Deila & drottna og kvik bestun

Fleiri lausnaraðferðir

Atli Fannar Franklín

28. janúar 2019

Efnisyfirlit

Pairs

- Í C++ býr pair<A, B> til par af einhverju af tagi A og B. Ná má í fyrra stakið með .first og seinna með .second. Raðað er fyrst m.t.t. fyrra staks og seinna staks svo ef fyrri eru jöfn.

Pairs

- Í C++ býr pair<A, B> til par af einhverju af tagi A og B. Ná má í fyrra stakið með .first og seinna með .second. Raðað er fyrst m.t.t. fyrra staks og seinna staks svo ef fyrri eru jöfn.
- Oft notum við pair<int,int> eða pair<long long, long long>

Custom sort

- Hægt er að sorta hlut með custom sort með eftirfarandi hætti (hér erum við að bera saman pairs af ints eins og þau væru brot)

Custom sort

- Hægt er að sorta hlut með custom sort með eftirfarandi hætti (hér erum við að bera saman pairs af ints eins og þau væru brot)
- Einnig er hægt að láta sets, maps, priority queues o.s.frv. nota custom sort

Custom sort kóði

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int,int> ii;

struct iicomp {
    bool inline operator()(const ii& a, const ii& b) {
        return a.first * b.second < b.first * a.second;
    }
};

