

Kvik bestun

Bergur Snorrason

28. janúar 2022

Rakningarvensl

- ▶ Talnarunan a_1, a_2, \dots kallast *k-ta stigs rakningarvensl* ef til er fall $f: \mathbb{R}^k \rightarrow \mathbb{R}$ þannig að

$$a_n = f(a_{n-1}, a_{n-2}, \dots, a_{n-k})$$

fyrir öll $n > k$.

Rakningarvensl

- ▶ Talnarunan a_1, a_2, \dots kallast *k-ta stigs rakningarvensl* ef til er fall $f: \mathbb{R}^k \rightarrow \mathbb{R}$ þannig að

$$a_n = f(a_{n-1}, a_{n-2}, \dots, a_{n-k})$$

fyrir öll $n > k$.

- ▶ Frægasta dæmið um rakningarvensl er *Fibonacci runan*.

Rakningarvensl

- ▶ Talnarunan a_1, a_2, \dots kallast *k-ta stigs rakningarvensl* ef til er fall $f: \mathbb{R}^k \rightarrow \mathbb{R}$ þannig að

$$a_n = f(a_{n-1}, a_{n-2}, \dots, a_{n-k})$$

fyrir öll $n > k$.

- ▶ Frægasta dæmið um rakningarvensl er *Fibonacci runan*.
- ▶ Hún er k -stigs rakningarvensl gefin með fallinu $f(x, y) = x + y$.

Rakningarvensl

- ▶ Talnarunan a_1, a_2, \dots kallast k -ta stigs rakningarvensl ef til er fall $f: \mathbb{R}^k \rightarrow \mathbb{R}$ þannig að

$$a_n = f(a_{n-1}, a_{n-2}, \dots, a_{n-k})$$

fyrir öll $n > k$.

- ▶ Frægasta dæmið um rakningarvensl er *Fibonacci runan*.
- ▶ Hún er annars stigs rakningarvensl gefin með fallinu $f(x, y) = x + y$.

Rakningarvensl

- ▶ Talnarunan a_1, a_2, \dots kallast k -ta stigs rakningarvensl ef til er fall $f: \mathbb{R}^k \rightarrow \mathbb{R}$ þannig að

$$a_n = f(a_{n-1}, a_{n-2}, \dots, a_{n-k})$$

fyrir öll $n > k$.

- ▶ Frægasta dæmið um rakningarvensl er *Fibonacci runan*.
- ▶ Hún er annars stigs rakningarvensl gefin með fallinu $f(x, y) = x + y$.
- ▶ Reikna má upp úr þessum venslum endurkvæmt.

```
1 #include <stdio.h>
2
3 int fib(int x)
4 {
5     if (x < 3) return 1;
6     return fib(x - 1) + fib(x - 2);
7 }
8
9 int main()
10 {
11     int n;
12     scanf("%d", &n);
13     printf("%d\n", fib(n));
14     return 0;
15 }
```

- ▶ Í hverju skrefi skiptist endurkvæmnin í tvennt svo þetta forrit hefur tímaflækju $\mathcal{O}(\quad)$.

- ▶ Í hverju skrefi skiptist endurkvæmnin í tvennt svo þetta forrit hefur tímaflækju $\mathcal{O}(2^n)$.

- ▶ Í hverju skrefi skiptist endurkvæmnin í tvennt svo þetta forrit hefur tímaflækju $\mathcal{O}(2^n)$.
- ▶ Við getum þó bætt þetta til muna með því að geyma niðurstöðuna úr hverju kalli.

- ▶ Í hverju skrefi skiptist endurkvæmnin í tvennt svo þetta forrit hefur tímaflækju $\mathcal{O}(2^n)$.
- ▶ Við getum þó bætt þetta til muna með því að geyma niðurstöðuna úr hverju kalli.
- ▶ Þá nægir að reikna hvert gildi einu sinni.

- ▶ Í hverju skrefi skiptist endurkvæmnin í tvennt svo þetta forrit hefur tímaflækju $\mathcal{O}(2^n)$.
- ▶ Við getum þó bætt þetta til muna með því að geyma niðurstöðuna úr hverju kalli.
- ▶ Þá nægir að reikna hvert gildi einu sinni.
- ▶ Þessi viðbót kallast *minnun* (e. *memoization*).

```
1 #include <stdio.h>
2 #define MAXN 1000000
3
4 int d[MAXN]; // Hér geymum við skilagildi fib(...).
5             // Ef d[i] = -1 þá eigum við eftir að reikna fib(i).
6 int fib(int x)
7 {
8     if (d[x] != -1) return d[x];
9     if (x < 2) return 1;
10    return d[x] = fib(x - 1) + fib(x - 2);
11 }
12
13 int main()
14 {
15     int n, i;
16     scanf("%d", &n);
17     for (i = 0; i < n; i++) d[i] = -1;
18     printf("%d\n", fib(n - 1));
19     return 0;
20 }
```

Ofansækin kvik bestun

- ▶ Nú reiknum við hvert gildi aðeins einu sinni.

Ofansækin kvik bestun

- ▶ Nú reiknum við hvert gildi aðeins einu sinni.
- ▶ Við þurfum að reikna n gildi og hvert gildi má reikna í $\mathcal{O}(\quad)$ tíma, svo í heildina er forritið $\mathcal{O}(\quad)$.

Ofansækin kvik bestun

- ▶ Nú reiknum við hvert gildi aðeins einu sinni.
- ▶ Við þurfum að reikna n gildi og hvert gildi má reikna í $\mathcal{O}(1)$ tíma, svo í heildina er forritið $\mathcal{O}(n)$.

Ofansækin kvik bestun

- ▶ Nú reiknum við hvert gildi aðeins einu sinni.
- ▶ Við þurfum að reikna n gildi og hvert gildi má reikna í $\mathcal{O}(1)$ tíma, svo í heildina er forritið $\mathcal{O}(n)$.

Ofansækin kvik bestun

- ▶ Nú reiknum við hvert gildi aðeins einu sinni.
- ▶ Við þurfum að reikna n gildi og hvert gildi má reikna í $\mathcal{O}(1)$ tíma, svo í heildina er forritið $\mathcal{O}(n)$.
- ▶ Án minnunar náum við með erfiðum að reikna fertugust Fibonacci töluna (því eframatið $\mathcal{O}(2^n)$ mætti bæta ögn) en með minnun náum við hæglega að reikna milljónustu Fibonacci töluna (hún mun þó ekki einu sinni passa í 64 bita).

Ofansækin kvik bestun

- ▶ Nú reiknum við hvert gildi aðeins einu sinni.
- ▶ Við þurfum að reikna n gildi og hvert gildi má reikna í $\mathcal{O}(1)$ tíma, svo í heildina er forritið $\mathcal{O}(n)$.
- ▶ Án minnunar náum við með erfiðum að reikna fertugust Fibonacci töluna (því eframatið $\mathcal{O}(2^n)$ mætti bæta ögn) en með minnun náum við hæglega að reikna milljónustu Fibonacci töluna (hún mun þó ekki einu sinni passa í 64 bita).
- ▶ Ef lausnin okkar er endurkvæm með minnun kallast hún *ofansækin kvik bestun* (e. *top down dynamic programming*).

Neðansækin kvik bestun

- ▶ Það er þó lítið mál að breyta endurkvæmnu lausninni okkar í ítraða lausn.

Neðansækin kvik bestun

- ▶ Það er þó lítið mál að breyta endurkvæmnu lausninni okkar í ítraða lausn.
- ▶ Eina sem við þurfum að passa er að reikna gildin í vaxandi röð.

Neðansækin kvik bestun

- ▶ Það er þó lítið mál að breyta endurkvæmnu lausninni okkar í ítraða lausn.
- ▶ Eina sem við þurfum að passa er að reikna gildin í vaxandi röð.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int n, i;
6     scanf("%d", &n);
7     int a[n];
8     a[0] = a[1] = 1;
9     for (i = 2; i < n; i++) a[i] = a[i - 1] + a[i - 2];
10    printf("%d\n", a[n - 1]);
11    return 0;
12 }
```

Neðansækin kvik bestun

- ▶ Það er þó lítið mál að breyta endurkvæmnu lausninni okkar í ítraða lausn.
- ▶ Eina sem við þurfum að passa er að reikna gildin í vaxandi röð.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int n, i;
6     scanf("%d", &n);
7     int a[n];
8     a[0] = a[1] = 1;
9     for (i = 2; i < n; i++) a[i] = a[i - 1] + a[i - 2];
10    printf("%d\n", a[n - 1]);
11    return 0;
12 }
```

- ▶ Þegar ofansækin kvik bestunar lausn er útfærð með ítrun köllum við það *neðansækin kvik bestun* (e. *bottom up dymanic programming*).

- ▶ Í neðansækinni kvikri bestun byrjum við með grunntilfellin og smíðum flóknari lausnirnar út frá þeim.

- ▶ Í neðansækinni kvikri bestun byrjum við með grunntilfellin og smíðum flóknari lausnirnar út frá þeim.
- ▶ Í ofansækinni kvikri bestun brjótum við fyrst flóknu dæmin niður í smærri dæmi sem við vitum svarið við og reiknum svo út úr því.

- ▶ Í neðansækinni kvikri bestun byrjum við með grunntilfellin og smíðum flóknari lausnirnar út frá þeim.
- ▶ Í ofansækinni kvikri bestun brjótum við fyrst flóknu dæmin niður í smærri dæmi sem við vitum svarið við og reiknum svo út úr því.
- ▶ Ef endurkvæmnafallið okkar er háð k breytum þá segjum við að lausnin okkar sé k *víð kvik bestun*.

- ▶ Í neðansækinni kvikri bestun byrjum við með grunntilfellin og smíðum flóknari lausnirnar út frá þeim.
- ▶ Í ofansækinni kvikri bestun brjótum við fyrst flóknu dæmin niður í smærri dæmi sem við vitum svarið við og reiknum svo út úr því.
- ▶ Ef endurkvæmnafallið okkar er háð k breytum þá segjum við að lausnin okkar sé k víð kvik bestun.
- ▶ Ofansækin kvik bestun hentar þegar við erum að vinna með fleiri en eina vídd.

- ▶ Í neðansækinni kvikri bestun byrjum við með grunntilfellin og smíðum flóknari lausnirnar út frá þeim.
- ▶ Í ofansækinni kvikri bestun brjótum við fyrst flóknu dæmin niður í smærri dæmi sem við vitum svarið við og reiknum svo út úr því.
- ▶ Ef endurkvæmnafallið okkar er háð k breytum þá segjum við að lausnin okkar sé k víð kvik bestun.
- ▶ Ofansækin kvik bestun hentar þegar við erum að vinna með fleiri en eina vídd.
- ▶ Þá getur verið erfitt að ítra í gegnum stöðurnar í “rétttri röð”.

