Luku 6

Dynaaminen ohjelmointi

Dynaamisessa ohjelmoinnissa on ideana jakaa ongelman ratkaisu pienempiin osaongelmiin, jotka voidaan ratkaista toisistaan riippumattomasti. Jokaisen osaongelman ratkaisu tallennetaan muistiin, minkä ansiosta samaa osaongelmaa ei tarvitse ratkaista moneen kertaan. Tämä vähentää ratkaisevasti ongelman suurten tapausten käsittelyyn vaadittavaa aikaa.

Dynaamista ohjelmointia voi hyödyntää usein algoritmien suunnittelussa. Vaikka menetelmän perusidea on melko suoraviivainen, dynaamisen ohjelmoinnin soveltaminen eri tilanteissa vaatii kokemusta.

6.1 Funktion muisti

Dynaamisen ohjelmoinnin lähtökohtana on, että ongelman ratkaisu voidaan esittää rekursiivisena funktiona. Tämän jälkeen tehokkaan ratkaisun saamiseksi laskenta toteutetaan niin, että sama funktion arvo riittää laskea vain kerran. Käytännössä tämä tapahtuu tallentamalla funktion arvoja taulukkoon muistiin sitä mukaa kuin niitä lasketaan.

Tarkastellaan esimerkkinä Fibonaccin lukujen laskemista. Fibonaccin luvut voidaan määritellä seuraavasti rekursiivisesti:

$$F(n) = \begin{cases} 0 & \text{jos } n = 0 \\ 1 & \text{jos } n = 1 \\ F(n-2) + F(n-1) & \text{jos } n \geq 2 \end{cases}$$

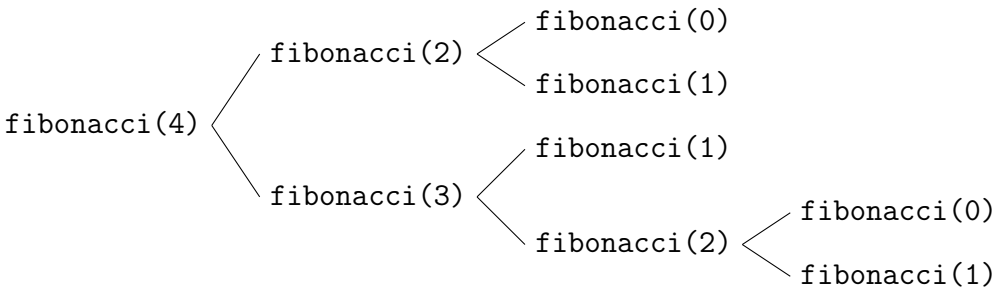
Määritelmä tarkoittaa, että tietty Fibonaccin luku saadaan kahden edellisen Fibonaccin luvun summana. Ensimmäiset Fibonaccin luvut ovat 0, 1, 1, 2, 3, 5, 8, 13 ja 21.

Fibonaccin lukujen rekursiivinen määritelmä voidaan suoraan muuttaa C++-funktioksi seuraavasti:

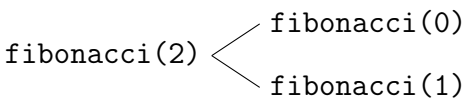
```
long fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n-2) + fibonacci(n-1);
}
```

Tämä funktio toimii pienillä n:n arvoilla mainiosti, mutta funktio on liian hidas suurempien Fibonaccin lukujen laskemiseen. Esimerkiksi testikoneella arvon $n = 40$ laskeminen kestää noin 2 sekuntia, arvon $n = 45$ laskeminen noin 20 sekuntia ja arvon $n = 50$ laskeminen jo lähes 4 minuuttia.

Funktion hitaus johtuu siitä, että se tekee paljon työtä moneen kertaan. Esimerkiksi arvon $n = 4$ laskennassa muodostuu seuraava rekursiopuu:



Ongelmana on, että kutsu `fibonacci(2)` laskee tuloksen rekursiivisesti kahdessa eri kohdassa seuraavasti:



Kun n:n arvo on suurempi, hyvin monessa kohtaa rekursiopuuta on toistuvia rekursiivisia funktiokutsuja ja laskenta kestää kauan. Suuretkin Fibonaccin luvut lasketaan loppujen lopuksi luvuista 0 ja 1 muodostuvana summana, koska funktio ei palauta mitään muuta lukua suoraan.