int main() {
        vector<ii> a(20, ii(0, 0));
        sort(a.begin(), a.end(), iicomp());
        set<ii, iicomp> s;
        priority_queue<ii, vector<ii>, iicomp> pq;
}
```

Efnisyfirlit

Hvað er deila & drottna?

- Lausnaraðferð sem felur í sér að skipta vandamálinu niður í smærri hliðstæð vandamál

Hvað er deila & drottna?

- Lausnaraðferð sem felur í sér að skipta vandamálinu niður í smærri hliðstæð vandamál
- Útfærsla felur þá í sér endurkvæmt fall sem skoðar smærri og smærri hlutvandamál þar til að vandamálið sé nógu lítið til að auðvelt sé að finna lausnina

Hvað er deila & drottna?

- Lausnaraðferð sem felur í sér að skipta vandamálinu niður í smærri hliðstæð vandamál
- Útfærsla felur þá í sér endurkvæmt fall sem skoðar smærri og smærri hlutvandamál þar til að vandamálið sé nógu lítið til að auðvelt sé að finna lausnina
- En hvað má leysa með þessum hætti?

Dæmi um deila og drottna reiknirit

- Binary search (lang **LANG** mikilvægast)

Dæmi um deila og drottna reiknirit

- Binary search (lang **LANG** mikilvægast)
- Röðunarreiknirit, t.d. quicksort og mergesort

Dæmi um deila og drottna reiknirit

- Binary search (lang **LANG** mikilvægast)
- Röðunarreiknirit, t.d. quicksort og mergesort
- Closest point pair í $\mathcal{O}(n \log(n))$

Dæmi um deila og drottna reiknirit

- Binary search (lang **LANG** mikilvægast)
- Röðunarreiknirit, t.d. quicksort og mergesort
- Closest point pair í $\mathcal{O}(n \log(n))$
- Strassen algorithm

Dæmi um deila og drottna reiknirit

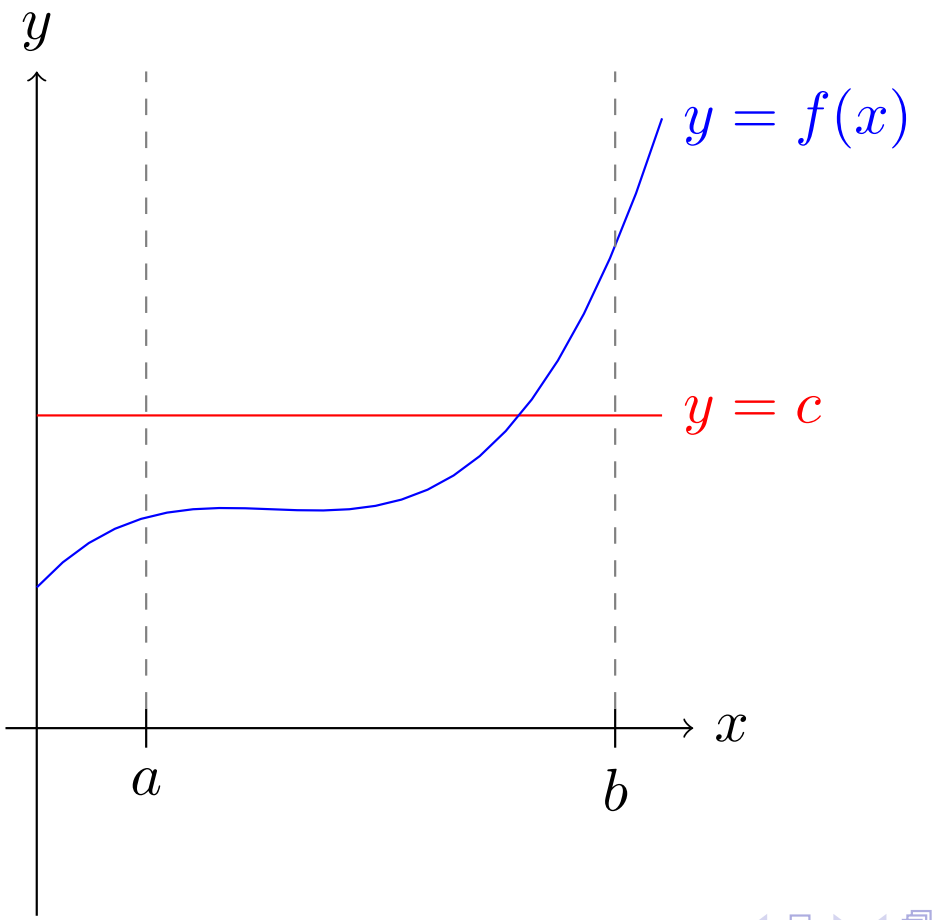
- Binary search (lang **LANG** mikilvægast)
- Röðunarreiknirit, t.d. quicksort og mergesort
- Closest point pair í $\mathcal{O}(n \log(n))$
- Strassen algorithm
- Cooley-Tukey Fast Fourier Transform

Hvað er binary search?

- Segjum að við höfum samfellt fall f og tölu c þ.a. til séu einhver $a < b$ þannig að $f(a)$ og $f(b)$ séu sitt hvoru megin við c

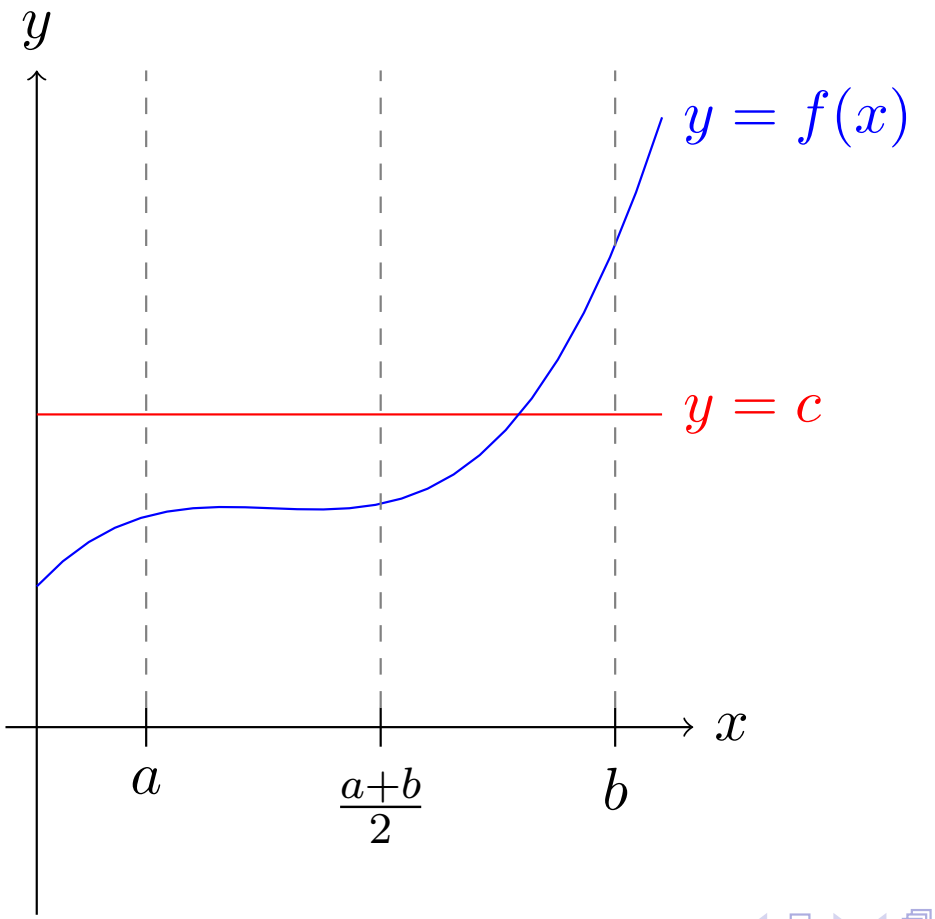
Hvað er binary search?

- Segjum að við höfum samfellt fall f og tölu c þ.a. til séu einhver $a < b$ þannig að $f(a)$ og $f(b)$ séu sitt hvoru megin við c



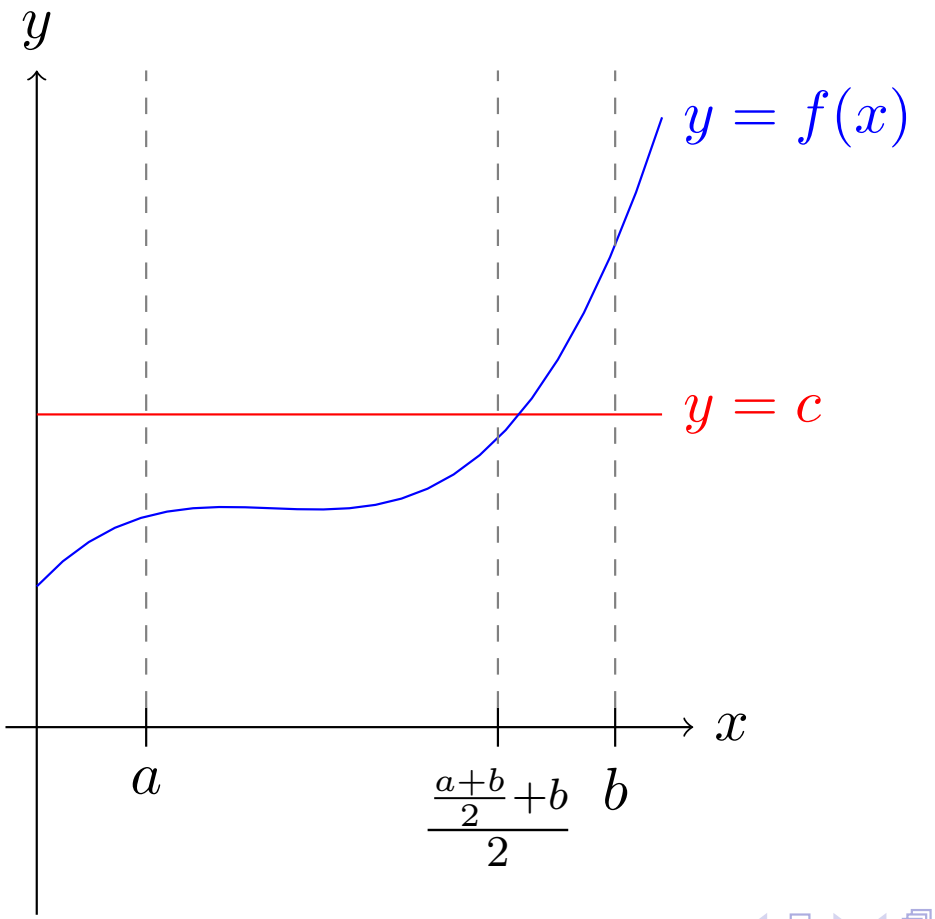
Hvað er binary search?

- Segjum að við höfum samfellt fall f og tölu c þ.a. til séu einhver $a < b$ þannig að $f(a)$ og $f(b)$ séu sitt hvoru megin við c
- Þá getum við kíkt hvort fallið sé fyrir ofan eða neðan í miðjunni og haldið áfram að leita í réttum helmingi



Hvað er binary search?

- Segjum að við höfum samfellt fall f og tölu c þ.a. til séu einhver $a < b$ þannig að $f(a)$ og $f(b)$ séu sitt hvoru megin við c
- Þá getum við kíkt hvort fallið sé fyrir ofan eða neðan í miðjunni og haldið áfram að leita í réttum helmingi



Af hverju er þetta svona nytsamlegt?

- Þetta er mjög hratt! Ef billengdin okkar hefur lengd L og fallið tekur $T(n)$ tíma að reikna finnum við svarið (eða nógu góða nálgun) í $\mathcal{O}(T(n)\log(L))$ tíma

Af hverju er þetta svona nytsamlegt?

- Þetta er mjög hratt! Ef billengdin okkar hefur lengd L og fallið tekur $T(n)$ tíma að reikna finnum við svarið (eða nógu góða nálgun) í $\mathcal{O}(T(n)\log(L))$ tíma
- Þetta á ekki bara við um föll. Ef við erum bara með *eitthvað* sem vex samfellt (t.d. er einhalla) og við viljum finna núllstöð á, þá getum við einfaldað málið

Af hverju er þetta svona nytsamlegt?

- Þetta er mjög hratt! Ef billengdin okkar hefur lengd L og fallið tekur $T(n)$ tíma að reikna finnum við svarið (eða nógu góða nálgun) í $\mathcal{O}(T(n)\log(L))$ tíma
- Þetta á ekki bara við um föll. Ef við erum bara með *eitthvað* sem vex samfellt (t.d. er einhalla) og við viljum finna núllstöð á, þá getum við einfaldað málið