- ▶ Annar kostur ofansækinnar kvikrar bestunar er að lausnirnar geta verið nokkuð einsleitar.

- ▶ Annar kostur ofansækinna kvikra bestunar er að lausnirnar geta verið nokkuð einsleitar.

```
1 #include <stdio.h>
2 #define MAXN 1000000
3
4 int d[MAXN];
5 int dp_lookup(int x)
6 {
7     if (d[x] != -1) return d[x];
8     if (/* Er þetta grunntilfelli? */)
9     {
10         /* Skila tilheyrandi grunnsvari */
11     }
12     /* Reikna d[x] */
13     return d[x];
14 }
15
16 int main()
17 {
18     int n, i;
19     scanf("%d", &n);
20     for (i = 0; i < MAXN; i++) d[i] = -1;
21     printf("%d\n", dp_lookup(n));
22     return 0;
23 }
```

- ▶ Neðansækin kvik bestun hefur sýna kosti.

- ▶ Neðansækin kvik bestun hefur sýna kosti.
- ▶ Það er stundum hægt að bæta tímaflækjuna með til dæmis sniðugri gagnagrind.

- ▶ Neðansækin kvik bestun hefur sýna kosti.
- ▶ Það er stundum hægt að bæta tímaflækjuna með til dæmis sniðugri gagnagrind.
- ▶ Sumar þessara bætinga krefjast þess að útfærsla sé neðan sækin.

Lengsta sameiginlega hlutruna

- ▶ Tökum annað dæmi.

Lengsta sameiginlega hlutruna

- ▶ Tökum annað dæmi.
- ▶ Látum $S = s_1 s_2 \dots s_n$ og $T = t_1 t_2 \dots t_m$ vera strengi af lengd n og m , þannig að $1 \leq n, m \leq 10^3$.

Lengsta sameiginlega hlutruna

- ▶ Tökum annað dæmi.
- ▶ Látum $S = s_1 s_2 \dots s_n$ og $T = t_1 t_2 \dots t_m$ vera strengi af lengd n og m , þannig að $1 \leq n, m \leq 10^3$.
- ▶ Hver er lengd lengsta strengs X þannig að hann sé hlutruna í bæði S og T ?

Lengsta sameiginlega hlutruna

- ▶ Tökum annað dæmi.
- ▶ Látum $S = s_1s_2\dots s_n$ og $T = t_1t_2\dots t_m$ vera strengi af lengd n og m , þannig að $1 \leq n, m \leq 10^3$.
- ▶ Hver er lengd lengsta strengs X þannig að hann sé hlutruna í bæði S og T ?
- ▶ Takið eftir að "12" og "13" eru hlutrunur í "123" en "21" er það ekki.

- ▶ Getum við sett upp dæmið með þægilegum rakningarvenslum?

- ▶ Getum við sett upp dæmið með þægilegum rakningarvenslum?
- ▶ Ef svo er þá getum við notað kvika bestun.

- ▶ Getum við sett upp dæmið með þægilegum rakningarvenslum?
- ▶ Ef svo er þá getum við notað kvika bestun.
- ▶ Það er yfirleitt þægilegast að hugsa um rakningarvenslin sem fall, frekar en runu.

- ▶ Getum við sett upp dæmið með þægilegum rakningarvenslum?
- ▶ Ef svo er þá getum við notað kvika bestun.
- ▶ Það er yfirleitt þægilegast að hugsa um rakningarvenslin sem fall, frekar en runu.
- ▶ Látum $f(i, j)$ tákna lengstu sameiginlegu hlutrunu strengjanna $s_1 s_2 \dots s_i$ og $t_1 t_2 \dots t_j$.

- ▶ Getum við sett upp dæmið með þægilegum rakningarvenslum?
- ▶ Ef svo er þá getum við notað kvika bestun.
- ▶ Það er yfirleitt þægilegast að hugsa um rakningarvenslin sem fall, frekar en runu.
- ▶ Látum $f(i, j)$ tákna lengstu sameiginlegu hlutrunu strengjanna $s_1 s_2 \dots s_i$ og $t_1 t_2 \dots t_j$.
- ▶ Okkur mun svo nægja að reikna $f(n, m)$.

► Viď vitum aď $f(0, i) = f(j, 0) = 0$.

- ▶ Við vitum að $f(0, i) = f(j, 0) = 0$.
- ▶ Þetta munu vera grunntilfellin okkar.

- ▶ Við vitum að $f(0, i) = f(j, 0) = 0$.
- ▶ Þetta munu vera grunntilfellin okkar.
- ▶ Almennt gildir að ef við erum að reikna $f(i, j)$ og $s_i = t_j$ þá getum við látið þann staf vera aftastan í sameiginlegu hlutrununni.

- ▶ Við vitum að $f(0, i) = f(j, 0) = 0$.
- ▶ Þetta munu vera grunntilfellin okkar.
- ▶ Almennt gildir að ef við erum að reikna $f(i, j)$ og $s_i = t_j$ þá getum við látið þann staf vera aftastan í sameiginlegu hlutrununni.
- ▶ Svo $f(i, j) = f(i - 1, j - 1) + 1$ ef $s_i = t_j$.

- ▶ Við vitum að $f(0, i) = f(j, 0) = 0$.
- ▶ Þetta munu vera grunntilfellin okkar.
- ▶ Almennt gildir að ef við erum að reikna $f(i, j)$ og $s_i = t_j$ þá getum við látið þann staf vera aftastan í sameiginlegu hlutrununni.
- ▶ Svo $f(i, j) = f(i - 1, j - 1) + 1$ ef $s_i = t_j$.
- ▶ Ef $s_i \neq t_j$ þá verður annað stakið (eða bæði stökin) að vera ekki í hlutrununni.

- ▶ Við vitum að $f(0, i) = f(j, 0) = 0$.
- ▶ Þetta munu vera grunntilfellin okkar.
- ▶ Almennt gildir að ef við erum að reikna $f(i, j)$ og $s_i = t_j$ þá getum við látið þann staf vera aftastan í sameiginlegu hlutrununni.
- ▶ Svo $f(i, j) = f(i - 1, j - 1) + 1$ ef $s_i = t_j$.
- ▶ Ef $s_i \neq t_j$ þá verður annað stakið (eða bæði stökin) að vera ekki í hlutrununni.
- ▶ Við veljum að sjálfsgöðu að sleppa þeim sem gefur okkur betra svar, það er að segja $f(i, j) = \max(f(i - 1, j), f(i, j - 1))$.

- ▶ Við vitum að $f(0, i) = f(j, 0) = 0$.
- ▶ Þetta munu vera grunntilfellin okkar.
- ▶ Almennt gildir að ef við erum að reikna $f(i, j)$ og $s_i = t_j$ þá getum við látið þann staf vera aftastan í sameiginlegu hlutrununni.
- ▶ Svo $f(i, j) = f(i - 1, j - 1) + 1$ ef $s_i = t_j$.
- ▶ Ef $s_i \neq t_j$ þá verður annað stakið (eða bæði stökin) að vera ekki í hlutrununni.
- ▶ Við veljum að sjálfsögðu að sleppa þeim sem gefur okkur betra svar, það er að segja $f(i, j) = \max(f(i - 1, j), f(i, j - 1))$.
- ▶ Við getum svo sett allt saman og fengið

$$f(i, j) = \begin{cases} 0, & \text{ef } i = 0 \text{ eða } j = 0 \\ f(i - 1, j - 1) + 1, & \text{annars, og ef } s_i = t_j \\ \max(f(i - 1, j), f(i, j - 1)), & \text{annars.} \end{cases}$$

```

1 #include <stdio.h>
2 #include <string.h>
3 #define MAXN 1001
4 int max(int a, int b) { if (a < b) return b; return a; }
5
6 char s[10001], t[10001];
7 int d[MAXN][MAXN];
8 int dp_lookup(int x, int y)
9 {
10     if (d[x][y] != -1) return d[x][y];
11     if (x == 0 || y == 0) return 0;
12     if (s[x - 1] == t[y - 1]) return d[x][y] = dp_lookup(x - 1, y - 1) + 1;
13     return d[x][y] = max(dp_lookup(x - 1, y), dp_lookup(x, y - 1));
14 }
15
16 int main()
17 {
18     int n, m, i, j;
19     fgets(s, MAXN, stdin);
20     fgets(t, MAXN, stdin);
21     n = strlen(s) - 1;
22     m = strlen(t) - 1;
23     for (i = 0; i < n + 1; i++) for (j = 0; j < m + 1; j++) d[i][j] = -1;
24     printf("%d\n", dp_lookup(n, m));
25     return 0;
26 }

```

- Það er þessi virði að bera saman `dp_lookup(...)` fallið í forritinu og $f(i, j)$ af glærunni í framan.

$$f(i, j) = \begin{cases} 0, & \text{ef } i = 0 \text{ eða } j = 0 \\ f(i-1, j-1) + 1, & \text{annars, og ef } s_i = t_j \\ \max(f(i-1, j), f(i, j-1)), & \text{annars.} \end{cases}$$

```
8 int dp_lookup(int x, int y)
9 {
10     if (d[x][y] != -1) return d[x][y];
11     if (x == 0 || y == 0) return 0;
12     if (s[x-1] == t[y-1]) return d[x][y] = dp_lookup(x-1, y-1) + 1;
13     return d[x][y] = max(dp_lookup(x-1, y), dp_lookup(x, y-1));
14 }
```

- ▶ Forritið okkar þarf í versta falli að reikna öll möguleg gildi á $f(i, j)$, sem eru $(n + 1) \cdot (m + 1)$ talsins.

- ▶ Forritið okkar þarf í versta falli að reikna öll möguleg gildi á $f(i, j)$, sem eru $(n + 1) \cdot (m + 1)$ talsins.
- ▶ En hvert gildi má reikna í $\mathcal{O}(\quad)$ tíma.

- ▶ Forritið okkar þarf í versta falli að reikna öll möguleg gildi á $f(i, j)$, sem eru $(n + 1) \cdot (m + 1)$ talsins.
- ▶ En hvert gildi má reikna í $\mathcal{O}(1)$ tíma.

- ▶ Forritið okkar þarf í versta falli að reikna öll möguleg gildi á $f(i, j)$, sem eru $(n + 1) \cdot (m + 1)$ talsins.
- ▶ En hvert gildi má reikna í $\mathcal{O}(1)$ tíma.
- ▶ Svo forritið hefur tímaflækjuna $\mathcal{O}(\quad)$.

