Dünaamiline planeerimine

Dünaamilise planeerimise meetod võimaldab mõnikord kergelt teha kiiremaks variantide läbivaatamisel põhineva aeglase algoritmi.

"Ekspresstakso"

Vaatleme sellist ülesannet: *"Ühel pikal tänaval on ekspresstaksode liiklus organiseeritud järgmisel viisil. Iga kilomeetri järel on taksopeatus. Igast peatusest sõidab ekspresstakso peatumata edasi mööda tänavat 1, 2, 3, ..., 10 kilomeetrit. Iga pikkusega distantsi jaoks on fikseeritud erinev hind. Reisija tahab sõita **n** kilomeetrit **(1 <= n <= 100)**. Milliseid distantse sõitvaid taksosid peaks reisija kasutama, et tema reis oleks võimalikult odav?"* (BOI'95 - Ekspresstakso).

Vaatleme sellist programmi:

```
var
      hind : array [1..10] of longint;

function maksumus(n : longint) : longint;
var
      i, min : longint;
begin
      if n = 0 then begin
            maksmus := 0;
            exit;
      end;
      min := MaxLongint;
      for i := 1 to 10 do begin
            if i <= n then begin
                  t := maksmus(n - i) + hind[i];
                  if t < min then min := t;
            end;
      end;
      maksmus := min;
end;
```

Kui **hind[i]** on sõidu pikkusega **i** hind, siis **maksumus(n)** arvutab tee pikkusega **n** minimaalse maksmuse. Kahjuks töötab see algoritm väga aeglaselt. Kui märkame, et funktsiooni **maksumus** kutsutakse sama parameetri **n** väärtusega välja palju kordi ja iga kord arvutatakse vastus uuesti otsast peale, võime asja parandada, võttes kasutusele veel kaks massiivi:

```
var
      oli : array [1..100] of boolean;
      vastus : array [1..100] of longint;
```

Massiivis **oli** tähistame need **n**-id mille jaoks on vastus juba leitud, massiivis **vastus** hoiame varem leitud vastuseid. Funktsiooni **maksumus** peab muutma nii, et alguses kontrollitakse, kas **oli[n]=true** ja sel juhul tagastatakse **vastus[n]**, muidu arvutatakse nagu ennegi. Enne arvutatud vastuse tagastamist märgitakse **oli[n]:=true** ja salvestatakse vastus massiivi **vastus**. Nüüd arvutatakse vastust iga **n**-i jaoks ainult üks kord. Seega teeb

algoritm umbes **n·10** sammu. Pannes tähele, et suuremate **n** väärtuste jaos vastuste arvutamiseks on vaja vastuseid ainult väiksemate **n** väärtuste jaoks, saab sama asja panna kirja veel lihtsamalt:

```
vastus[0] := 0;
for i := 1 to n do begin
      vastus[i] := MaxLongint;
      for j := 1 to 10 do begin
            if j <= i then begin
                  if vastus[i - j] + hind[j] < vastus[i] then begin
                        vastus[i] := vastus[i - j] + hind[j];
                        kuidas[i] := j;
                  end;
            end;
      end;
end;
```

Massiivi **kuidas** salvestatakse, millise takso abil jõuti vastavasse **n**-i. Järgnev algoritm trükib vajalikud reiside pikkused vastupidises järjekorras:

```
while n > 0 do begin
      writeln(kuidas[n]);
      n := n - kuidas[n];
end;
```

Tegelikult saab seda arvutada ka ainult massiivi **vastus** alusel. Siit näeme, et seda ülesannet saab lahendada nii rekursiivselt kui ka tabelit täites. Peaks märkima, et sama lahendus töötab ka juhul, kui reiside hinnad on erinevates peatustes erinevad.

"Õiglane jagamine"

Vaatleme sellist ülesannet: *"Kaks venda, Alan ja Bob, tahavad jagada omavahel hulka kingitusi. Iga kingituse peab andma kas Alanile või Bobil; kingitusi poolitada ei saa. Igal kingitusel on väärtus. **A** ja **B** tähistavad vastavalt Alanile ja Bobile antud kingituste kogusummat. Minimeerida vahe **A-B** absoluutväärtus. Kingituste arv **N** ei ületa 100. Kingituste väärtused on positiivsed täisarvud, mis ei ületa 200."* (CEOI'95 – Õiglane jagamine). Seda ülesannet saab lahendada järgneva funktsiooni abil:

```
var
      vaartus : array [1..100] of byte;

function saab(a, b, n : longint) : boolean;
begin
      if n = 0 then begin
            saab := (a = 0) and (b = 0);
            exit;
      end;
      saab := true;
      if a - vaartus[n] >= 0 then
            if saab(a - vaartus[n], b, n - 1) then exit;
      if b - vaartus[n] >= 0 then
            if saab(a, b - vaartus[n], n - 1) then exit;
      saab := false;
end;
```

Funktsioon **saab(a,b,n)** arvutab, kas saab jagada **n** esimest kingitust vendade vahel nii, et Alan saab **a** ja Bob **b** väärtuses kingitusi. Kasutades seda funktsiooni, saab leida vajalikud **A** ja **B**, proovides läbi kõik kombinatsioonid (**A+B** peab olema võrdne kingituste väärtuste summaga). Antud juhul on neid kombinatsioone ülimalt **100*200=20000**.