- Almennt er málið að við viljum finna minnstu/stærstu töluna sem eitthvað gildir um. Ef það að eitthvað gildi um töluna gefi að það gildi um allar tölur fyrir ofan/neðan hana er hægt að helmingunarleita að tölunni

Af hverju er þetta svona nytsamlegt?

- Þetta er mjög hratt! Ef billengdin okkar hefur lengd L og fallið tekur $T(n)$ tíma að reikna finnum við svarið (eða nógu góða nálgun) í $\mathcal{O}(T(n)\log(L))$ tíma
- Þetta á ekki bara við um föll. Ef við erum bara með *eitthvað* sem vex samfellt (t.d. er einhalla) og við viljum finna núllstöð á, þá getum við einfaldað málið
- Almennt er málið að við viljum finna minnstu/stærstu töluna sem eitthvað gildir um. Ef það að eitthvað gildi um töluna gefi að það gildi um allar tölur fyrir ofan/neðan hana er hægt að helmingunarleita að tölunni
- Hvers vegna hjálpar þetta? Þá er búið að einfalda vandamálið úr 'Hver er minnsta slíka talan?' í 'Er þetta hægt fyrir töluna x?' sem er oft *miklu* auðveldara að leysa

Tökum dæmi

- Við höfum einhverja punkta $0 \leq x_1 \leq \cdots \leq x_n \leq 10^9$ sem eru staðsetningar á túristaíbúðum meðfram beinni strandlengju. Við höfum svo að $k \leq n$ túristar vilji leigja íbúð. Túristarnir vilja verða sem minnst varir við aðra túrista, svo við viljum hafa bilið milli túristanna sem búa næst hvoröðrum sem lengst. Hver er þessi hámarks stysta lengd?

Tökum dæmi

- Við höfum einhverja punkta $0 \leq x_1 \leq \cdots \leq x_n \leq 10^9$ sem eru staðsetningar á túristaíbúðum meðfram beinni strandlengju. Við höfum svo að $k \leq n$ túristar vilji leigja íbúð. Túristarnir vilja verða sem minnst varir við aðra túrista, svo við viljum hafa bilið milli túristanna sem búa næst hvoröðrum sem lengst. Hver er þessi hámarks stysta lengd?
- Ef þetta er hægt með stystu lengd r er þetta líka hægt fyrir styttri stystu lengd. Því helmingunarleitum við yfir stystu lengdina. Þá er spunringin orðin, er hægt að raða túristunum niður þannig að það sé a.m.k. r langt milli þeirra allra?

Tökum dæmi

- Við höfum einhverja punkta $0 \leq x_1 \leq \cdots \leq x_n \leq 10^9$ sem eru staðsetningar á túristaíbúðum meðfram beinni strandlengju. Við höfum svo að $k \leq n$ túristar vilji leigja íbúð. Túristarnir vilja verða sem minnst varir við aðra túrista, svo við viljum hafa bilið milli túristanna sem búa næst hvoröðrum sem lengst. Hver er þessi hámarks stysta lengd?
- Ef þetta er hægt með stystu lengd r er þetta líka hægt fyrir styttri stystu lengd. Því helmingunarleitum við yfir stystu lengdina. Þá er spunringin orðin, er hægt að raða túristunum niður þannig að það sé a.m.k. r langt milli þeirra allra?
- Nú getum við gert restina gráðugt. Það er alltaf betra að setja túrista í hús á endunum og eins nálægt endunum og hægt er. Því setjum við bara túrista í húsið 'lengst til vinstri', svo næsta hús sem er a.m.k. r í burtu o.s.frv. Þá ef við komum túristunum fyrir er svarið já fyrir r, annars nei.

Lausnin

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    long long n, k;
    cin >> n >> k;
    vector<long long> v(n);
    for(int i = 0; i < n; ++i) cin >> v[i];
    long long ans = 1, minans = 1, maxans = 10000000000;
    while(maxans - minans > 0) {
        long long pos = v[0], cnt = 1, midans = (maxans + minans) / 2;
        for(int i = 1; i < n; ++i) {
            if(v[i] - pos >= midans) cnt++, pos = v[i];
        }
        if(cnt < k) maxans = midans - 1;
        else ans = midans, minans = midans + 1;
    }
    cout << ans << endl;
    return 0;
}
```