- ▶ Forritið okkar þarf í versta falli að reikna öll möguleg gildi á $f(i, j)$, sem eru $(n + 1) \cdot (m + 1)$ talsins.
- ▶ En hvert gildi má reikna í $\mathcal{O}(1)$ tíma.
- ▶ Svo forritið hefur tímaflækjuna $\mathcal{O}(n \cdot m)$.

Skiptimyndadæmið

- ▶ Skoðum aftur Skiptimyntadæmið úr síðust viku.

Skiptimyndadæmið

- ▶ Skoðum aftur Skiptimyntadæmið úr síðust viku.
- ▶ Þú ert með ótakmarkað magn af m mismunandi myntum.

Skiptimyndadæmið

- ▶ Skoðum aftur Skiptimyntadæmið úr síðust viku.
- ▶ Þú ert með ótakmarkað magn af m mismunandi myntum.
- ▶ Þær eru virði x_1, x_2, \dots, x_m . Til þæginda gerum við ráð fyrir því að $x_1 = 1$.

Skiptimyndadæmið

- ▶ Skoðum aftur Skiptimyntadæmið úr síðust viku.
- ▶ Þú ert með ótakmarkað magn af m mismunandi myntum.
- ▶ Þær eru virði x_1, x_2, \dots, x_m . Til þæginda gerum við ráð fyrir því að $x_1 = 1$.
- ▶ Hver er minnsti nauðsynlegi fjöldi af klinki ef þú vilt gefa n krónur til baka.

- Gerum ráð fyrir að við byrjum að gefa til baka x_j krónur.

- ▶ Gerum ráð fyrir að við byrjum að gefa til baka x_j krónur.
- ▶ Þá erum við búin að smækka dæmið niður í $n - x_j$.

- ▶ Gerum ráð fyrir að við byrjum að gefa til baka x_j krónur.
- ▶ Þá erum við búin að smækka dæmið niður í $n - x_j$.
- ▶ Við getum því skoðað öll mögulega gildi x_j og séð hvað er best.

- ▶ Gerum ráð fyrir að við byrjum að gefa til baka x_j krónur.
- ▶ Þá erum við búin að smækka dæmið niður í $n - x_j$.
- ▶ Við getum því skoðað öll mögulega gildi x_j og séð hvað er best.
- ▶ Við viljum því reikna gildin á fallinu

$$f(i) = \begin{cases} \infty, & \text{ef } i < 0 \\ 0, & \text{ef } i = 0 \\ \min_{j=1,2,\dots,m} f(i - x_j) + 1, & \text{annars.} \end{cases}$$


```

1 #include <stdio.h>
2 #define MAXN 10001
3 #define MAXM 10001
4 #define INF (1 << 30)
5 int min(int a, int b) { if (a < b) return a; return b; }
6
7 int n, m, a[MAXN];
8 int d[MAXN];
9 int dp_lookup(int x)
10 {
11     int i;
12     if (x < 0) return INF; // Þessi lína þarf að vera fremst!
13     if (d[x] != -1) return d[x];
14     if (x == 0) return 0;
15     d[x] = INF;
16     for (i = 0; i < m; i++) d[x] = min(d[x], dp_lookup(x - a[i]) + 1);
17     return d[x];
18 }
19
20 int main()
21 {
22     int i;
23     scanf("%d%d", &n, &m);
24     for (i = 0; i < n + 1; i++) d[i] = -1;
25     for (i = 0; i < m; i++) scanf("%d", &a[i]);
26     printf("%d\n", dp_lookup(n));
27     return 0;
28 }

```

- ▶ Þetta dæmi má þó hæglega gera neðansækið.

```

1 #include <stdio.h>
2 #define INF (1 << 30)
3 int min(int a, int b) { if (a < b) return a; return b; }
4
5 int main()
6 {
7     int i, j, n, m;
8     scanf("%d%d", &n, &m);
9     int d[n + 1], a[m];
10    for (i = 0; i < m; i++) scanf("%d", &a[i]);
11    for (i = 0; i < n + 1; i++) d[i] = INF;
12    d[0] = 0;
13    for (i = 0; i < m; i++)
14    {
15        for (j = 0; j < n + 1 - a[i]; j++) if (d[j] < INF)
16        {
17            d[j + a[i]] = min(d[j + a[i]], d[j] + 1);
18        }
19    }
20
21    printf("%d\n", d[n]);
22    return 0;
23 }

```

- ▶ Breytum dæminu örlítið.

- ▶ Breytum dæminu örlítið.
- ▶ Núna höfum við takmarkað magn af hverju klinki.

- ▶ Breytum dæminu örlítið.
- ▶ Núna höfum við takmarkað magn af hverju klinki.
- ▶ Nánar tiltekið höfum við m klink að andvirði x_1, x_2, \dots, x_m (núna geta verið endurtekin gildi).

- ▶ Breytum dæminu örlítið.
- ▶ Núna höfum við takmarkað magn af hverju klinki.
- ▶ Nánar tiltekið höfum við m klink að andvirði x_1, x_2, \dots, x_m (núna geta verið endurtekin gildi).
- ▶ Hver er minnsti fjöldi að klinki sem þarf til að gefa til baka n krónur, ef það er á annað borð hægt.

- ▶ Breytum dæminu örlítið.
- ▶ Núna höfum við takmarkað magn af hverju klinki.
- ▶ Nánar tiltekið höfum við m klink að andvirði x_1, x_2, \dots, x_m (núna geta verið endurtekin gildi).
- ▶ Hver er minnsti fjöldi að klinki sem þarf til að gefa til baka n krónur, ef það er á annað borð hægt.
- ▶ Nú er óþarfi að gera ráð fyrir því að $x_1 = 1$.

- ▶ Breytum dæminu örlítið.
- ▶ Núna höfum við takmarkað magn af hverju klinki.
- ▶ Nánar tiltekið höfum við m klink að andvirði x_1, x_2, \dots, x_m (núna geta verið endurtekin gildi).
- ▶ Hver er minnsti fjöldi að klinki sem þarf til að gefa til baka n krónur, ef það er á annað borð hægt.
- ▶ Nú er óþarfi að gera ráð fyrir því að $x_1 = 1$.
- ▶ Hvernig mætti breyta neðansæknu lausninni til að höndla þetta?

- ▶ Breytum dæminu örlítið.
- ▶ Núna höfum við takmarkað magn af hverju klinki.
- ▶ Nánar tiltekið höfum við m klink að andvirði x_1, x_2, \dots, x_m (núna geta verið endurtekin gildi).
- ▶ Hver er minnsti fjöldi að klinki sem þarf til að gefa til baka n krónur, ef það er á annað borð hægt.
- ▶ Nú er óþarfi að gera ráð fyrir því að $x_1 = 1$.
- ▶ Hvernig mætti breyta neðansæknu lausninni til að höndla þetta?
- ▶ Skoðum aftur neðansæknu lausnina.

Hefðbundna skiptimyntadæmið

```
1 #include <stdio.h>
2 #define INF (1 << 30)
3 int min(int a, int b) { if (a < b) return a; return b; }
4
5 int main()
6 {
7     int i, j, n, m;
8     scanf("%d%d", &n, &m);
9     int d[n + 1], a[m];
10    for (i = 0; i < m; i++) scanf("%d", &a[i]);
11    for (i = 0; i < n + 1; i++) d[i] = INF;
12    d[0] = 0;
13    for (i = 0; i < m; i++)
14    {
15        for (j = 0; j < n + 1 - a[i]; j++) if (d[j] < INF)
16        {
17            d[j + a[i]] = min(d[j + a[i]], d[j] + 1);
18        }
19    }
20
21    printf("%d\n", d[n]);
22    return 0;
23 }
```

Nýja dæmið

```
1 #include <stdio.h>
2 #define INF (1 << 30)
3 int min(int a, int b) { if (a < b) return a; return b; }
4
5 int main()
6 {
7     int i, j, n, m;
8     scanf("%d%d", &n, &m);
9     int d[n + 1], a[m];
10    for (i = 0; i < m; i++) scanf("%d", &a[i]);
11    for (i = 0; i < n + 1; i++) d[i] = INF;
12    d[0] = 0;
13    for (i = 0; i < m; i++)
14    {
15        for (j = n - a[i]; j >= 0; j--) if (d[j] < INF)
16        {
17            d[j + a[i]] = min(d[j + a[i]], d[j] + 1);
18        }
19    }
20
21    printf("%d\n", d[n]);
22    return 0;
23 }
```

- ▶ Skoðum báðar aðferðirnar á litlu sýnidæmi.

- ▶ Skoðum báðar aðferðirnar á litlu sýnidæmi.
- ▶ Skoðum fyrst með endurtekningum og síðan án endurtekninga.

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]  
    ^   |
```

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0   1   2   3   4   5   6   7   8   9  10  
d = [0,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1]  
    ^   |
```



```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1]  
      ^    |
```

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2, -1, -1, -1, -1, -1, -1, -1, -1]  
      ^    |
```

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2, -1, -1, -1, -1, -1, -1, -1, -1]  
      ^    |
```

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2,  3, -1, -1, -1, -1, -1, -1, -1]
```

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,   1,   2,   3, -1, -1, -1, -1, -1, -1, -1]  
      ^      |
```

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2,  3,  4, -1, -1, -1, -1, -1, -1]
```

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,   1,   2,   3,   4,  -1,  -1,  -1,  -1,  -1,  -1]  
      ^      |
```

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2,  3,  4,  5, -1, -1, -1, -1, -1]  
      ^      |
```



```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9    10  
d = [0,  1,  2,  3,  4,  5, -1, -1, -1, -1, -1]  
      ^      |
```

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2,  3,  4,  5,  6, -1, -1, -1, -1]  
      ^    |
```

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9    10
d = [0,  1,  2,  3,  4,  5,  6, -1, -1, -1, -1]
```

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	3,	4,	5,	6,	7,	-1,	-1,	-1]
							^				

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,   1,   2,   3,   4,   5,   6,   7,  -1,  -1,  -1]  
                                ^    |
```

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2,  3,  4,  5,  6,  7,  8, -1, -1]
```

```
n = 10
```

```
a = [1, 3, 5]
```

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	3,	4,	5,	6,	7,	8,	-1,	-1]
									^		

```
n = 10
```

```
a = [1, 3, 5]
```

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	-1]
									^		


```
n = 10
```

```
a = [1, 3, 5]
```

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	-1]
										^	

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9    10
d = [0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
```

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
```

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2,  1,  4,  5,  6,  7,  8,  9, 10]
```

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	4,	5,	6,	7,	8,	9,	10]
		^									

```
n = 10
```

```
a = [1, 3, 5]
```