Ratkaisu ongelmaan on muuttaa funktion toteutusta niin, että se laskee kunkin n:n arvon vain kerran. Aiemmin laskettujen arvojen muisti on mahdollista yhdistää suoraan rekursiiviseen funktioon seuraavasti:

```
long muisti[1000];

long fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    // jos arvo on jo laskettu, palautetaan se suoraan
    if (muisti[n]) return muisti[n];
    // muuten lasketaan arvo ja laitetaan se muistiin
    muisti[n] = fibonacci(n-2) + fibonacci(n-1);
    return muisti[n];
}
```

Tämän muutoksen seurauksena funktio laskee minkä tahansa Fibonaccin luvun salamannopeasti, kunhan tulos mahtuu long-tyyppiin.

Vaihtoehtoinen tapa soveltaa dynaamista ohjelmointia on luopua rekursiosta ja laskea sen sijaan tulokset muistiin silmukalla pienimmästä arvosta suurimpaan. Näin saadaan seuraava ratkaisu:

```
long fibonacci(int n) {
    long muisti[n+1];
    muisti[0] = 0;
    muisti[1] = 1;
    for (int i = 2; i <= n; i++) {
        muisti[i] = muisti[i-2] + muisti[i-1];
    }
    return muisti[n];
}
```

Yleensä on parempi vaihtoehto toteuttaa dynaamisen ohjelmoinnin ratkaisu silmukalla. Silmukkatoteutus on jonkin verran lyhyempi ja siinä ei ole vaarana pinomuistin loppuminen liian syvän rekursion vuoksi. Kuitenkin rekursiivinen funktio voi auttaa ongelman hahmottamista, vaikka lopullinen toteutus käyttäisikin silmukkaa.

6.2 Esimerkki: Nopanheitot

Seuraava esimerkki on tyypillinen dynaamisen ohjelmoinnin tehtävä. Suoraviivainen rekursiivinen ratkaisu on liian hidas, koska erilaisten yhdistelmien määrä kasvaa räjähdysmäisesti. Dynaaminen ohjelmointi kuitenkin tehostaa laskentaa yhdistämällä samanlaisten tilanteiden käsittelyn.

Tehtävä

Laske tehokkaasti, kuinka monella eri tavalla annettu kokonaisluku on mahdollista tuottaa nopan silmälukujen summana.

Esimerkiksi luku 4 on mahdollista tuottaa 8 eri tavalla:

- $1+1+1+1$
- $1+1+2$
- $1+2+1$
- $2+1+1$
- $1+3$
- $3+1$
- $2+2$
- 4

Pohdinta

Olkoon $N(n)$ erilaisten luvun n muodostavien summien määrä. Tehtävänannon perusteella $N(4) = 8$. Tarkastellaan esimerkin vuoksi ennen yleistä ratkaisua summia, jotka muodostavat luvun 50. Oleellinen havainto on, että jokaisen mahdollisen summan muoto on yksi seuraavista:

- $\ldots + 1 = 50$
- $\ldots + 2 = 50$
- $\ldots + 3 = 50$
- $\ldots + 4 = 50$
- $\ldots + 5 = 50$
- $\ldots + 6 = 50$

Nyt esimerkiksi sellaisia summia, joissa viimeinen luku on 1, on yhtä monta kuin luvun 49 muodostavia summia. Vastaava päättely soveltuu kaikkiin muihinkin tapauksiin, minkä ansiosta $N(50)$ voidaan laskea näin:

$$N(50) = N(49) + N(48) + N(47) + N(46) + N(45) + N(44)$$

Yleisesti ratkaisuun saadaan seuraava rekursiivinen funktio:

$$N(n) = \begin{cases} 0 & \text{jos } n < 0 \\ 1 & \text{jos } n = 0 \\ \sum_{i=1}^{6} N(n-i) & \text{jos } n > 0 \end{cases}$$

Tässä $\sum_{i=1}^{6} N(n-i)$ tarkoittaa $N(n-1)+N(n-2)+\cdots+N(n-6)$.