Önnur deila og drottna reiknirit

- Helmingunarleit er einfalt í útfærslu miðað við önnur deila og drottna reiknirit (þó svo að það að sjá út hvenær á að nota það getur verið erfitt)

Önnur deila og drottna reiknirit

- Helmingunarleit er einfalt í útfærslu miðað við önnur deila og drottna reiknirit (þó svo að það að sjá út hvenær á að nota það getur verið erfitt)
- Stundum þarf að skipta dæminu upp og skoða alla partana sem skipt er upp í og svo jafnvel að sameina lausnir í lokin

Önnur deila og drottna reiknirit

- Helmingunarleit er einfalt í útfærslu miðað við önnur deila og drottna reiknirit (þó svo að það að sjá út hvenær á að nota það getur verið erfitt)
- Stundum þarf að skipta dæminu upp og skoða alla partana sem skipt er upp í og svo jafnvel að sameina lausnir í lokin
- Þetta eru ekki rosa algeng dæmi, en gott er að þekkja til þeirra ef maður þarf að leysa dæmi með þessum hætti

Önnur deila og drottna reiknirit

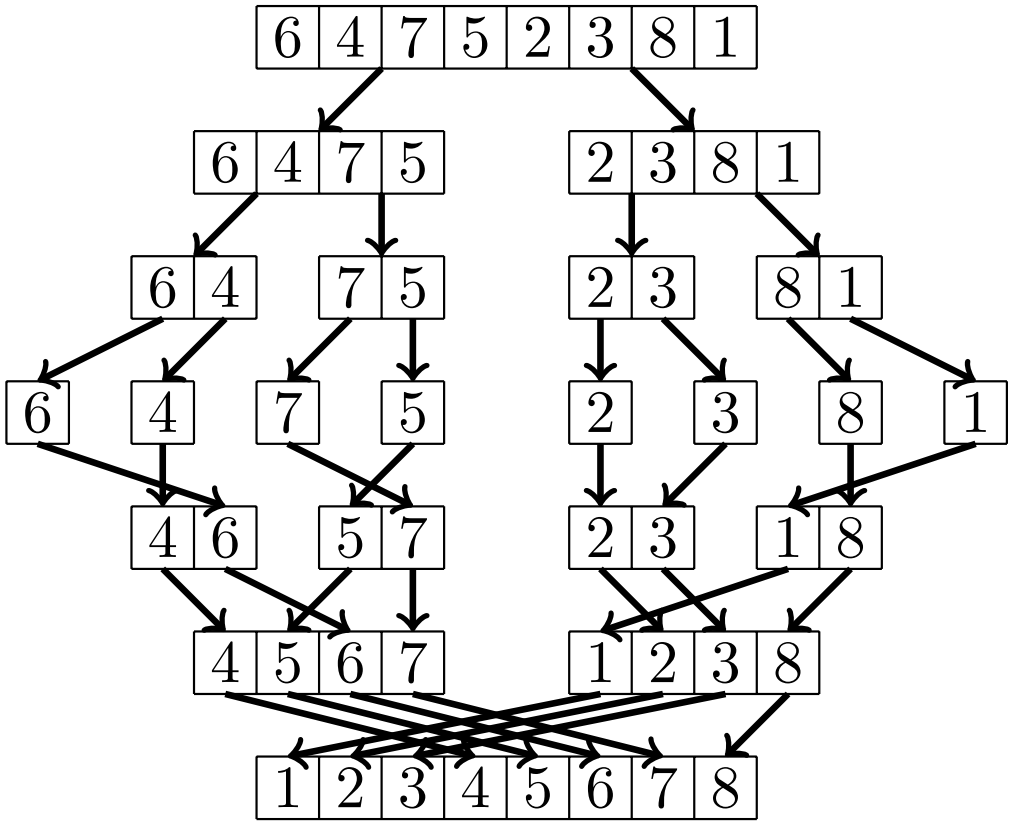
- Helmingunarleit er einfalt í útfærslu miðað við önnur deila og drottna reiknirit (þó svo að það að sjá út hvenær á að nota það getur verið erfitt)
- Stundum þarf að skipta dæminu upp og skoða alla partana sem skipt er upp í og svo jafnvel að sameina lausnir í lokin
- Þetta eru ekki rosa algeng dæmi, en gott er að þekkja til þeirra ef maður þarf að leysa dæmi með þessum hætti
- Þó svo að það sé ekki beint keppnisforritunardæmi lýsir mergesort reikniritið þessu mjög vel, skoðum því mergesort snöggvast

Mergesort

- Mergesort felur það í sér að skipta lista í tvennt, kalla endurkvæmt á sjálft sig í báðum helmingum og sameina niðurstöður með því að sameina báða röðuðu listana í línulegum tíma

Mergesort

- Mergesort felur það í sér að skipta lista í tvennt, kalla endurkvæmt á sjálft sig í báðum helmingum og sameina niðurstöður með því að sameina báða röðuðu listana í línulegum tíma



Útfærsla

```
#include <bits/stdc++.h>
using namespace std;

void merge(int* a, int l, int m, int r) {
    int cnt1 = l, cnt2 = m + 1, cnt3 = 0;
    int acopy[r - l + 1];
    while(cnt1 <= m || cnt2 <= r) {
        if(cnt2 > r) acopy[cnt3] = a[cnt1++];
        else if(cnt1 > m) acopy[cnt3] = a[cnt2++];
        else if(a[cnt1] <= a[cnt2]) acopy[cnt3] = a[cnt1++];
        else acopy[cnt3] = a[cnt2++];
        cnt3++;
    }
    for(int i = l; i <= r; ++i) a[i] = acopy[i - l];
}

void mergesort(int* a, int l, int r) {
        if(r <= l) return;
        int m = l + (r - l) / 2;
        mergesort(a, l, m);
        mergesort(a, m + 1, r);
        merge(a, l, m, r);
}

int main() {
        int a[8] = {6, 4, 7, 5, 2, 3, 8, 1};
        mergesort(a, 0, 7);
        for(int i = 0; i < 8; ++i) cout << a[i] << ' ';
        cout << endl;
}
```

Efnisyfirlit

Hvað er kvik bestun?

- Það er í raun almennari útgáfa af deila og drottna

Hvað er kvik bestun?

- Það er í raun almennari útgáfa af deila og drottna
- Við erum áfram að skipta vandamálinu niður í smærri vandamál en við geymum lausnirnar á smærri vandamálunum

Hvað er kvik bestun?

- Það er í raun almennari útgáfa af deila og drottna
- Við erum áfram að skipta vandamálinu niður í smærri vandamál en við geymum lausnirnar á smærri vandamálunum
- Þetta þýðir að ef sama hlutvandamálið kemur upp oftar en einu sinni, þá erum við geymda lausn á því og þurfum ekki að reikna það aftur

Hvað er kvik bestun?