^

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2,  1,  2,  5,  6,  7,  8,  9, 10]
```

^

|

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2,  1,  2,  5,  6,  7,  8,  9, 10]
```

```
n = 10
```

```
a = [1, 3, 5]
```

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	3,	6,	7,	8,	9,	10]
			^								


```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2,  1,  2,  3,  6,  7,  8,  9, 10]
```

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	3,	2,	7,	8,	9,	10]
				^							

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2,  1,  2,  3,  2,  7,  8,  9, 10]
```

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	3,	2,	3,	8,	9,	10]
					^						

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9    10
d = [0,  1,  2,  1,  2,  3,  2,  3,  8,  9, 10]
```

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9    10
d = [0,  1,  2,  1,  2,  3,  2,  3,  4,  9, 10]
```

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9    10
d = [0,  1,  2,  1,  2,  3,  2,  3,  4,  9, 10]
```

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	3,	2,	3,	4,	3,	10]
							^				


```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	3,	2,	3,	4,	3,	10]
								^			

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	3,	2,	3,	4,	3,	4]
								^			

```
n = 10
```

```
a = [1, 3, 5]  
      ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1,  2,  1,  2,  3,  2,  3,  4,  3,  4]  
      ^                               |
```

```
n = 10
```

```
a = [1, 3, 5]
```

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	1,	2,	3,	4,	3,	4]
	^										

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	1,	2,	3,	4,	3,	4]

^

|

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	1,	2,	3,	4,	3,	4]
		^									

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	1,	2,	3,	4,	3,	4]
			^								

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	1,	2,	3,	4,	3,	4]
			^								


```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	1,	2,	3,	4,	3,	4]
				^							

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	1,	2,	3,	2,	3,	4]
				^							

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	1,	2,	3,	2,	3,	4]
					^						

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	1,	2,	3,	2,	3,	4]
					^						

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	1,	2,	3,	2,	3,	4]
						^					

```
n = 10
```

```
a = [1, 3, 5]  
      ^
```

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	1,	2,	3,	2,	3,	2]
						^					

```
n = 10
```

```
a = [1, 3, 5]
```

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	2,	1,	2,	1,	2,	3,	2,	3,	2]

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9    10  
d = [0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]  
                                     ^    |
```



```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1]
									^		

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9    10
d = [0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1]
							^				

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]  
      ^      |
```

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]  
      ^      |
```

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1]
				^							

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0   1   2   3   4   5   6   7   8   9  10  
d = [0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]  
      ^   |
```



```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0   1   2   3   4   5   6   7   8   9  10  
d = [0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]  
    ^   |
```

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9    10  
d = [0,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1]  
    ^   |
```

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1]
								^			

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1]
							^				

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1]
						^					

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1]
					^						

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1]
				^							

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1,	-1]
			^								


```
n = 10
```

```
a = [1, 3, 5]
```

^

```
      0    1    2    3    4    5    6    7    8    9    10  
d = [0,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

^

|

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	-1,	2,	-1,	-1,	-1,	-1,	-1,	-1]
		^									

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	-1,	2,	-1,	-1,	-1,	-1,	-1,	-1]
	^										

```
n = 10
```

```
a = [1, 3, 5]  
    ^
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1, -1,  1,  2, -1, -1, -1, -1, -1, -1]  
    ^                |
```

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	1,	2,	-1,	-1,	-1,	-1,	-1,	-1]
						^					

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	1,	2,	-1,	-1,	-1,	-1,	-1,	-1]
					^						

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9    10
d = [0,  1, -1,  1,  2, -1, -1, -1, -1,  3, -1]
```

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	1,	2,	-1,	-1,	-1,	-1,	3,	-1]
				^							


```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	1,	2,	-1,	-1,	-1,	2,	3,	-1]
				^							

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9    10  
d = [0,  1, -1,  1,  2, -1, -1, -1,  2,  3, -1]
```

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1, -1,  1,  2, -1, -1, -1,  2,  3, -1]
```

```
n = 10
```

```
a = [1, 3, 5]
```

^

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	1,	2,	-1,	2,	-1,	2,	3,	-1]
		^									

```
n = 10
```

```
a = [1, 3, 5]
```

```
      0    1    2    3    4    5    6    7    8    9   10  
d = [0,  1, -1,  1,  2, -1,  2, -1,  2,  3, -1]
```

```
n = 10
```

```
a = [1, 3, 5]
```

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	1,	2,	1,	2,	-1,	2,	3,	-1]
	^										

```
n = 10
```

```
a = [1, 3, 5]
```

	0	1	2	3	4	5	6	7	8	9	10
d =	[0,	1,	-1,	1,	2,	1,	2,	-1,	2,	3,	-1]

- ▶ Hvernig myndum við þó leysa seinna dæmið með ofansækinni kvikri bestun?

- ▶ Hvernig myndum við þó leysa seinna dæmið með ofansækinni kvikri bestun?
- ▶ Við þurfum að hugsa það aðeins öðruvísi.

- ▶ Hvernig myndum við þó leysa seinna dæmið með ofansækinni kvikri bestun?
- ▶ Við þurfum að hugsa það aðeins öðruvísi.
- ▶ Nú höfum við um tvennt að velja fyrir hvern pening.

- ▶ Hvernig myndum við þó leysa seinna dæmið með ofansækinni kvikri bestun?
- ▶ Við þurfum að hugsa það aðeins öðruvísi.
- ▶ Nú höfum við um tvennt að velja fyrir hvern pening.
- ▶ Annaðhvort notum við hann, eða ekki.

- ▶ Hvernig myndum við þó leysa seinna dæmið með ofansækinni kvikri bestun?
- ▶ Við þurfum að hugsa það aðeins öðruvísi.
- ▶ Nú höfum við um tvennt að velja fyrir hvern pening.
- ▶ Annaðhvort notum við hann, eða ekki.
- ▶ Svo við látum $f(n, j)$ tákna minnsta fjölda af klinki sem þarf til að gefa til baka n krónur, ef við megum nota klink x_j, x_{j+1}, \dots, x_m .

- ▶ Hvernig myndum við þó leysa seinna dæmið með ofansækinni kvikri bestun?
- ▶ Við þurfum að hugsa það aðeins öðruvísi.
- ▶ Nú höfum við um tvennt að velja fyrir hvern pening.
- ▶ Annaðhvort notum við hann, eða ekki.
- ▶ Svo við látum $f(n, j)$ tákna minnsta fjölda af klinki sem þarf til að gefa til baka n krónur, ef við megum nota klink x_j, x_{j+1}, \dots, x_m .
- ▶ Þá fáum við að

$$f(i, j) = \begin{cases} \infty, & \text{ef } i < 0 \\ \infty, & \text{ef } i \neq 0 \text{ og } j = m + 1 \\ 0, & \text{ef } i = 0 \text{ og } j = m + 1 \\ \min(f(i, j + 1), & \\ f(i - x_j, j + 1) + 1), & \text{annars.} \end{cases}$$

```

1 #include <stdio.h>
2 #define MAXN 10001
3 #define MAXM 10001
4 #define INF (1 << 30)
5 int min(int a, int b) { if (a < b) return a; return b; }
6
7 int n, m, a[MAXM];
8 int d[MAXN][MAXM];
9 int dp_lookup(int x, int y)
10 {
11     if (x < 0) return INF;
12     if (d[x][y] != -1) return d[x][y];
13     if (y == m) return x == 0 ? 0 : INF;
14     return d[x][y] = min(dp_lookup(x, y + 1),
15                          dp_lookup(x - a[y], y + 1) + 1);
16 }
17
18 int main()
19 {
20     int i, j;
21     scanf("%d%d", &n, &m);
22     for (i = 0; i < n + 1; i++) for (j = 0; j < m + 1; j++) d[i][j] = -1;
23     for (i = 0; i < m; i++) scanf("%d", &a[i]);
24     printf("%d\n", dp_lookup(n, 0));
25     for (i = 0; i < n + 1; i++)
26         printf("%3d ", dp_lookup(i, 0) == INF ? -1 : dp_lookup(i, 0));
27     printf("\n");
28     return 0;
29 }

```

- ▶ Það er nokkuð létt að meta tímaflækjurnar á neðansæknu lausnunum.

- ▶ Það er nokkuð létt að meta tímaflækjurnar á neðansæknu lausnunum.
- ▶ Þær eru báðar tvöfaldar for-lykkjur, sú ytri af lengd m og innri af lengd $\mathcal{O}(n)$.

- ▶ Það er nokkuð létt að meta tímaflækjurnar á neðansæknu lausnunum.
- ▶ Þær eru báðar tvöfaldar for-lykkjur, sú ytri af lengd m og innri af lengd $\mathcal{O}(n)$.
- ▶ Svo tímaflækjurnar eru $\mathcal{O}(\quad)$.

- ▶ Það er nokkuð létt að meta tímaflækjurnar á neðansæknu lausnunum.
- ▶ Þær eru báðar tvöfaldar for-lykkjur, sú ytri af lengd m og innri af lengd $\mathcal{O}(n)$.
- ▶ Svo tímaflækjurnar eru $\mathcal{O}(n \cdot m)$.

- Í ofansæknu lausninni á hefðbundna dæminu þarf að reikna, allt að, $n + 1$ fallgildi.

```
9 int dp_lookup(int x)
10 {
11     int i;
12     if (x < 0) return INF; // Þessi lína þarf að vera fremst!
13     if (d[x] != -1) return d[x];
14     if (x == 0) return 0;
15     d[x] = INF;
16     for (i = 0; i < m; i++) d[x] = min(d[x], dp_lookup(x - a[i]) + 1);
17     return d[x];
18 }
```

- ▶ Í ofansæknu lausninni á hefðbundna dæminu þarf að reikna, allt að, $n + 1$ fallgildi.
- ▶ Hvert gildi má reikna í $\mathcal{O}(\quad)$ tíma.

```
9 int dp_lookup(int x)
10 {
11     int i;
12     if (x < 0) return INF; // Þessi lína þarf að vera fremst!
13     if (d[x] != -1) return d[x];
14     if (x == 0) return 0;
15     d[x] = INF;
16     for (i = 0; i < m; i++) d[x] = min(d[x], dp_lookup(x - a[i]) + 1);
17     return d[x];
18 }
```

- ▶ Í ofansæknu lausninni á hefðbundna dæminu þarf að reikna, allt að, $n + 1$ fallgildi.
- ▶ Hvert gildi má reikna í $\mathcal{O}(m)$ tíma.