Selle lahendusega on sama probleem, mis eelmises näites: funktsooni kutsutakse välja samade **a,b,n** kombinatsioonide jaoks välja korduvalt ja iga kord sooritatakse arvutused algusest peale. Olukorda parandada eelmise ülesandega sarnasel moel: salvestada kombinatsioonide **a,b,n** jaoks arvutatud vastused tabelisse. Kui veel arvestada, et **a+b** on võrdne **n** esimese kingituse väärtuste summaga, võib võtta arvesse ainult **a,n** kuna **b** on nende poolt üheselt määratud. Nagu eelmises ülesandes, saab vastuste tabeli abil leida ka, millised kingitused peab kumbki vend valima. Kuna kombinatsioonide **a,n** sobivus sõltub ainult kombinatsioonidest **n-1** jaoks, võib tabelit arvutada kihtide kaupa: kihis **n=0** sobib ainult kombinatsioon **0,n**. Edasi arvutame kihi **i+1** kihi **i** alusel. Lõpuks saame kihi **n**, kust saame lugeda välja vastuse. Tulemusena saame lahenduse, mis teeb umbes **100*200*100=2000000** sammu ja kasutab sama palju mälu. Seega, programm töötab piisavalt kiiresti, kuid kasutatakse liiga palju mälu. Lahenduseks on hoida iga **a** kohta vähim **n**, mil see oli saavutatud. Pärast seda, kui selline struktuur on täidetud, saab sealt lugeda välja ka, millised kingitused peab millisele vennale andma. Vaja läheb ainult **100*200=20000** mäluühikut.

"Võrdsed sõned"

Vaatleme sellist ülesannet**: "***On antud kaks väikestest ladina tähtedest sõnet **u** ja **v** pikkustega kuni **100**. Leida minimaalne tähtede arv, mida peab neist eemaldama, et tulemused oleksid võrdsed"*. Vaatleme funktsiooni:

```
var
      u, v : string;

function muuta(a, b : integer) : integer;
begin
      if (a = 0) or (b = 0) then begin
            muuta := max(a,b);
            exit;
      end;
      if u[a] = u[b] then
            muuta := muuta(a - 1, b - 1);
      else
            muuta := 1 + min(muuta(a - 1, b), muuta(a, b - 1));
end;
```

Funktsioon **muuta(a,b)** arvutab, mitu muutust peab minimaalselt tegema et sõnede **u** ja **v** vastavalt **a**-täheline ja **b**-täheline prefiks ühtiksid (**i**-täheline prefiks on **i** esimest tähte). Seega **muuta(length(u), length(v))** arvutab otsitava vastuse. Nagu varemgi, saab teha tabeli **100x100** ja sooritada need arvutused seal. Selle tabeli alusel saab leida ka, millised tähed peab kummastki sõnast eemaldama.

Kuidas tunda ära dünaamilise planeerimisega lahendatav ülesanne?

Ülesanne peab olema taandatav sama tüüpi väiksemateks ülesanneteks. Näiteks, ülesandes "Õiglane jagamine" arvutati kombinatsiooni **a,b,n** sobivust kahe (**n-1)**-kombinatsiooni alusel. Ülesandes "Võrdsed sõned" otsustati, mitu muutust peab tegema, kasutades sama ülesande lahendusi lühemate sõnede jaoks. Samas peab erinevate vajalike alamülesannete hulk olema väike.

Tihti on mugav üritada mõelda välja lahendus, mis põhineb rekursiivsel variantide läbivaatamisel, nagu seda on tehtud näidetes. Sel juhul peab võimalike rekursiivse funktsiooni argumentide hulk olema väike. Võit saadakse selle arvelt, et sama alamülesande lahendust läheb vaja korduvalt. Pärast alamülesande esimest lahendamist salvestatakse vastus mällu ja järgmisel korral võetakse see sealt ja ei hakata algusest peale arvutama.

Ülesandeid olümpiaadidelt

- Lõppvoor 2004 Mustkunstnik
- Valikvõistlus 2002 Avaldis
- Lõppvoor 2002 Linker
- Lõppvoor 2002 Kassa
- Koolivoor 2001 Summa
- Piirkonnavoor 2001 Valimised
- Piirkonnavoor 1999 Alamjada
- Lahtine võistlus 1997 Summa

- BOI 2000 Muteksid
- BOI 1999 Kaubavedu
- BOI 1999 Sõnede vastavus
- BOI 1999 Avaldised
- BOI 1998 Puuviljad
- BOI 1995 Ekspresstakso

- CEOI 1997 Domino
- CEOI 1995 Fair sharing

- IOI 1996 Longest prefix
- IOI 1995 Shopping Offers
- IOI 1993 Canadian flights