6,781)LL,XLOW(LL),XBE(LL),XUPP(LL)
     IF(LL.LE.N)WRITE(3,781)LL,XLOW(LL),XBE(LL),XUPP(LL)
     IF(LL.GT.N)WRITE(6,782)LL,XBE(LL)
     IF(LL.GT.N)WRITE(3,782)LL,XBE(LL)
781  FORMAT( .. )
782  FORMAT( .. )
7813 CONTINUE
     WRITE(6,7812)
     WRITE(3,7812)
7812 FORMAT( .. )
 78  CONTINUE
600  CONTINUE
 60  CONTINUE
999  KNC=0
     ...
     IF (N.NE.0) GOTO 100
100  CONTINUE
```

Ohjelmassa on useita sisäkkäisiä silmukoita, jotka päättyvät `continue`-lauseisiin. Silmukoiden alkua on kuitenkin vaikea löytää. Osoitteet eivät ole minkäänlaisessa järjestyksessä, joten on vaikea selvittää, mihin kontrolli siirtyy hyppykäskystä.

Erityisen hankalia ovat aritmeettiset `if`-lauseet ja assigned `go to` -lauseet. Fortran 90 -standardissa nämä on merkitty vanheneviksi (obsolescent).

Seuraava esimerkki on kirjasta Kernighan, Plaugerin: *Elements of programming style*:

```
 1   SUBROUTINE MERGE(P,Q,R,S,N)
 2   DIMENSION P(N), Q(N), R(N), S(N)
 3   LP=1
 4   LQ=1
 5   LR=1
 6   LS=1
 7   CALL ORDER(P,Q,R,LP,LQ,LR,N)
 8   IF (P(LP)) 10,9,10
 9   IF (Q(LQ)) 10,13,10
10   CALL ORDER(P,Q,S,LP,LQ,LS,N)
11   IF (P(LP)) 7,12,7
12   IF (Q(LQ)) 7,13,7
13   RETURN
14   END
```

Nerokkaat ratkaisut

Nerokas tapa:

```
real x(100,100)
do i=1,n
do j=1,n
 x(j,i)=(i/j)*(j/i)
end do
end do
```

Tehokas tapa:

```
real x(100,100), y(10000)
equivalence (x(1,1), y(1))
do i=1,10000
 y(i)=0.0
end do
do i=1,10000,101
 y(i)=1.0
end do
```

Luonnollinen tapa:

```
real x(100,100)
do i=1,100
 do j=1,100
  x(i,j)=0.0
 end do
 x(i,i)=1.0
end do
```

Fortran 90:

```
real, dimension (100,100) :: x
integer i
x = 0.0
do i=1,100
 x(i,i)=1.0
end do
```

Sivuvaikutukset

Funktionaaliset ohjelmamoduulit ovat funktioita, jotka palauttavat arvonaan suorittamansa laskennan tuloksen. Niillä ei ole mitään sivuvaikutuksia: ne eivät muuta niille välitettyjä argumentteja tai globaaleja muuttujia eivätkä suorita I/O-toimenpiteitä. Tällaisten ohjelmamoduulien käyttö on turvallista.

Aliohjelmat perustuvat pelkästään sivuvaikutuksiin. Koska ne eivät palauta mitään arvoa, niiden täytyy tehdä jotakin muuta, jotta niillä ylipäänsä olisi vaikutusta.

Vaarallisia ovat funktiot, joilla on myös sivuvaikutuksia.

```
real function f(x)
x=x+1.0
f=x**2
return
end
```

Funktiota voi kutsua esimerkiksi seuraavasti:

```
x=1.0
z=f(x)+f(x)
if (f(x).gt.z .or. f(x).lt.0.0) z=x
```

Mikä on muuttujan `z` arvo?

Toimiiko seuraava optimoitu versio samalla tavalla?

```
x=1.0
y=f(x)
z=2*y
if (y.gt.z .or. y.lt.0.0) z=x
```

Mitä tämä ohjelma mahtaa tulostaa?