```
9 int dp_lookup(int x)
10 {
11     int i;
12     if (x < 0) return INF; // Þessi lína þarf að vera fremst!
13     if (d[x] != -1) return d[x];
14     if (x == 0) return 0;
15     d[x] = INF;
16     for (i = 0; i < m; i++) d[x] = min(d[x], dp_lookup(x - a[i]) + 1);
17     return d[x];
18 }
```

- ▶ Í ofansæknu lausninni á hefðbundna dæminu þarf að reikna, allt að, $n + 1$ fallgildi.
- ▶ Hvert gildi má reikna í $\mathcal{O}(m)$ tíma.
- ▶ Svo í heildina er hún $\mathcal{O}(\quad)$.

```
9 int dp_lookup(int x)
10 {
11     int i;
12     if (x < 0) return INF; // Þessi lína þarf að vera fremst!
13     if (d[x] != -1) return d[x];
14     if (x == 0) return 0;
15     d[x] = INF;
16     for (i = 0; i < m; i++) d[x] = min(d[x], dp_lookup(x - a[i]) + 1);
17     return d[x];
18 }
```

- ▶ Í ofansæknu lausninni á hefðbundna dæminu þarf að reikna, allt að, $n + 1$ fallgildi.
- ▶ Hvert gildi má reikna í $\mathcal{O}(m)$ tíma.
- ▶ Svo í heildina er hún $\mathcal{O}(n \cdot m)$.

```
9 int dp_lookup(int x)
10 {
11     int i;
12     if (x < 0) return INF; // Þessi lína þarf að vera fremst!
13     if (d[x] != -1) return d[x];
14     if (x == 0) return 0;
15     d[x] = INF;
16     for (i = 0; i < m; i++) d[x] = min(d[x], dp_lookup(x - a[i]) + 1);
17     return d[x];
18 }
```

- Í ofansæknu lausninni á hinu dæminu þarf að reikna, allt að, $(n + 1) \cdot (m + 1)$ fallgildi.

```
9 int dp_lookup(int x, int y)
10 {
11     if (x < 0) return INF;
12     if (d[x][y] != -1) return d[x][y];
13     if (y == m) return x == 0 ? 0 : INF;
14     return d[x][y] = min(dp_lookup(x, y + 1),
15                          dp_lookup(x - a[y], y + 1) + 1);
16 }
```


- ▶ Í ofansæknu lausninni á hinu dæminu þarf að reikna, allt að, $(n + 1) \cdot (m + 1)$ fallgildi.
- ▶ Hvert gildi má þó reikna í $\mathcal{O}(\quad)$ tíma.

```
9 int dp_lookup(int x, int y)
10 {
11     if (x < 0) return INF;
12     if (d[x][y] != -1) return d[x][y];
13     if (y == m) return x == 0 ? 0 : INF;
14     return d[x][y] = min(dp_lookup(x, y + 1),
15                          dp_lookup(x - a[y], y + 1) + 1);
16 }
```

- ▶ Í ofansæknu lausninni á hinu dæminu þarf að reikna, allt að, $(n + 1) \cdot (m + 1)$ fallgildi.
- ▶ Hvert gildi má þó reikna í $\mathcal{O}(1)$ tíma.

```
9 int dp_lookup(int x, int y)
10 {
11     if (x < 0) return INF;
12     if (d[x][y] != -1) return d[x][y];
13     if (y == m) return x == 0 ? 0 : INF;
14     return d[x][y] = min(dp_lookup(x, y + 1),
15                          dp_lookup(x - a[y], y + 1) + 1);
16 }
```

- ▶ Í ofansæknu lausninni á hinu dæminu þarf að reikna, allt að, $(n + 1) \cdot (m + 1)$ fallgildi.
- ▶ Hvert gildi má þó reikna í $\mathcal{O}(1)$ tíma.
- ▶ Svo í heildina er hún $\mathcal{O}(\quad)$.

```
9 int dp_lookup(int x, int y)
10 {
11     if (x < 0) return INF;
12     if (d[x][y] != -1) return d[x][y];
13     if (y == m) return x == 0 ? 0 : INF;
14     return d[x][y] = min(dp_lookup(x, y + 1),
15                          dp_lookup(x - a[y], y + 1) + 1);
16 }
```

- ▶ Í ofansæknu lausninni á hinu dæminu þarf að reikna, allt að, $(n + 1) \cdot (m + 1)$ fallgildi.
- ▶ Hvert gildi má þó reikna í $\mathcal{O}(1)$ tíma.
- ▶ Svo í heildina er hún $\mathcal{O}(n \cdot m)$.

```
9 int dp_lookup(int x, int y)
10 {
11     if (x < 0) return INF;
12     if (d[x][y] != -1) return d[x][y];
13     if (y == m) return x == 0 ? 0 : INF;
14     return d[x][y] = min(dp_lookup(x, y + 1),
15                          dp_lookup(x - a[y], y + 1) + 1);
16 }
```

- ▶ Hvað gerum við ef við viljum vita *hvaða* klink á að gefa til baka, ekki bara hversu mikið?

- ▶ Hvað gerum við ef við viljum vita *hvaða* klink á að gefa til baka, ekki bara hversu mikið?
- ▶ Það er yfirleitt farið aðra af tveimur leiðum.

- ▶ Hvað gerum við ef við viljum vita *hvaða* klink á að gefa til baka, ekki bara hversu mikið?
- ▶ Það er yfirleitt farið aðra af tveimur leiðum.
- ▶ Takið eftir að þegar við reiknum, til dæmis, $\min(dp(x, y + 1), dp(x - a[y], y + 1) + 1)$ þá erum við í raun að velja hvort er betra: $dp(x, y + 1)$ eða $dp(x - a[y], y + 1) + 1$.

- ▶ Hvað gerum við ef við viljum vita *hvaða* klink á að gefa til baka, ekki bara hversu mikið?
- ▶ Það er yfirleitt farið aðra af tveimur leiðum.
- ▶ Takið eftir að þegar við reiknum, til dæmis, $\min(\text{dp}(x, y + 1), \text{dp}(x - a[y], y + 1) + 1)$ þá erum við í raun að velja hvort er betra: $\text{dp}(x, y + 1)$ eða $\text{dp}(x - a[y], y + 1) + 1$.
- ▶ Fyrri aðferðin felur í sér að geyma fyrir hvert inntak í `dp_lookup(...)` hver besta leiðin er.

- ▶ Hvað gerum við ef við viljum vita *hvaða* klink á að gefa til baka, ekki bara hversu mikið?
- ▶ Það er yfirleitt farið aðra af tveimur leiðum.
- ▶ Takið eftir að þegar við reiknum, til dæmis, $\min(\text{dp}(x, y + 1), \text{dp}(x - a[y], y + 1) + 1)$ þá erum við í raun að velja hvort er betra: $\text{dp}(x, y + 1)$ eða $\text{dp}(x - a[y], y + 1) + 1$.
- ▶ Fyrri aðferðin felur í sér að geyma fyrir hvert inntak í $\text{dp_lookup}(\dots)$ hver besta leiðin er.
- ▶ Kvik bestun byggir á því að besta leiðin sé alltaf sú sama.

- ▶ Hvað gerum við ef við viljum vita *hvaða* klink á að gefa til baka, ekki bara hversu mikið?
- ▶ Það er yfirleitt farið aðra af tveimur leiðum.
- ▶ Takið eftir að þegar við reiknum, til dæmis, $\min(\text{dp}(x, y + 1), \text{dp}(x - a[y], y + 1) + 1)$ þá erum við í raun að velja hvort er betra: $\text{dp}(x, y + 1)$ eða $\text{dp}(x - a[y], y + 1) + 1$.
- ▶ Fyrri aðferðin felur í sér að geyma fyrir hvert inntak í $\text{dp_lookup}(\dots)$ hver besta leiðin er.
- ▶ Kvik bestun byggir á því að besta leiðin sé alltaf sú sama.
- ▶ Síðan er eftir á hægt að þræða sig í gegn og finna klinkið sem þarf.

Finnur bara fjöldann

```
1 #include <stdio.h>
2 #define INF (1 << 30)
3 int min(int a, int b) { if (a < b) return a; return b; }
4
5 int main()
6 {
7     int i, j, n, m, x;
8     scanf("%d%d", &n, &m);
9     int d[n + 1], a[m];
10    for (i = 0; i < m; i++) scanf("%d", &a[i]);
11    for (i = 0; i < n + 1; i++) d[i] = INF;
12    d[0] = 0;
13    for (i = 0; i < m; i++)
14        for (j = 0; j < n + 1 - a[i]; j++)
15            if (d[j] < INF && d[j + a[i]] > d[j] + 1)
16                {
17                    d[j + a[i]] = d[j] + 1;
18                }
19
20    printf("%d\n", d[n]);
21
22
23
24
25
26
27
28    return 0;
29 }
30 }
```

Finnur hvaða klink

```
1 #include <stdio.h>
2 #define INF (1 << 30)
3 int min(int a, int b) { if (a < b) return a; return b; }
4
5 int main()
6 {
7     int i, j, n, m, x;
8     scanf("%d%d", &n, &m);
9     int d[n + 1], a[m], e[n + 1];
10    for (i = 0; i < m; i++) scanf("%d", &a[i]);
11    for (i = 0; i < n + 1; i++) d[i] = INF;
12    d[0] = 0;
13    for (i = 0; i < m; i++)
14        for (j = 0; j < n + 1 - a[i]; j++)
15            if (d[j] < INF && d[j + a[i]] > d[j] + 1)
16            {
17                d[j + a[i]] = d[j] + 1;
18                e[j + a[i]] = a[i];
19            }
20
21    printf("%d\n", d[n]);
22    x = n;
23    while (x != 0)
24    {
25        printf("%d ", e[x]);
26        x -= e[x];
27    }
28    printf("\n");
29    return 0;
30 }
```

- ▶ Hin aðferðin hentar oft betur ef við erum með ofansækina kvika bestun.

- ▶ Hin aðferðin hentar oft betur ef við erum með ofansækina kvika bestun.
- ▶ Þá búum við til annað endurkvæmt fall sem notar `dp_lookup(...)` til að finna besta skrefið.

- ▶ Hin aðferðin hentar oft betur ef við erum með ofansækina kvika bestun.
- ▶ Þá búum við til annað endurkvæmt fall sem notar `dp_lookup(...)` til að finna besta skrefið.
- ▶ Þetta fall er auðvelt að smíða því það mun vera næstum eins og `dp_lookup(...)`.

- ▶ Hin aðferðin hentar oft betur ef við erum með ofansækina kvika bestun.
- ▶ Þá búum við til annað endurkvæmt fall sem notar `dp_lookup(...)` til að finna besta skrefið.
- ▶ Þetta fall er auðvelt að smíða því það mun vera næstum eins og `dp_lookup(...)`.
- ▶ Þegar það er búið að finna besta gildið prentar það hvert gildið er, og heldur svo áfram endurkvæmt.