- Það er í raun almennari útgáfa af deila og drottna
- Við erum áfram að skipta vandamálinu niður í smærri vandamál en við geymum lausnirnar á smærri vandamálunum
- Þetta þýðir að ef sama hlutvandamálið kemur upp oftar en einu sinni, þá erum við geymda lausn á því og þurfum ekki að reikna það aftur
- Tökum (contrived) dæmi um þetta

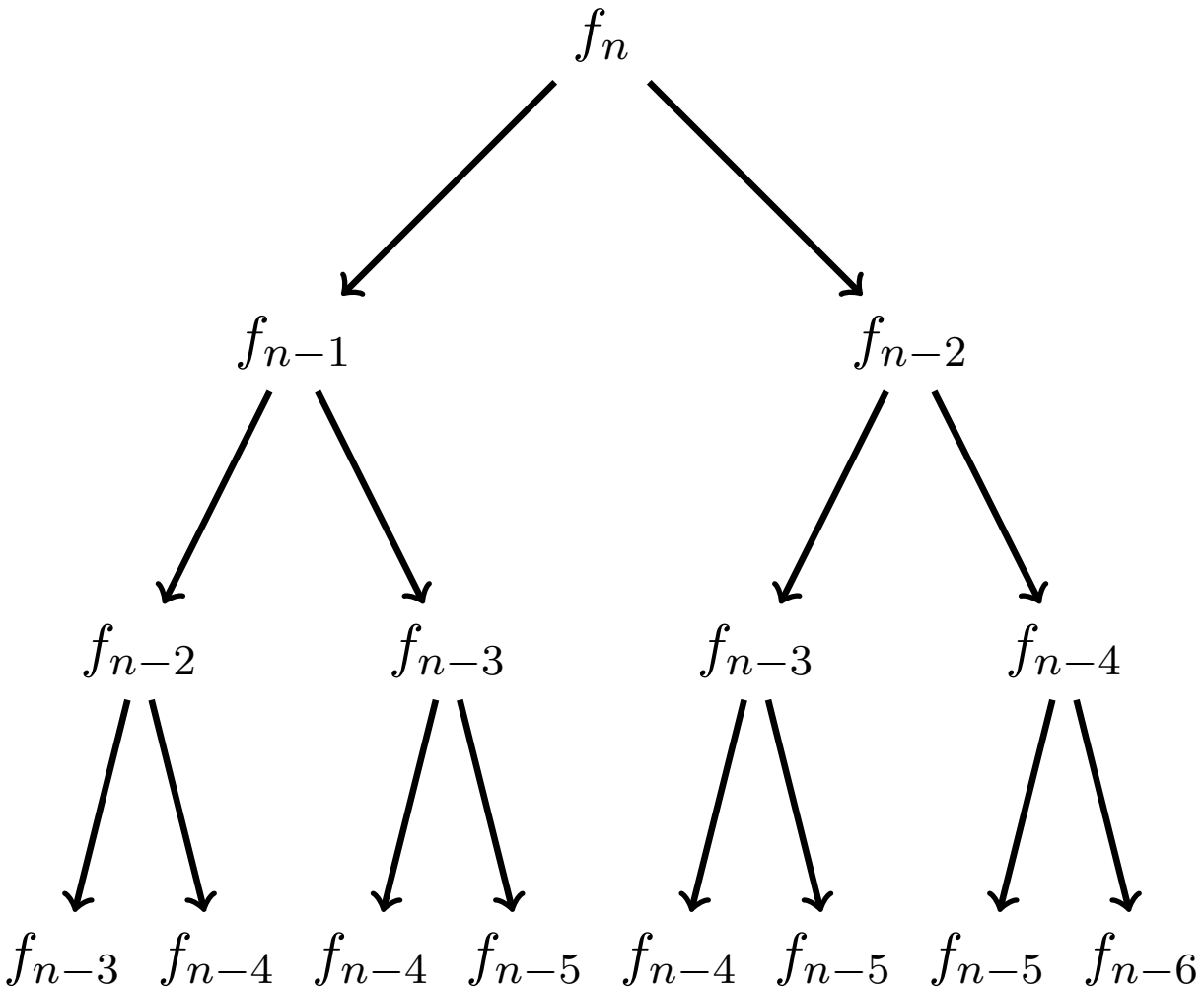
- Við viljum reikna n-tu Fibonnacci töluna, f_n. Hvernig má reikna hana?

- Við viljum reikna n-tu Fibonnacci töluna, f_n. Hvernig má reikna hana?
- Nú við vitum $f_n = f_{n-1} + f_{n-2}$. Sjáum hvað gerist ef við reiknum þetta með deila og drottna.

- Við viljum reikna n-tu Fibonnacci töluna, f_n. Hvernig má reikna hana?
- Nú við vitum $f_n = f_{n-1} + f_{n-2}$. Sjáum hvað gerist ef við reiknum þetta með deila og drottna.
- Þá getum við sagt að ef það tekur $T(n)$ tíma að reikna f_n þá gildi jafnan $T(n) = T(n-1) + T(n-2)$

- Við viljum reikna n-tu Fibonnacci töluna, f_n. Hvernig má reikna hana?
- Nú við vitum $f_n = f_{n-1} + f_{n-2}$. Sjáum hvað gerist ef við reiknum þetta með deila og drottna.
- Þá getum við sagt að ef það tekur $T(n)$ tíma að reikna f_n þá gildi jafnan $T(n) = T(n-1) + T(n-2)$
- Leysa má út úr þessu og fá að $T(n)$ sé $\mathcal{O}(2^n)$. Þetta er *rosalega* hægt, við komumst varla upp í f_{30}

- Við viljum reikna n-tu Fibonnacci töluna, f_n. Hvernig má reikna hana?
- Nú við vitum $f_n = f_{n-1} + f_{n-2}$. Sjáum hvað gerist ef við reiknum þetta með deila og drottna.
- Þá getum við sagt að ef það tekur $T(n)$ tíma að reikna f_n þá gildi jafnan $T(n) = T(n-1) + T(n-2)$
- Leysa má út úr þessu og fá að $T(n)$ sé $\mathcal{O}(2^n)$. Þetta er *rosalega* hægt, við komumst varla upp í f_{30}
- Hvert er vandamálið okkar? Hvernig má bæta þetta?


f_n
f_{n-1}
f_{n-2}
f_{n-2}
f_{n-3}
f_{n-3}
f_{n-4}
f_{n-3}
f_{n-4}
f_{n-4}
f_{n-5}
f_{n-4}
f_{n-5}
f_{n-5}
f_{n-6}

Endurtekningar

- Sjáum að við erum að reikna sömu töluna margoft

Endurtekningar

- Sjáum að við erum að reikna sömu töluna margoft
- Því myndi þetta ganga miklu betur ef við geymdum tölurnar hingað til

Endurtekningar

- Sjáum að við erum að reikna sömu töluna margoft
- Því myndi þetta ganga miklu betur ef við geymdum tölurnar hingað til
- Við búum til fylki $f[n]$ sem geymir $f_1, \ldots, f_n$ og þá getum við reiknað hverja tölu f_i í föstum tíma með því að leggja saman hinar tvær á undan í fylkinu

Endurtekningar

- Sjáum að við erum að reikna sömu töluna margoft
- Því myndi þetta ganga miklu betur ef við geymdum tölurnar hingað til
- Við búum til fylki $f[n]$ sem geymir $f_1, \ldots, f_n$ og þá getum við reiknað hverja tölu f_i í föstum tíma með því að leggja saman hinar tvær á undan í fylkinu
- Þá er hægt að reikna þetta í $\mathcal{O}(n)$, sem er **miklu** hraðara!