Funktioon liittyy kaksi erikoistapausta: Jos n on negatiivinen, niin tulos on 0, koska negatiivista summaa ei voi muodostaa mitenkään. Jos taas n on 0, niin tulos on 1, koska tyhjän summan arvona on 0.

Ratkaisu

Tässä on dynaamisen ohjelmoinnin ratkaisu ajassa $O(n)$:

```
long nopat(int n) {
    long muisti[n+1];
    muisti[0] = 1;
    for (int i = 1; i <= n; i++) {
        muisti[i] = 0;
        for (int j = 1; j <= 6; j++) {
            if (i-j >= 0) {
                muisti[i] += muisti[i-j];
            }
        }
    }
    return muisti[n];
}
```

Nyt saadaan laskettua tehokkaasti esimerkiksi seuraavat tulokset:

- $N(10) = 492$
- $N(20) = 463968$
- $N(30) = 437513522$
- $N(40) = 412567404640$
- $N(50) = 389043663364337$

6.3 Lisätekniikoita

Seuraavat tekniikat tulevat usein vastaan dynaamisen ohjelmoinnin tehtävissä. Luvun loppuosan esimerkit esittelevät näiden tekniikoiden käyttöä mutkikkaammissa dynaamisen ohjelmoinnin sovelluksissa.

Moniulotteisuus

Joskus ongelman ratkaisuna on rekursiivinen funktio, jossa on useita parametreja. Tällöin vastaavasti dynaamisen ohjelmoinnin taulukossa on useita ulottuvuuksia. Esimerkiksi ruudukkoon liittyvissä ongelmissa kaksiulotteinen taulukko on luonteva ratkaisu. Joskus tehtävään on olemassa monta erilaista dynaamisen ohjelmoinnin ratkaisutapaa, joissa taulukon ulottuvuuksien määrä vaihtelee.

Ratkaisun muodostus

Dynaamisen ohjelmoinnin tehtävissä täytyy välillä ilmoittaa yhden lukuarvon sijasta tarkemmin, miten halutunlaisen ratkaisun voi muodostaa. Esimerkiksi jos etsittävänä on pienin ratkaisu, ratkaisun kustannuksen lisäksi täytyy tulostaa yksi mahdollinen ratkaisu. Joskus ratkaisusta täytyy tallentaa lisätietoa muistiin, jotta sen voi muodostaa lopuksi tehokkaasti.

Muistin säästäminen

Jos rekursiivisen funktion erilaisia parametriyhdistelmiä on suuri määrä, dynaamisen ohjelmoinnin taulukosta tulee iso. Tämä ei ole yleensä ongelma, mutta joskus tehtävän muistirajat saattavat tulla vastaan. Usein kuitenkin muistinkäyttöä saa vähennettyä merkittävästi pitämällä muistissa vain kulloinkin tarvittavaa taulukon osaa.

Esimerkkinä tästä on Fibonaccin lukujen määrittely.

$$F(n) = \begin{cases} 0 & \text{jos } n = 0 \\ 1 & \text{jos } n = 1 \\ F(n-2) + F(n-1) & \text{jos } n \geq 2 \end{cases}$$

Arvon $F(n)$ laskennassa ei ole välttämätöntä varata taulukkoa n arvolle, vaan riittää pitää muistissa funktion kaksi viimeisintä arvoa.

6.4 Esimerkki: Ruudukon reitti

Seuraavassa tehtävässä käsiteltävänä on ruudukko, minkä vuoksi rekursiivisessa funktiossa on kaksi parametria ja dynaamisen ohjelmoinnin taulukossa on kaksi ulottuvuutta. Tehtävän laajennoksessa ratkaisun täytyy myös ilmoittaa, miten reitti muodostuu.