```

1 #include <stdio.h>
2 #define MAXN 10001
3 #define MAXM 10001
4 #define INF (1 << 30)
5 int min(int a, int b) { if (a < b) return a; return b; }
6 int n, m, a[MAXN], d[MAXN];
7 int dp_lookup(int x)
8 {
9     int i;
10    if (x < 0) return INF; // Þessi lína þarf að vera fremst!
11    if (d[x] != -1) return d[x];
12    if (x == 0) return 0;
13    d[x] = INF;
14    for (i = 0; i < m; i++) d[x] = min(d[x], dp_lookup(x - a[i]) + 1);
15    return d[x];
16 }
17 int dp_traverse(int x)
18 {
19     if (x < 0) return INF;
20     if (x == 0) return 0;
21     int i, mn = INF, mni;
22     for (i = 0; i < m; i++) if (mn > dp_lookup(x - a[i]) + 1)
23         mn = dp_lookup(x - a[i]) + 1, mni = i;
24     printf("%d ", a[mni]), dp_traverse(x - a[mni]);
25     return mn;
26 }
27 int main()
28 {
29     int i;
30     scanf("%d%d", &n, &m);
31     for (i = 0; i < n + 1; i++) d[i] = -1;
32     for (i = 0; i < m; i++) scanf("%d", &a[i]);
33     printf("%d\n", dp_lookup(n));
34     dp_traverse(n);
35     printf("\n");
36     return 0;
37 }

```

- ▶ Helsti kostur fyrri aðferðarinnar er að besta skrefið er ákvarðað í $\mathcal{O}(n)$ tíma.

- ▶ Helsti kostur fyrri aðferðarinnar er að besta skrefið er ákvarðað í $\mathcal{O}(1)$ tíma.

- ▶ Helsti kostur fyrri aðferðarinnar er að besta skrefið er ákvarðað í $\mathcal{O}(1)$ tíma.
- ▶ Í seinni aðferðinni tekur það jafnalangan tíma og `dp_lookup(...)` tekur að meta hverja stöðu.

- ▶ Helsti kostur fyrri aðferðarinnar er að besta skrefið er ákvarðað í $\mathcal{O}(1)$ tíma.
- ▶ Í seinni aðferðinni tekur það jafnalangan tíma og `dp_lookup(...)` tekur að meta hverja stöðu.
- ▶ Þetta má þó bæta með minnun.

- ▶ Helsti kostur fyrri aðferðarinnar er að besta skrefið er ákvarðað í $\mathcal{O}(1)$ tíma.
- ▶ Í seinni aðferðinni tekur það jafnalangan tíma og `dp_lookup(...)` tekur að meta hverja stöðu.
- ▶ Þetta má þó bæta með minnun.
- ▶ Þetta kemur bara til með að gera nógu góða lausn hæga ef það þarf að reikna fyrir mörg n .

- ▶ Helsti kostur fyrri aðferðarinnar er að besta skrefið er ákvarðað í $\mathcal{O}(1)$ tíma.
- ▶ Í seinni aðferðinni tekur það jafnalangan tíma og `dp_lookup(...)` tekur að meta hverja stöðu.
- ▶ Þetta má þó bæta með minnun.
- ▶ Þetta kemur bara til með að gera nógu góða lausn hæga ef það þarf að reikna fyrir mörg n .
- ▶ Skoðum nú hvernig við gætum nýtt þetta til að finna lengsta sameiginlega hlutstreng.

```

1 #include <stdio.h>
2 #include <string.h>
3 #define MAXN 1001
4 int max(int a, int b) { if (a < b) return b; return a; }
5 char s[MAXN], t[MAXN];
6 int d[MAXN][MAXN];
7 int dp_lookup(int x, int y)
8 {
9     if (d[x][y] != -1) return d[x][y];
10    if (x == 0 || y == 0) return 0;
11    if (s[x - 1] == t[y - 1]) return d[x][y] = dp_lookup(x - 1, y - 1) + 1;
12    return d[x][y] = max(dp_lookup(x - 1, y), dp_lookup(x, y - 1));
13 }
14
15 void dp_traverse(int x, int y)
16 {
17     if (x == 0 || y == 0) return;
18     if (s[x - 1] == t[y - 1])
19     {
20         dp_traverse(x - 1, y - 1);
21         printf("%c", s[x - 1]);
22     }
23     else if (dp_lookup(x - 1, y) > dp_lookup(x, y - 1)) dp_traverse(x - 1, y);
24     else dp_traverse(x, y - 1);
25 }
26
27 int main()
28 {
29     int n, m, i, j;
30     fgets(s, MAXN, stdin), fgets(t, MAXN, stdin);
31     n = strlen(s) - 1, m = strlen(t) - 1;
32     for (i = 0; i < n + 1; i++) for (j = 0; j < m + 1; j++) d[i][j] = -1;
33     dp_traverse(n, m);
34     printf("\n");
35     return 0;
36 }

```


- ▶ Seinni skiptadæmið er náskylt *hlutmengjasummudæminu* (e. *Subset Sum Problem*).

- ▶ Seinni skiptadæmið er náskylt *hlutmengjasummudæminu* (e. *Subset Sum Problem*).
- ▶ Dæmið er einfalt.

- ▶ Seinni skiptadæmið er náskylt *hlutmengjasummudæminu* (e. *Subset Sum Problem*).
- ▶ Dæmið er einfalt.
- ▶ Gefnar eru n tölur a_1, \dots, a_n ásamt tölu c .

- ▶ Seinni skiptadæmið er náskyld *hlutmengjasummudæminu* (e. *Subset Sum Problem*).
- ▶ Dæmið er einfalt.
- ▶ Gefnar eru n tölur a_1, \dots, a_n ásamt tölu c .
- ▶ Hvaða hlutruna af tölu gefur hæstu summun án þess að fara yfir c .

c = 15

b = 0

a: 1 2 7 9

s: 0

```
a: 1    2    7    9
    ^
s: 1

c = 15
b = 1
```

```
a: 1  2  7  9
    ^
s: 2

c = 15
b = 2
```

```
a: 1    2    7    9
    ^
s: 7

c = 15
b = 7
```



```
a: 1    2    7    9
    ^
s: 9

c = 15
b = 9
```

```
a: 1    2    7    9
    ^    ^
s: 3

c = 15
b = 9
```

```
a: 1    2    7    9
    ^      ^
s: 8
```

c = 15
b = 9

```
a: 1    2    7    9
    ^      ^
s: 10
```

c = 15
b = 10

c = 15

b = 10

a: 1 2 7 9
 ^ ^

s: 9

```
a: 1    2    7    9
    ^      ^
s: 11

c = 15
b = 11
```

```
a: 1    2    7    9
    ^    ^
s: 16
```

c = 15
b = 11

```
a: 1    2    7    9
    ^    ^    ^
s: 10
```

c = 15
b = 11


```
a: 1    2    7    9
    ^    ^
s: 12
```

c = 15
b = 12

```
a: 1    2    7    9
    ^      ^      ^
s: 17
```

```
c = 15
b = 12
```

```
a: 1    2    7    9
    ^    ^    ^
s: 18

c = 15
b = 12
```

```
a: 1    2    7    9
    ^    ^    ^    ^
s: 19
```

c = 15
b = 12

► Látum

$$f(i, j) = \begin{cases} 1, & \text{ef til er hlutruna af } a_1, \dots, a_i \text{ sem hefur summu } j, \\ 0, & \text{annars.} \end{cases}$$

► Látum

$$f(i, j) = \begin{cases} 1, & \text{ef til er hlutruna af } a_1, \dots, a_i \text{ sem hefur summu } j, \\ 0, & \text{annars.} \end{cases}$$

► Við getum nú umritað

$$f(i, j) = \begin{cases} 1, & \text{ef } i = 0 \text{ og } j = 0, \\ 0, & \text{ef } i = 0 \text{ og } j \neq 0, \text{ eða } j < 0, \\ \min(1, f(i-1, j) \\ \quad + f(i-1, j-a_i)), & \text{ef } i \neq 0. \end{cases}$$