Endurtekningar

- Sjáum að við erum að reikna sömu töluna margoft
- Því myndi þetta ganga miklu betur ef við geymdum tölurnar hingað til
- Við búum til fylki $f[n]$ sem geymir $f_1, \ldots, f_n$ og þá getum við reiknað hverja tölu f_i í föstum tíma með því að leggja saman hinar tvær á undan í fylkinu
- Þá er hægt að reikna þetta í $\mathcal{O}(n)$, sem er **miklu** hraðara!
- Við komumst upp í f_{10^8} án neinna vandræða, miklu betra en að ná varla f_{30}

Endurtekningar

- Sjáum að við erum að reikna sömu töluna margoft
- Því myndi þetta ganga miklu betur ef við geymdum tölurnar hingað til
- Við búum til fylki $f[n]$ sem geymir $f_1, \ldots, f_n$ og þá getum við reiknað hverja tölu f_i í föstum tíma með því að leggja saman hinar tvær á undan í fylkinu
- Þá er hægt að reikna þetta í $\mathcal{O}(n)$, sem er **miklu** hraðara!
- Við komumst upp í f_{10^8} án neinna vandræða, miklu betra en að ná varla f_{30}
- Reyndar má reikna f_n í $\mathcal{O}(\log(n))$ tíma. Þeir sem eru áhugasamir mega reyna pæla í því hvernig má gera það. Hint: Notið fylkjamargföldun

LCS

- Skoðum klassískt vandamál sem má leysa með kvikri bestun (og er besta slíka lausn)

LCS

- Skoðum klassískt vandamál sem má leysa með kvikri bestun (og er besta slíka lausn)
- Höfum tvær raðir af tölum $x_1, \ldots, x_n$ og $y_1, \ldots, y_m$ og viljum finna lengd lengstu runu af tölum $z_1, \ldots, z_k$ sem er hlutruna í þeim báðum (LCS stendur fyrir Longest Common Subsequence)

LCS

- Skoðum klassískt vandamál sem má leysa með kvikri bestun (og er besta slíka lausn)
- Höfum tvær raðir af tölum $x_1, \ldots, x_n$ og $y_1, \ldots, y_m$ og viljum finna lengd lengstu runu af tölum $z_1, \ldots, z_k$ sem er hlutruna í þeim báðum (LCS stendur fyrir Longest Common Subsequence)
- Að prófa allar hlutrunur í þeim báðum og bera saman er $\mathcal{O}(\max(n, m)2^{\min(n,m)})$ sem er óhugnalega hægt

LCS

- Skoðum klassískt vandamál sem má leysa með kvikri bestun (og er besta slíka lausn)
- Höfum tvær raðir af tölum $x_1, \ldots, x_n$ og $y_1, \ldots, y_m$ og viljum finna lengd lengstu runu af tölum $z_1, \ldots, z_k$ sem er hlutruna í þeim báðum (LCS stendur fyrir Longest Common Subsequence)
- Að prófa allar hlutrunur í þeim báðum og bera saman er $\mathcal{O}(\max(n, m)2^{\min(n,m)})$ sem er óhugnalega hægt
- Notum kvika bestun! En hvernig á taflan okkar að líta út? Og hvernig skiptum við vandamálinu upp?

LCS

- Búum til töflu T af stærð nm og látum $T[i][j]$ vera lengd lengstu sameiginlegu hlutrunu $x_1, \ldots, x_i$ og $y_1, \ldots, y_j$

LCS

- Búum til töflu T af stærð nm og látum $T[i][j]$ vera lengd lengstu sameiginlegu hlutrunu $x_1, \ldots, x_i$ og $y_1, \ldots, y_j$
- Sem grunntilvik er ljóst að $T[i][0] = T[0][j] = 0$ því allar hlutrunur í tómri runu eru tómar

LCS

- Búum til töflu T af stærð nm og látum $T[i][j]$ vera lengd lengstu sameiginlegu hlutrunu $x_1, \ldots, x_i$ og $y_1, \ldots, y_j$
- Sem grunntilvik er ljóst að $T[i][0] = T[0][j] = 0$ því allar hlutrunur í tómri runu eru tómar
- Skoðum nú hvernig við getum reiknað $T[i][j]$. Ef $x_i = y_j$ þá er best að láta þau vera í hlutrununni, svo þá er $T[i][j] = T[i-1][j-1] + 1$. Ef $x_i \neq y_j$ geta þau ekki bæði verið í hlutrununni sem sameiginlegt stak, svo við þurfum að sleppa x_i eða y_j. En þá fæst að $T[i][j] = \max(T[i-1][j], T[i][j-1])$.

LCS

- Búum til töflu T af stærð nm og látum $T[i][j]$ vera lengd lengstu sameiginlegu hlutrunu $x_1, \ldots, x_i$ og $y_1, \ldots, y_j$
- Sem grunntilvik er ljóst að $T[i][0] = T[0][j] = 0$ því allar hlutrunur í tómri runu eru tómar
- Skoðum nú hvernig við getum reiknað $T[i][j]$. Ef $x_i = y_j$ þá er best að láta þau vera í hlutrununni, svo þá er $T[i][j] = T[i-1][j-1] + 1$. Ef $x_i \neq y_j$ geta þau ekki bæði verið í hlutrununni sem sameiginlegt stak, svo við þurfum að sleppa x_i eða y_j. En þá fæst að $T[i][j] = \max(T[i-1][j], T[i][j-1])$.
- Þar sem að reikna út $T[i][j]$ tekur fastan tíma gefur þetta þá af sér $\mathcal{O}(nm)$ lausn á vandamálinu!

```cpp
#include <bits/stdc++.h>
using namespace std;

int lcs(int* x, int n, int* y, int m) {
    int t[n + 1][m + 1];
    memset(t, 0, sizeof(t));
    for(int i = 1; i <= n; ++i) {
        for(int j = 1; j <= m; ++j) {
            if(x[i - 1] == y[j - 1]) {
                t[i][j] = t[i - 1][j - 1] + 1;
            } else {
                t[i][j] = max(t[i - 1][j], t[i][j - 1]);
            }
        }
    }
    return t[n][m];
}

int main() {
    int x[12] = {'a', 't', 'l', 'i', 'f', 'r', 'a', 'n', 'k', 'l', 'i', 'n'};
    int y[15] = {'b', 'e', 'r', 'g', 'u', 'r', \
        's', 'n', 'o', 'r', 'r', 'a', 's', 'o', 'n'};
    cout << lcs(x, 12, y, 15) << endl;
}
```