Tehtävä

Uolevi aloittaa ruudukon vasemmasta ylänurkasta ja kulkee oikeaan alanurkkaan liikkumalla joka vaiheessa ruudun alaspäin tai oikealle. Tehtävänä on selvittää suurin mahdollinen lukujen summa tällaisella reitillä.

Esimerkiksi alla olevassa ruudukossa suurin mahdollinen summa on 67, jonka saa kulkemalla merkittyä reittiä.

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

Pohdinta

Olkoot $R(y,x)$ ruudukon tietyssä kohdassa oleva luku ja $S(y,x)$ suurin summa reitillä ruudukon vasemmasta yläkulmasta kyseiseen kohtaan. Ruudukon vasemman yläkulman koordinaatit ovat $(0,0)$. Esimerkiksi tehtävänannon tapauksen ratkaisu on $S(4,4) = 67$.

Dynaamisen ohjelmoinnin ratkaisu laskee arvon $S(y,x)$ lisäämällä arvoon $R(y,x)$ suuremman arvoista $S(y-1,x)$ (reitti tulee ylhäältä) ja $S(y,x-1)$ (reitti tulee vasemmalta). Erikoistapauksina ovat tilanteet, joissa toinen tai kumpikin koordinaatti on 0. Rekursiivinen funktio on seuraava:

$$S(y,x) = R(y,x) + \begin{cases} 0 & \text{jos } y = 0 \text{ ja } x = 0 \\ S(y,x-1) & \text{jos } y = 0 \\ S(y-1,x) & \text{jos } x = 0 \\ \max(S(y,x-1), S(y-1,x)) & \text{muuten} \end{cases}$$

Ratkaisu

Ratkaisun voi toteuttaa käytännössä näin ajassa $O(n^2)$:

```
int n = 5;
int r[5][5] = {{3,7,9,2,7},
               {9,8,3,5,5},
               {1,7,9,8,5},
               {3,8,6,4,10},
               {6,3,9,7,8}};

int s[n][n];

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        int a = (i > 0) ? s[i-1][j] : 0;
        int b = (j > 0) ? s[i][j-1] : 0;
        s[i][j] = r[i][j] + max(a, b);
    }
}

cout << s[n-1][n-1] << endl;
```

Reitin muodostus

Dynaamisen ohjelmoinnin taulukosta s on mahdollista myös selvittää paras reitti. Ideana on lähteä liikkeelle ruudukon oikeasta alanurkasta ja liikkua askel kerrallaan ylös tai vasemmalle sen mukaan, kummassa suunnassa taulukoitu summa on suurempi. Seuraava koodi tulostaa suurimman summan tuottavan reitin käänteisessä järjestyksessä:

```
int y = n-1, x = n-1;
while (true) {
    cout << y << " " << x << endl;
    if (y == 0 && x == 0) break;
    int a = (y > 0) ? s[y-1][x] : 0;
    int b = (x > 0) ? s[y][x-1] : 0;
    if (a > b) y--; else x--;
}
```

6.5 Esimerkki: Alijonot

Tehtävästä on välillä vaikeaa nähdä päältä päin, että siinä voi soveltaa dynaamista ohjelmointia. Seuraavassa on yksi esimerkki tällaisesta tehtävästä. Tehtävä vaikuttaa aluksi laskennallisesti haastavalta, mutta siihen on olemassa lyhyt dynaamisen ohjelmoinnin ratkaisu.

Tehtävä

Merkkijonon *alijono* saadaan, kun merkkijonosta poistetaan merkkejä ja muiden merkkien järjestys säilyy ennallaan. Tehtävänä on laskea tehokkaasti annetun merkkijonon erilaisten alijonojen määrä. Voit olettaa, että merkkijono muodostuu kirjaimista A–Z.

Esimerkiksi merkkijonon HAAPA erilaiset alijonot ovat A, AA, AAA, AAP, AAPA, AP, APA, H, HA, HAA, HAAA, HAAP, HAAPA, HAP, HAPA, HP, HPA, P ja PA. Yhteensä erilaisia alijonoja on siis 19.