```
call s(1)
write(*,*) 1
end
subroutine s(i)
i=i+1
return
end
```

Muista, että Fortranissa kaikki parametrit ovat viiteparametreja, eli aliohjelmalle välitetään todellisen parametrin osoite.

Syöttö ja tulostus

Ohjelman ei pidä vaatia käyttäjältä turhia tietoja, ja sen tulisi ymmärtää hyvin vapaamuotoista syöttötietoa. Tulostuksen on oltava niin itsensä selittävää, ettei sen tulkintaan tarvita monimutkaisia käyttöohjeita.

Testaa syöttöarvojen järkevyys. Identifioi järjettömät syöttöarvot, ja toivu tilanteesta mikäli mahdollista. Testaa erityisesti (odottamattomat) tiedostojen loput.

Tee syöttö mahdollisimman yksinkertaiseksi ja helpoksi. Vapaamuotoinen syöttö on järkevää pieniä kontrolliarvoja varten, vain suuret data-aineistot kannattaa lukea määrämuotoisena. Tulostus sen sijaan kannattaa useimmiten tehdä määrämuotoisesti.

Sijoita syöttö- ja tulostus selkeisiin aliohjelmiin. Älä sirottele tulostuskomentoja joka paikkaan.

Käytä syöttöarvoille oletuksia, jos mahdollista. Kaiuta syötetyt arvot tulostukseen (lokiin).

Vältä myös `format`-lauseita ja niiden numerointia. Vain kerran tai kahdesti käytettävän `format`-lauseen voi sijoittaa merkkijonovakiona suoraan sitä käyttävään `write`- tai `read`-lauseeseen, jolloin kyseinen I/O-lause ja sitä vastaava formaatti ovat samassa paikassa helposti löydettävissä.

Lopeta aineisto sopivaan syöttötietoon, tiedoston loppuun yms. Älä käytä lukumäärää:

```
      do i=1,1000
        read (1,*,END=100) z
        x(i)=z
      end do
  100 continue
```

Anna virheellisistä syöttötiedoista selkeä virheilmoitus.

Sekalaista

- Eristä laiteriippuvat osat erillisiksi aliohjelmiksi

- Valitse huolella tietorakenteet

- Jätä "likainen työ" tietokoneelle

```
I='41'O
I=ICHAR('A')
```

- Vältä ylimääräisiä apumuuttujia

```
xx = x0-x1
yy = y0-x1
z = xx**2+yy**2

z = (x0-x1)**2+(y0-y1)**2
```

- Testaa rajatapaukset. Varo taulukkojen indeksien ylityksiä, ja laskureiden "yhdellä pielessä" tilanteita. Tarkkaile muuttujien arvojen erikoistapauksia, ja muita erikoistilanteita, jotka usein ovat virheiden syinä.

- Alusta kaikki muuttujat, älä luota, että ne ovat nollia automaattisesti. (Poikkeus: suuria taulukoita ei aina kannata nollata, jos niitä ei tarvitakaan; muisti saatetaan varata vasta, jos sitä todella käytetään.)

- Varo pyöristysvirheitä: 10.0×0.1 on tuskin koskaan 1.0! Älä vertaa liukulukujen yhtäsuuruutta.

Tehokkuus

- Käytä valmiita ratkaisuja (kirjastorutiinit)

- Valitse tehtävään sopiva algoritmi: $N \log N$ -menetelmä on suurissa tehtävissä paljon tehokkaampi kuin N^2-menetelmä.

- Kirjoita huono koodi uudestaan

- Pyri muistiviittausten lokaalisuuteen (ongelma erityisesti useampiulotteisissa taulukoissa)

- Jos ohjelman suoritus kestää kauan, talleta ohjelman tila ajoittain. Jos ajo katkeaa jostakin syystä (sähkökatko tms.), koko työ ei mene hukkaan.