```

7 int d[MAXN][MAXC], b[MAXN];
8 int dp_lookup(int x, int y)
9 {
10     if (x < 0 && y == 0) return 1;
11     if (y < 0 || x < 0) return 0;
12     if (d[x][y] != -1) return d[x][y];
13     return d[x][y] = dp_lookup(x - 1, y) || dp_lookup(x - 1, y - b[x]);
14 }
15
16 int subsetsum(int *a, int n, int c)
17 {
18     int i, j;
19     for (i = 0; i < n; i++) for (j = 0; j < c + 1; j++) d[i][j] = -1;
20     for (i = 0; i < n; i++) b[i] = a[i];
21     while (!dp_lookup(n - 1, c)) c--;
22     return c;
23 }

```

- ▶ Við reiknum hvert gildi á $f(i,j)$ í $\mathcal{O}(1)$.

- ▶ Við reiknum hvert gildi á $f(i, j)$ í $\mathcal{O}(1)$.

- ▶ Við reiknum hvert gildi á $f(i, j)$ í $\mathcal{O}(1)$.
- ▶ Við þurfum í versta falli að reinka $n \cdot (c + 1)$ slíka gildi.

- ▶ Við reiknum hvert gildi á $f(i, j)$ í $\mathcal{O}(1)$.
- ▶ Við þurfum í versta falli að reinka $n \cdot (c + 1)$ slíka gildi.
- ▶ Svo það tekur okkur $\mathcal{O}(\quad)$ tíma að reikna $f(i, j)$.

- ▶ Við reiknum hvert gildi á $f(i, j)$ í $\mathcal{O}(1)$.
- ▶ Við þurfum í versta falli að reinka $n \cdot (c + 1)$ slíka gildi.
- ▶ Svo það tekur okkur $\mathcal{O}(n \cdot c)$ tíma að reikna $f(i, j)$.

- ▶ Við reiknum hvert gildi á $f(i, j)$ í $\mathcal{O}(1)$.
- ▶ Við þurfum í versta falli að reinka $n \cdot (c + 1)$ slíka gildi.
- ▶ Svo það tekur okkur $\mathcal{O}(n \cdot c)$ tíma að reikna $f(i, j)$.
- ▶ En sökum minnunar tekura það $\mathcal{O}(q + n \cdot c)$ tíma að reikna $f(i, j)$ q sinnum.

- ▶ Við reiknum hvert gildi á $f(i, j)$ í $\mathcal{O}(1)$.
- ▶ Við þurfum í versta falli að reinka $n \cdot (c + 1)$ slíka gildi.
- ▶ Svo það tekur okkur $\mathcal{O}(n \cdot c)$ tíma að reikna $f(i, j)$.
- ▶ En sökum minnunar tekura það $\mathcal{O}(q + n \cdot c)$ tíma að reikna $f(i, j)$ q sinnum.
- ▶ Við þurfum að reikna það, í versta falli, c sinnum, svo forritið er $\mathcal{O}(\quad)$.

- ▶ Við reiknum hvert gildi á $f(i, j)$ í $\mathcal{O}(1)$.
- ▶ Við þurfum í versta falli að reinka $n \cdot (c + 1)$ slíka gildi.
- ▶ Svo það tekur okkur $\mathcal{O}(n \cdot c)$ tíma að reikna $f(i, j)$.
- ▶ En sökum minnunar tekura það $\mathcal{O}(q + n \cdot c)$ tíma að reikna $f(i, j)$ q sinnum.
- ▶ Við þurfum að reikna það, í versta falli, c sinnum, svo forritið er $\mathcal{O}(n \cdot c)$.

- ▶ Ein algeng hagnýting á þessu er tvískipting talna.

- ▶ Ein algeng hagnýting á þessu er tvískipting talna.
- ▶ Látum a_1, \dots, a_n vera heiltölur.

- ▶ Ein algeng hagnýting á þessu er tvískipting talna.
- ▶ Látum a_1, \dots, a_n vera heiltölur.
- ▶ Hvernig er best að skipta þeim í tvo hópa þannig að mismunur summa hvors hóps sé sem minnstur.

- ▶ Ein algeng hagnýting á þessu er tvískipting talna.
- ▶ Látum a_1, \dots, a_n vera heiltölur.
- ▶ Hvernig er best að skipta þeim í tvo hópa þannig að mismunur summa hvors hóps sé sem minnstur.
- ▶ Ef tölurnar eru $(10, 2, 10, 30, 15, 2, 30, 10)$ þá skiptum við í $(2, 2, 10, 10, 30)$ og $(10, 15, 30)$.

- ▶ Ein algeng hagnýting á þessu er tvískipting talna.
- ▶ Látum a_1, \dots, a_n vera heiltölur.
- ▶ Hvernig er best að skipta þeim í tvo hópa þannig að mismunur summa hvors hóps sé sem minnstur.
- ▶ Ef tölurnar eru $(10, 2, 10, 30, 15, 2, 30, 10)$ þá skiptum við í $(2, 2, 10, 10, 30)$ og $(10, 15, 30)$.
- ▶ Summurnar eru 54 og 55, og mismunur þeirra er 1.

- ▶ Hvernig getum við leyst þetta?

- ▶ Hvernig getum við leyst þetta?
- ▶ Látum T vera summu allra talnanna.

- ▶ Hvernig getum við leyst þetta?
- ▶ Látum T vera summu allra talnanna.
- ▶ Við getum nú notað `subsetsum(...)` með $c = \lfloor T/2 \rfloor$.

- ▶ Hvernig getum við leyst þetta?
- ▶ Látum T vera summu allra talnanna.
- ▶ Við getum nú notað `subsetsum(...)` með $c = \lfloor T/2 \rfloor$.
- ▶ Það gefur okkur annan hópinn og hinn hópurinn verður afgangurinn.

- ▶ Hvernig getum við leyst þetta?
- ▶ Látum T vera summu allra talnanna.
- ▶ Við getum nú notað `subsetsum(...)` með $c = \lfloor T/2 \rfloor$.
- ▶ Það gefur okkur annan hópinn og hinn hópurinn verður afgangurinn.
- ▶ Okkur vantar þó að finna hvaða tölur eiga að vera í hvorum hóp.