Keppnisforritunardæmi

- Kvik bestunardæmi eru afar algeng í forritunarkeppnum

Keppnisforritunardæmi

- Kvik bestunardæmi eru afar algeng í forritunarkeppnum
- Eiginlega alla vega 2 af 10-12 dæmum um kvika bestun

Keppnisforritunardæmi

- Kvik bestunardæmi eru afar algeng í forritunarkeppnum
- Eiginlega alla vega 2 af 10-12 dæmum um kvika bestun
- Því er gott að æfa sig vel í að læra að hugsa um hvernig megi skipta dæminu upp og hvernig eigi að geyma hlutvandamálin í töflu

Keppnisforritunardæmi

- Kvik bestunardæmi eru afar algeng í forritunarkeppnum
- Eiginlega alla vega 2 af 10-12 dæmum um kvika bestun
- Því er gott að æfa sig vel í að læra að hugsa um hvernig megi skipta dæminu upp og hvernig eigi að geyma hlutvandamálin í töflu
- Skoðum því dæmi úr keppni og leysum það saman

Shopaholic, erfiðari útgáfa

- Fáum gefin $1 \leq n \leq 2000$ og $1 \leq d \leq 20$ og svo n tölur $1 \leq p_1, \ldots, p_n \leq 10000$. p_i-in eru vörunar okkar í þeirri röð sem við erum búin að raða þeim á vörubeltið. d er fjöldi skiptibúta sem við erum með sem við getum lagt á vörubandið til að skipta kaupunum upp til að borga hlutina í stit hvoru lagi. Búðin rúnnar hlutina að næsta margfeldi af 10 krónum (5 upp á við). Hversu mikið þurfum við að borga fyrir vörurnar í minnsta lagi?

Shopaholic, erfiðari útgáfa

- Fáum gefin $1 \leq n \leq 2000$ og $1 \leq d \leq 20$ og svo n tölur $1 \leq p_1, \ldots, p_n \leq 10000$. p_i-in eru vörunar okkar í þeirri röð sem við erum búin að raða þeim á vörubeltið. d er fjöldi skiptibúta sem við erum með sem við getum lagt á vörubandið til að skipta kaupunum upp til að borga hlutina í stit hvoru lagi. Búðin rúnnar hlutina að næsta margfeldi af 10 krónum (5 upp á við). Hversu mikið þurfum við að borga fyrir vörurnar í minnsta lagi?
- Hvernig töflu viljum við búa til? Hvernig viljum við reikna gildi töflunnar?

Uppsetning töflu

- Látum $T[i][j]$ vera hvað við þurfum að borga í minnsta lagi fyrir fyrstu i vörurnar ef við höfum j skiptibúta til umráða

Uppsetning töflu

- Látum $T[i][j]$ vera hvað við þurfum að borga í minnsta lagi fyrir fyrstu i vörurnar ef við höfum j skiptibúta til umráða
- Hver eru grunntilvik okkar? Ef $i = 0$ höfum við engar vörur svo $T[0][j] = 0$. Ef $j = 0$ er svarið bara hvað við þurfum að borga mikið fyrir fyrstu i vörurnar.

Uppsetning töflu

- Látum $T[i][j]$ vera hvað við þurfum að borga í minnsta lagi fyrir fyrstu i vörurnar ef við höfum j skiptibúta til umráða
- Hver eru grunntilvik okkar? Ef $i = 0$ höfum við engar vörur svo $T[0][j] = 0$. Ef $j = 0$ er svarið bara hvað við þurfum að borga mikið fyrir fyrstu i vörurnar.
- Hvað annars? Einhver skiptibútur þarf að vera lengst til hægri af öllum skiptibútum okkar. Því getum við ítrað yfir $k = 1, \ldots, i$ og prófað að setja bútinn okkar sem er lengst til hægri þar. Ef við setjum hann þar er kostnaðurinn $T[k][j-1]$ fyrir það sem er vinstra megin við skiptibútinn en bara kostnaðurinn af restinni fyrir það sem er vinstra megin (því þar verða engir skiptibútar).

Lausnin

```cpp
#include <bits/stdc++.h>
using namespace std;

int dp[2005][25];
int pref[2005];
int vals[2005];

int myround(int n) {
    return n % 10 < 5 ? n - (n % 10) : n - (n % 10) + 10;
}

int dplookup(int i, int j) {
    if(dp[i][j] != -1) return dp[i][j];
    if(i == 0) return dp[i][j] = 0;
    if(j == 0) return dp[i][j] = myround(pref[i - 1]);
    int mn = INT_MAX;
    for(int k = 0; k <= i; ++k) {
        mn = min(mn, dplookup(k, j - 1) + myround(pref[i - 1] - pref[k - 1]));
    }
    return dp[i][j] = mn;
}

int main() {
    int n, d;
    memset(dp, -1, sizeof(dp));
    cin >> n >> d;
    for(int i = 0; i < n; ++i) cin >> vals[i];
    pref[0] = vals[0];
    for(int i = 1; i < n; ++i) pref[i] = pref[i - 1] + vals[i];
    cout << dplookup(n, d) << endl;
}
```

Nokkur atriði varðandi kvika bestun

- Í sjálfu sér er þetta kannski ekki flóknasta aðferðafræðin, en dæmi af þessari gerð má gera óguðlega erfið

Nokkur atriði varðandi kvika bestun

- Í sjálfu sér er þetta kannski ekki flóknasta aðferðafræðin, en dæmi af þessari gerð má gera óguðlega erfið
- Fyrsta sem benda má á er að það séu í raun tvær tegundir lausna, bottom-up og top-down

Nokkur atriði varðandi kvika bestun

- Í sjálfu sér er þetta kannski ekki flóknasta aðferðafræðin, en dæmi af þessari gerð má gera óguðlega erfið
- Fyrsta sem benda má á er að það séu í raun tvær tegundir lausna, bottom-up og top-down
- Í bottom up ítrum við einu sinni í gegnum töfluna og reiknum út öll gildin og sækjum svo svarið okkar (eins og við gerðum fyrir LCS)