Pohdinta

Merkkijonon alijonojen yhteismäärä on $2^n - 1$, koska jokainen merkkien osajoukko on alijono tyhjää joukkoa lukuun ottamatta. Yksi tapa ratkaista tehtävä on muodostaa kaikki alijonot ja lisätä ne joukkorakenteeseen. Tällöin kuitenkin aika- ja tilavaativuudet ovat ainakin luokkaa $O(2^n)$.

Dynaamisen ohjelmoinnin ratkaisussa on ideana laskea jokaiselle merkkijonon merkille, montako erilaista alijonoa on olemassa, jotka päättyvät kyseiseen merkkiin. Seuraavassa ovat merkkijonon HAAPA alijonojen määrät. Esimerkiksi toiseen A:han päättyvät alijonot ovat A, AA, HA ja HAA ja P:hen päättyvät alijonot ovat AAP, AP, HAAP, HAP, HP ja P.

H	A	A	P	A
1	2	4	6	12

Tarkastellaan esimerkiksi P:hen päättyvien alijonojen laskemista. Ensinnäkin yksi alijonoista on pelkkä kirjain P. Muissa alijonoissa ennen P:tä on jokin toinen kirjain. Tässä tapauksessa kirjain on joko H tai A, koska muita kirjaimia ei ole esiintynyt merkkijonossa ennen P:tä. Kunkin kirjaimen kohdalla alijonojen määrä on yhtä suuri kuin niiden alijonojen määrä, jotka päättyvät kyseisen kirjaimen edelliseen esiintymään. Siis HP- ja AP-päätteisiä alijonoja on 1 ja 4. Yhteensä alijonoja on $1 + 1 + 4 = 6$.

Yleisesti jokaisen merkin kohdalla kyseiseen merkkiin päättyvien alijonojen määrän saa laskemalla yhteen luvun 1 (alijono on pelkkä merkki) sekä kunkin aakkoston merkin edellisen esiintymän alijonojen määrän. Vastaavalla idealla voi laskea myös koko merkkijonojen alijonojen määrän.

Ratkaisu 1

Seuraava ratkaisu laskee alijonojen määrän äskeisillä ideoilla ajassa $O(nm)$, jossa n on merkkijonon pituus ja m on aakkoston koko. Taulukossa `maarat` on laskuri jokaiselle aakkoston merkille.

```
string sana = "HAAPA";
long maarat[256] = {0};
for (int i = 0; i < sana.length(); i++) {
    long uusi = 1;
    for (char m = 'A'; m <= 'Z'; m++) {
        uusi += maarat[m];
    }
    maarat[sana[i]] = uusi;
}
long tulos = 0;
for (char m = 'A'; m <= 'Z'; m++) {
    tulos += maarat[m];
}
cout << tulos << endl;
```

Ratkaisu 2

Sama ratkaisu on mahdollista toteuttaa myös ajassa $O(n)$ seuraavasti:

```
string sana = "HAAPA";
long maarat[256] = {0};
long tulos = 0;
for (int i = 0; i < sana.length(); i++) {
    long uusi = 1+tulos;
    tulos = tulos-maarat[sana[i]]+uusi;
    maarat[sana[i]] = uusi;
}
cout << tulos << endl;
```

6.6 Ahneet algoritmit

Dynaaminen ohjelmointi käy läpi tehokkaasti kaikki mahdolliset tavat ratkaista ongelma. Joskus vielä tehokkaampi menetelmä on *ahne algoritmi*, joka valitsee joka vaiheessa suoraan oikean vaihtoehdon. Seuraavassa ongelmassa dynaamisen ohjelmoinnin sijasta voi käyttää ahnetta algoritmia. Mutta jos ongelmaa muuttaa vähän, niin ahne algoritmi ei enää toimi.