```

7 int d[MAXN][MAXC], b[MAXN];
8 int dp_lookup(int x, int y)
9 {
10     if (x < 0 && y == 0) return 1;
11     if (y < 0 || x < 0) return 0;
12     if (d[x][y] != -1) return d[x][y];
13     return d[x][y] = dp_lookup(x - 1, y) || dp_lookup(x - 1, y - b[x]);
14 }
15
16 void partition(int *a, int n)
17 {
18     int i, j, t = 0, c, g[n];
19     for (i = 0; i < n; i++) t += a[i], g[i] = 0, b[i] = a[i];
20     c = t/2;
21     for (i = 0; i < n; i++) for (j = 0; j < c + 1; j++) d[i][j] = -1;
22     printf("c = %d\n", c);
23     while (!dp_lookup(n - 1, c)) c--;
24     i = n - 1, j = c;
25     printf("c = %d\n", c);
26     while (i > 0 && j > 0)
27     {
28         printf(">> %d %d\n", i, j);
29         if (dp_lookup(i - 1, j) > dp_lookup(i - 1, j - a[i])) i--;
30         else g[i] = 1, j -= a[i], i--;
31     }
32     printf("Fyrri hópur: ");
33     for (i = 0; i < n; i++) if (g[i]) printf(" %2d", a[i]);
34     printf("\n");
35     printf("Seinni hópur: ");
36     for (i = 0; i < n; i++) if (!g[i]) printf(" %2d", a[i]);
37     printf("\n");
38 }

```

- ▶ Látum T vera summu allra talnanna.

- ▶ Látum T vera summu allra talnanna.
- ▶ Þá er tímalækjan $\mathcal{O}(\quad)$.

- ▶ Látum T vera summu allra talnanna.
- ▶ Þá er tímalfækjan $\mathcal{O}(n \cdot T)$.

- ▶ Látum T vera summu allra talnanna.
- ▶ Þá er tímalfækjan $\mathcal{O}(n \cdot T)$.
- ▶ Það er til skemmtileg leið til að fá tímaflækuna $\mathcal{O}(n \cdot w)$, þar sem w er stærsta talan í inntakinu.

- ▶ Látum T vera summu allra talnanna.
- ▶ Þá er tímalfækjan $\mathcal{O}(n \cdot T)$.
- ▶ Það er til skemmtileg leið til að fá tímaflækuna $\mathcal{O}(n \cdot w)$, þar sem w er stærsta talan í inntakinu.
- ▶ Við skoðum hana kannski í næstu viku.

- ▶ Hlutmengjadæmið er í raun sértilfelli af *bakpokadæminu* (e. *Knapsack Problem*).

- ▶ Hlutmengjadæmið er í raun sértilfelli af *bakpokadæminu* (e. *Knapsack Problem*).
- ▶ ...

- ▶ Stundum getur verið erfitt að ákvarða hvernig stöðurúmið okkar á að líta út.

- ▶ Stundum getur verið erfitt að ákvarða hvernig stöðurúmið okkar á að líta út.
- ▶ Tökum vel þekkt dæmi.

- ▶ Stundum getur verið erfitt að ákvarða hvernig stöðurúmið okkar á að líta út.
- ▶ Tökum vel þekkt dæmi.
- ▶ Gerum ráð fyrir að við séum með n stöður.

- ▶ Stundum getur verið erfitt að ákvarða hvernig stöðurúmið okkar á að líta út.
- ▶ Tökum vel þekkt dæmi.
- ▶ Gerum ráð fyrir að við séum með n stöður.
- ▶ Gefið er tvívítt fylki $(d_{ij})_{1 \leq i, j \leq n}$, þar sem d_{ij} táknar tímann sem það tekur að fara úr i -tu stöðunni í j -tu stöðuna.

- ▶ Stundum getur verið erfitt að ákvarða hvernig stöðurúmið okkar á að líta út.
- ▶ Tökum vel þekkt dæmi.
- ▶ Gerum ráð fyrir að við séum með n stöður.
- ▶ Gefið er tvívítt fylki $(d_{ij})_{1 \leq i, j \leq n}$, þar sem d_{ij} táknar tímann sem það tekur að fara úr i -tu stöðunni í j -tu stöðuna.
- ▶ Við viljum nú ferðast í gegnum allar stöðurnar í einhverri röð þannig að við byrjum og endum í sömu stöðu, förum í hverja stöðu nákvæmlega einu sinni (tvisvar í upphafsstöðuna) og tökum sem stystan tíma.

- ▶ Stundum getur verið erfitt að ákvarða hvernig stöðurúmið okkar á að líta út.
- ▶ Tökum vel þekkt dæmi.
- ▶ Gerum ráð fyrir að við séum með n stöður.
- ▶ Gefið er tvívítt fylki $(d_{ij})_{1 \leq i, j \leq n}$, þar sem d_{ij} táknar tímann sem það tekur að fara úr i -tu stöðunni í j -tu stöðuna.
- ▶ Við viljum nú ferðast í gegnum allar stöðurnar í einhverri röð þannig að við byrjum og endum í sömu stöðu, förum í hverja stöðu nákvæmlega einu sinni (tvisvar í upphafsstöðuna) og tökum sem stystan tíma.
- ▶ Þetta er fræga *Farandsölumannadæmið* (e. *Travelling Salesman Problem*).

- ▶ Stundum getur verið erfitt að ákvarða hvernig stöðurúmið okkar á að líta út.
- ▶ Tökum vel þekkt dæmi.
- ▶ Gerum ráð fyrir að við séum með n stöður.
- ▶ Gefið er tvívítt fylki $(d_{ij})_{1 \leq i, j \leq n}$, þar sem d_{ij} táknar tímann sem það tekur að fara úr i -tu stöðunni í j -tu stöðuna.
- ▶ Við viljum nú ferðast í gegnum allar stöðurnar í einhverri röð þannig að við byrjum og endum í sömu stöðu, förum í hverja stöðu nákvæmlega einu sinni (tvisvar í upphafsstöðuna) og tökum sem stystan tíma.
- ▶ Þetta er fræga *Farandsölumannadæmið* (e. *Travelling Salesman Problem*).
- ▶ Sígilt er að leysa þetta dæmi endurkvæmt í $\mathcal{O}((n+1)!)$ tíma.

- ▶ Stundum getur verið erfitt að ákvarða hvernig stöðurúmið okkar á að líta út.
- ▶ Tökum vel þekkt dæmi.
- ▶ Gerum ráð fyrir að við séum með n stöður.
- ▶ Gefið er tvívítt fylki $(d_{ij})_{1 \leq i, j \leq n}$, þar sem d_{ij} táknar tímann sem það tekur að fara úr i -tu stöðunni í j -tu stöðuna.
- ▶ Við viljum nú ferðast í gegnum allar stöðurnar í einhverri röð þannig að við byrjum og endum í sömu stöðu, förum í hverja stöðu nákvæmlega einu sinni (tvisvar í upphafsstöðuna) og tökum sem stystan tíma.
- ▶ Þetta er fræga *Farandsölumannadæmið* (e. *Travelling Salesman Problem*).
- ▶ Sígilt er að leysa þetta dæmi endurkvæmt í $\mathcal{O}((n+1)!)$ tíma.
- ▶ Við höfum nú tólin til að gera betur.

- ▶ Tökum fyrst eftir að það skiptir ekki máli í hvaða stöðu við byrjum.

- ▶ Tökum fyrst eftir að það skiptir ekki máli í hvaða stöðu við byrjum.
- ▶ Við getum því gert ráð fyrir að við byrjum í fyrstu stöðunni.

- ▶ Tökum fyrst eftir að það skiptir ekki máli í hvaða stöðu við byrjum.
- ▶ Við getum því gert ráð fyrir að við byrjum í fyrstu stöðunni.
- ▶ Látum P tákna mengi alla staða, A vera eiginlegt hlutmengi þar í og s vera stak utan A .

- ▶ Tökum fyrst eftir að það skiptir ekki máli í hvaða stöðu við byrjum.
- ▶ Við getum því gert ráð fyrir að við byrjum í fyrstu stöðunni.
- ▶ Látum P tákna mengi alla staða, A vera eiginlegt hlutmengi þar í og s vera stak utan A .
- ▶ Við getum þá látið $f(s, A)$ vera stysta leiðin til að fara í allar stöður A nákvæmlega einu sinni frá s og enda í fyrstu stöðunni.

- ▶ Tökum fyrst eftir að það skiptir ekki máli í hvaða stöðu við byrjum.
- ▶ Við getum því gert ráð fyrir að við byrjum í fyrstu stöðunni.
- ▶ Látum P tákna mengi alla staða, A vera eiginlegt hlutmengi þar í og s vera stak utan A .
- ▶ Við getum þá látið $f(s, A)$ vera stysta leiðin til að fara í allar stöður A nákvæmlega einu sinni frá s og enda í fyrstu stöðunni.
- ▶ Tökum eftir að $f(s, \emptyset) = d_{s1}$.

- ▶ Tökum fyrst eftir að það skiptir ekki máli í hvaða stöðu við byrjum.
- ▶ Við getum því gert ráð fyrir að við byrjum í fyrstu stöðunni.
- ▶ Látum P tákna mengi alla staða, A vera eiginlegt hlutmengi þar í og s vera stak utan A .
- ▶ Við getum þá látið $f(s, A)$ vera stysta leiðin til að fara í allar stöður A nákvæmlega einu sinni frá s og enda í fyrstu stöðunni.
- ▶ Tökum eftir að $f(s, \emptyset) = d_{s1}$.
- ▶ Við getum nú sett fallið f fram endurkvæmt með

$$f(s, A) = \begin{cases} d_{s1}, & \text{ef } A = \emptyset \\ \min_{e \in A} (d_{se} + f(e, A \setminus e)), & \text{annars.} \end{cases}$$

- ▶ Tökum fyrst eftir að það skiptir ekki máli í hvaða stöðu við byrjum.
- ▶ Við getum því gert ráð fyrir að við byrjum í fyrstu stöðunni.
- ▶ Látum P tákna mengi alla staða, A vera eiginlegt hlutmengi þar í og s vera stak utan A .
- ▶ Við getum þá látið $f(s, A)$ vera stysta leiðin til að fara í allar stöður A nákvæmlega einu sinni frá s og enda í fyrstu stöðunni.
- ▶ Tökum eftir að $f(s, \emptyset) = d_{s1}$.
- ▶ Við getum nú sett fallið f fram endurkvæmt með

$$f(s, A) = \begin{cases} d_{s1}, & \text{ef } A = \emptyset \\ \min_{e \in A} (d_{se} + f(e, A \setminus e)), & \text{annars.} \end{cases}$$

- ▶ Svárið við dæminu fæst svo með $f(\quad)$.

- ▶ Tökum fyrst eftir að það skiptir ekki máli í hvaða stöðu við byrjum.
- ▶ Við getum því gert ráð fyrir að við byrjum í fyrstu stöðunni.
- ▶ Látum P tákna mengi alla staða, A vera eiginlegt hlutmengi þar í og s vera stak utan A .
- ▶ Við getum þá látið $f(s, A)$ vera stysta leiðin til að fara í allar stöður A nákvæmlega einu sinni frá s og enda í fyrstu stöðunni.
- ▶ Tökum eftir að $f(s, \emptyset) = d_{s1}$.
- ▶ Við getum nú sett fallið f fram endurkvæmt með

$$f(s, A) = \begin{cases} d_{s1}, & \text{ef } A = \emptyset \\ \min_{e \in A} (d_{se} + f(e, A \setminus e)), & \text{annars.} \end{cases}$$

- ▶ Svárið við dæminu fæst svo með $f(1, P \setminus \{1\})$.

- ▶ Wikipedia kennir þessa aðferð við Held og Karp (1962) og segir að þetta sé með fyrstu hagnýtingum kvikrar bestunar.

- ▶ Wikipedia kennir þessa aðferð við Held og Karp (1962) og segir að þetta sé með fyrstu hagnýtingum kvikrar bestunar.
- ▶ Í versta falli þurfum við að reikna falligildi á f , ef við erum með n stöður.

- ▶ Wikipedia kennir þessa aðferð við Held og Karp (1962) og segir að þetta sé með fyrstu hagnýtingum kvikrar bestunar.
- ▶ Í versta falli þurfum við að reikna $n \cdot 2^{n-1}$ falligildi á f , ef við erum með n stöður.

- ▶ Wikipedia kennir þessa aðferð við Held og Karp (1962) og segir að þetta sé með fyrstu hagnýtingum kvikrar bestunar.
- ▶ Í versta falli þurfum við að reikna $n \cdot 2^{n-1}$ fallgildi á f , ef við erum með n stöður.
- ▶ Hvert fallgildi er reiknað í $\mathcal{O}(n)$ tíma.

- ▶ Wikipedia kennir þessa aðferð við Held og Karp (1962) og segir að þetta sé með fyrstu hagnýtingum kvikrar bestunar.
- ▶ Í versta falli þurfum við að reikna $n \cdot 2^{n-1}$ fallgildi á f , ef við erum með n stöður.
- ▶ Hvert fallgildi er reiknað í $\mathcal{O}(n)$ tíma.

- ▶ Wikipedia kennir þessa aðferð við Held og Karp (1962) og segir að þetta sé með fyrstu hagnýtingum kvikrar bestunar.
- ▶ Í versta falli þurfum við að reikna $n \cdot 2^{n-1}$ fallgildi á f , ef við erum með n stöður.
- ▶ Hvert fallgildi er reiknað í $\mathcal{O}(n)$ tíma.
- ▶ Svo í heildina er forritið $\mathcal{O}(\quad)$.

- ▶ Wikipedia kennir þessa aðferð við Held og Karp (1962) og segir að þetta sé með fyrstu hagnýtingum kvikrar bestunar.
- ▶ Í versta falli þurfum við að reikna $n \cdot 2^{n-1}$ fallgildi á f , ef við erum með n stöður.
- ▶ Hvert fallgildi er reiknað í $\mathcal{O}(n)$ tíma.
- ▶ Svo í heildina er forritið $\mathcal{O}(n^2 \cdot 2^n)$.

- ▶ Wikipedia kennir þessa aðferð við Held og Karp (1962) og segir að þetta sé með fyrstu hagnýtingum kvikrar bestunar.
- ▶ Í versta falli þurfum við að reikna $n \cdot 2^{n-1}$ fallgildi á f , ef við erum með n stöður.
- ▶ Hvert fallgildi er reiknað í $\mathcal{O}(n)$ tíma.
- ▶ Svo í heildina er forritið $\mathcal{O}(n^2 \cdot 2^n)$.
- ▶ Samkvæmt 10^8 reglunni náum við að leysa dæmi með $n \leq$.

- ▶ Wikipedia kennir þessa aðferð við Held og Karp (1962) og segir að þetta sé með fyrstu hagnýtingum kvikrar bestunar.
- ▶ Í versta falli þurfum við að reikna $n \cdot 2^{n-1}$ fallgildi á f , ef við erum með n stöður.
- ▶ Hvert fallgildi er reiknað í $\mathcal{O}(n)$ tíma.
- ▶ Svo í heildina er forritið $\mathcal{O}(n^2 \cdot 2^n)$.
- ▶ Samkvæmt 10^8 reglunni náum við að leysa dæmi með $n \leq 18$.

- ▶ Wikipedia kennir þessa aðferð við Held og Karp (1962) og segir að þetta sé með fyrstu hagnýtingum kvikrar bestunar.
- ▶ Í versta falli þurfum við að reikna $n \cdot 2^{n-1}$ fallgildi á f , ef við erum með n stöður.
- ▶ Hvert fallgildi er reiknað í $\mathcal{O}(n)$ tíma.
- ▶ Svo í heildina er forritið $\mathcal{O}(n^2 \cdot 2^n)$.
- ▶ Samkvæmt 10^8 reglunni náum við að leysa dæmi með $n \leq 18$.
- ▶ Við náum bara $n \leq$ með augljósum endurkvæmu lausninni.

- ▶ Wikipedia kennir þessa aðferð við Held og Karp (1962) og segir að þetta sé með fyrstu hagnýtingum kvikrar bestunar.
- ▶ Í versta falli þurfum við að reikna $n \cdot 2^{n-1}$ fallgildi á f , ef við erum með n stöður.
- ▶ Hvert fallgildi er reiknað í $\mathcal{O}(n)$ tíma.
- ▶ Svo í heildina er forritið $\mathcal{O}(n^2 \cdot 2^n)$.
- ▶ Samkvæmt 10^8 reglunni náum við að leysa dæmi með $n \leq 18$.
- ▶ Við náum bara $n \leq 10$ með augljósum endurkvæmu lausninni.

```

1 #include <stdio.h>
2 #include <assert.h>
3 #define MAXN 18
4 #define INF (1 << 30)
5 int min(int a, int b) { if (a < b) return a; return b; }
6
7 int a[MAXN][MAXN], d[MAXN][1 << MAXN], n;
8 int dp_lookup(int x, int y)
9 {
10     int i;
11     if (d[x][y] != -1) return d[x][y];
12     if (y == 0) return a[x][0];
13     d[x][y] = INF;
14     for (i = 0; i < n; i++) if ((y & (1 << i)) != 0)
15         d[x][y] = min(d[x][y], dp_lookup(i, y - (1 << i)) + a[x][i]);
16     return d[x][y];
17 }
18
19 int main()
20 {
21     int i, j;
22     scanf("%d", &n);
23     for (i = 0; i < n; i++) for (j = 0; j < (1 << n); j++) d[i][j] = -1;
24     for (i = 0; i < n; i++) for (j = 0; j < n; j++) scanf("%d", &a[i][j]);
25     printf("%d\n", dp_lookup(0, (1 << n) - 2));
26     return 0;
27 }

```