Nokkur atriði varðandi kvika bestun

- Í sjálfu sér er þetta kannski ekki flóknasta aðferðafræðin, en dæmi af þessari gerð má gera óguðlega erfið
- Fyrsta sem benda má á er að það séu í raun tvær tegundir lausna, bottom-up og top-down
- Í bottom up ítrum við einu sinni í gegnum töfluna og reiknum út öll gildin og sækjum svo svarið okkar (eins og við gerðum fyrir LCS)
- Í bottom up getur röð lykkja skipt höfuðmáli (dæmi eftir smá)

Nokkur atriði varðandi kvika bestun

- Í sjálfu sér er þetta kannski ekki flóknasta aðferðafræðin, en dæmi af þessari gerð má gera óguðlega erfið
- Fyrsta sem benda má á er að það séu í raun tvær tegundir lausna, bottom-up og top-down
- Í bottom up ítrum við einu sinni í gegnum töfluna og reiknum út öll gildin og sækjum svo svarið okkar (eins og við gerðum fyrir LCS)
- Í bottom up getur röð lykkja skipt höfuðmáli (dæmi eftir smá)
- Í top down búum við til hjálparfall sem sækir gildi í töfluna og reiknar það endurkvæmt með því að kalla í sjálft sig ef það er ekki búið að reikna gildið nú þegar

Nokkur atriði varðandi kvika bestun

- Í sjálfu sér er þetta kannski ekki flóknasta aðferðafræðin, en dæmi af þessari gerð má gera óguðlega erfið
- Fyrsta sem benda má á er að það séu í raun tvær tegundir lausna, bottom-up og top-down
- Í bottom up ítrum við einu sinni í gegnum töfluna og reiknum út öll gildin og sækjum svo svarið okkar (eins og við gerðum fyrir LCS)
- Í bottom up getur röð lykkja skipt höfuðmáli (dæmi eftir smá)
- Í top down búum við til hjálparfall sem sækir gildi í töfluna og reiknar það endurkvæmt með því að kalla í sjálft sig ef það er ekki búið að reikna gildið nú þegar
- Það fer bara eftir aðstæðum hvort er þægilegra eða betra

Lykkjuröð í bottom up

- Gefum okkur eitthvað fylki x af heiltölum. Skoðum hver munurinn er á að finna mögulegar summur þessarra talna þegar nota má sömu töluna oft og þegar nota má hverja tölu aðeins einu sinni.

Lykkjuröð í bottom up

- Gefum okkur eitthvað fylki x af heiltölum. Skoðum hver munurinn er á að finna mögulegar summur þessarra talna þegar nota má sömu töluna oft og þegar nota má hverja tölu aðeins einu sinni.
- Gerum fylki dp þar sem $dp[i]$ er hvort rita megi i sem summu af þessum tölum.

Lykkjuröð í bottom up

- Gefum okkur eitthvað fylki x af heiltölum. Skoðum hver munurinn er á að finna mögulegar summur þessarra talna þegar nota má sömu töluna oft og þegar nota má hverja tölu aðeins einu sinni.
- Gerum fylki dp þar sem $dp[i]$ er hvort rita megi i sem summu af þessum tölum.
- Sjáum nú tvær leiðir til að reikna út gildi dp eftir því hvort endurtaka megi gildin eða ekki

Með endurtekningum

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int x[4] = {1, 5, 13, 27};
    int dp[100];
    memset(dp, 0, sizeof(dp));
    dp[0] = 1;
    for(int i = 0; i < 4; ++i) {
        for(int j = 0; j < 100; ++j) {
            if(x[i] > j) continue;
            if(dp[j - x[i]] == 0) continue;
            dp[j] = 1;
        }
    }
    cout << dp[3] << endl; // prentar 1
}
```

Án endurtekninga

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int x[4] = {1, 5, 13, 27};
    int dp[100];
    memset(dp, 0, sizeof(dp));
    dp[0] = 1;
    for(int i = 0; i < 4; ++i) {
        for(int j = 100; j >= 0; --j) {
            if(x[i] > j) continue;
            if(dp[j - x[i]] == 0) continue;
            dp[j] = 1;
        }
    }
    cout << dp[3] << endl; // prentar 0
}
```

Sækja lausn

- Stundum þarf að skila runu af tölum eða streng eða einhverskonar heila lausn í stað þess að skila bara tölu

Sækja lausn

- Stundum þarf að skila runu af tölum eða streng eða einhverskonar heila lausn í stað þess að skila bara tölu
- T.d. skila hverjar skiptingarnar voru í cent savings eða hver lengsta sameiginlega runan er í LCS

Sækja lausn

- Stundum þarf að skila runu af tölum eða streng eða einhverskonar heila lausn í stað þess að skila bara tölu
- T.d. skila hverjar skiptingarnar voru í cent savings eða hver lengsta sameiginlega runan er í LCS
- Þá er oft gott að gera aðra töflu L sem er jafnstór hinni töflunni T. Svo fyrir hvert gildi sem við setjum í T setjum við í L gildið sem við notuðum til að búa til lausnina í T

Sækja lausn

- Stundum þarf að skila runu af tölum eða streng eða einhverskonar heila lausn í stað þess að skila bara tölu
- T.d. skila hverjar skiptingarnar voru í cent savings eða hver lengsta sameiginlega runan er í LCS
- Þá er oft gott að gera aðra töflu L sem er jafnstór hinni töflunni T. Svo fyrir hvert gildi sem við setjum í T setjum við í L gildið sem við notuðum til að búa til lausnina í T
- Þá getum við ítrað í gegnum L til að fá lausnina okkar, tökum dæmi með summu talna eins og að ofan

Sækja lausn sýnidæmi

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int x[4] = {1, 5, 13, 27};
    int dp[100], last[100];
    memset(dp, 0, sizeof(dp));
    dp[0] = 1;
    for(int i = 0; i < 4; ++i) {
        for(int j = 99; j >= 0; --j) {
            if(x[i] > j || !dp[j - x[i]] || dp[j]) continue;
            dp[j] = 1, last[j] = i;
        }
    }
    int s = 41;
    cout << s << " = ";
    while(s > 0) {
        cout << x[last[s]];
        s -= x[last[s]];
        if(s != 0) cout << " + ";
    }
    cout << endl;
}
```