Tarkastellaan annetun rahamäärän muodostamista käyttäen mahdollisimman vähän eurokolikoita. Eurokolikot ovat 1 ja 2 euron kolikot sekä 1, 2, 5, 10, 20 ja 50 sentin kolikot. Esimerkiksi 3,14 euron muodostamiseen tarvitaan vähimmillään 5 eurokolikkoa: 1 ja 2 euron kolikot, 10 sentin kolikko sekä kaksi 2 sentin kolikkoa.

Tämä ongelma on lähes sama kuin luvun 6.2 annetun summan tuottavien nopan heittosarjojen laskeminen. Erot ovat, että silmälukujen 1–6 sijasta käytetään eurokolikoita ja tehtävänä ei ole selvittää ratkaisujen yhteismäärää vaan pienin mahdollinen määrä summattavia. Seuraava dynaamisen ohjelmoinnin ratkaisu käsittelee rahamääriä sentteinä.

```
int kolikot(int rahat) {
    int[] arvot = {1, 2, 5, 10, 20, 50, 100, 200};
    int muisti[rahat+1];
    muisti[0] = 0;
    for (int i = 1; i <= rahat; i++) {
        muisti[i] = 999999999; // "ääretön"
        for (int j = 0; j < 8; j++) {
            if (i-arvot[j] >= 0) {
                int vanha = muisti[i-arvot[j]];
                muisti[i] = min(muisti[i], vanha+1);
            }
        }
    }
    return muisti[rahat];
}
```

Esimerkiksi pienin määrä kolikoita 3,14 euron muodostamiseksi selviää käymällä läpi mahdolliset valinnat summan viimeiseksi kolikoksi. Jos viimeinen kolikko on 1 sentin kolikko, riittää etsiä pienin määrä kolikoita 3,13 euron muodostamiseksi ja lisätä tähän yksi. Jos taas viimeinen kolikko on 2 sentin kolikko, tutkitaan 3,12 euron muodostamista jne.

Dynaamisen ohjelmoinnin algoritmi toimii varmasti, koska se käy läpi kaikki mahdolliset vaihtoehdot. Kuitenkin se tekee turhaa työtä, koska myös seuraava ahne algoritmi löytää aina oikean ratkaisun. Ideana on käydä kolikot läpi suurimmasta pienimpään ja vähentää jokaista kolikkoa mahdollisimman monta kertaa jäljellä olevasta rahamäärästä.

```
int kolikot(int rahat) {
    int[] arvot = {1, 2, 5, 10, 20, 50, 100, 200};
    int tulos = 0;
    for (int i = 7; i >= 0; i--) {
        while (rahat >= arvot[i]) {
            rahat -= arvot[i];
            tulos++;
        }
    }
    return tulos;
}
```

Esimerkiksi 3,14 euron muodostamisessa 2 euron ja 1 euron kolikot valitaan mukaan kerran. Tämän jälkeen 50 ja 20 sentin kolikoita ei valitaan ja 10 sentin kolikko valitaan kerran. Lopuksi 5 sentin kolikoita ei valita, 2 sentin kolikko valitaan kahdesti ja 1 sentin kolikoita ei valita.

Ahneissa algoritmeissa on hyvänä puolena, että ne ovat yleensä erittäin tehokkaita ja lyhyitä. Niihin liittyy kuitenkin myös vakava ongelma: ahneesta algoritmista voi olla vaikeaa tietää päältä päin, toimiiko se vai ei, ja useimmat ahneet algoritmit eivät toimi. Erityisesti ohjelmointikilpailuissa on suuri riski käyttää ahnetta algoritmia, jos ei ole varma sen toimivuudesta.

Yllä oleva ahne algoritmi toimii – eurokolikoiden tapauksessa. Jos kolikkokanta muuttuu, niin algoritmi ei ehkä toimi. Esimerkiksi jos 1 euron kolikon sijasta olisi 1,01 euron kolikko, algoritmi ei enää toimisi. Silloin algoritmi muodostaisi 1,50 euroa 6 kolikolla ($1,01+0,20+0,20+0,05+0,02+0,02$) vaikka pienin määrä olisi 3 kolikkoa ($0,50+0,50+0,50$).