Dinamičko programiranje

Knapsack

Dinamičko programiranje (*Dynamic Programing*) je strategija (metoda) kojoj je osnovno načelo postupno graditi rješenje složenog problema koristeći rješenja istovrsnih manje složenih problema (*bottom-up* pristup)

– primjenjiva kada se podproblemi "preklapaju" (*overlapping subproblems*)

– za razliku od podijeli pa vladaj (*divide and conquer*) strategije koja problem rješava *top-down* pristupom, ne ponavlja već obavljeni posao jer maksimalno iskorištava rezultate prethodnih koraka

- "programiranje" u kontekstu dinamičkog programiranja ne znači programiranje u uobičajenom smislu, nego je to samo naziv za strategiju (planski proveden postupak)
- u provedbi najčešće tablična metoda; *memoization*

- tipična primjena je u optimizacijskim problemima u kojima se do konačnog rješenja dolazi tek nakon niza odluka, pri čemu nakon svake odluke problem ostaje istovrstan, samo manje složenosti, a konačno rješenje se dobiva na temelju optimalnih rješenja podproblema koji nastaju nakon svake pojedine odluke i čija su rješenja najbolja moguća s obzirom na do tada postignuto stanje (Bellmanovo načelo optimalnosti; *Bellman's Principle of Optimality*)
- zbog postupne izgradnje konačnog rješenja korištenjem rješenja pod ... podproblema, dinamičko programiranje primjenjivo je samo na probleme rekurzivnog karaktera
- vrlo slično lakomoj (pohlepnoj; *greedy*) strategiji, ali ne i isto

Da bi bio rješiv po načelu dinamičkog programiranja, problem mora zadovoljavati sljedeća dva uvjeta:

1. optimalna podstruktura (*optimal substructure*)
   - svojstvo problema da optimalno rješenje sadrži u sebi optimalna rješenja nezavisnih podproblema (sastoji se od njih)
     - to samo po sebi nije dovoljno jer je inače i svojstvo koje upućuje na primjenu "lakome" (*greedy*) strategije
     - dobar primjer je problem traženja najkraćeg puta (Dijkstrin algoritam, lakoma strategija); najkraći put između dva vrha sastoji se od najkraćeg puta od polaznog vrha do nekog međuvrha i najkraćeg puta od međuvrha do završnog vrha $\Leftrightarrow$ optimalna rješenja dvaju nezavisnih podproblema
2. preklopljenost podproblema (*overlapping subproblems*)
   - svojstvo problema da njegovo rješavanje zahtijeva (vodi u) višekratno rješavanje identičnih pod ... podproblema pa se prethodni rezultati mogu iskoristiti za brže rješavanje kasnijih koraka (primjer: Fibbonacievi brojevi)
   - pod ... podproblemi i dalje moraju biti nezavisni

Rješavanje problema primjenom dinamičkog programiranja podrazumijeva četiri osnovna koraka:

1. uočiti strukturu optimalnog rješenja
   a) Zadovoljava li problem uvjet optimalne podstrukture?
   b) Jesu li podproblemi preklopljeni?
   - ovo je korak u kojem procjenjujemo rješivost problema dinamičkim programiranjem
   - potpuno ovisi o intuiciji
2. postaviti rekurzivnu formulu za izračunavanje *vrijednosti* konačnog rješenja
   - "vrijednost" je veličina (funkcija) koja se optimira, tj. čiji se minimum ili maksimum traži
3. izračunati optimalnu *vrijednost* konačnog rješenja koristeći rješenja manjih podproblema (*bottom-up* pristup, *memoization*)
4. izgraditi (konstruirati) konačno rješenje (odrediti optimalni skup odluka)
   - ovdje je problem već riješen (znamo optimalnu vrijednost) pa se ovaj korak ne obavlja uvijek jer je za njegovo ostvarenje najčešće potrebno čuvati dodatne informacije tijekom prethodna tri koraka

<u>Primjer:</u> problem naprtnjače (*Knapsack problem*)

- 0,1 ili *integer* inačica (za razliku od *fractional* inačice)

Lopov ima samo jednu vreću volumena (kapaciteta) C=12 u koju ne stanu sve stvari koje su mu dostupne. Kolika je najveća ukupna vrijednost koju može ukrasti ako sve što uzme mora stati u vreću? Koje su to stvari (koji je odabir najbolji mogući, tj. optimalan)?

	stvar 1	stvar 2	stvar 3	stvar 4
vrijednost	8	6	12	5
volumen ("cijena")	5	2	7	3

- Dakle, problem je optimizirati odabir ukradenih stvari; tražimo maksimum vrijednosti za zadani volumen vreće, pri čemu je "cijena" svake stvari (odluke) prostor koji ona zauzima.

Za ovako mali broj stvari problem je dovoljno jednostavan da ga možemo riješiti napamet i lako nalazimo:

- najveća vrijednost koju lopov može ukrasti $v_{max} = 23$
- ukupna cijena (volumen) stvari koje čine najbolji odabir je točno $c = 12$
  (znači, u ovom slučaju lopov može potpuno iskoristiti vreću)
- najbolji odabir: druga, treća i četvrta stvar

	stvar 1	stvar 2	stvar 3	stvar 4
vrijednost	8	6	12	5
volumen ("cijena")	5	2	7	3

Za veći broj stvari intuitivno rješavanje postaje nemoguće i trebamo algoritam koji će nas sigurno dovesti do najboljeg mogućeg rješenja!

Ali nije važno samo doći do rješenja – više nego ukrasti najviše što može, lopov bi želio pobjeći prije dolaska policije!

- već su dvojica njegovih "kolega" prije njega pokušala istu krađu, ali jedan je kasnije ustanovio da nije ukrao naviše što je mogao, a drugi je "zaglavio". Bilo je to ovako...

Prvi lopov je bio prelakom – uzimao je redom najvrijednije stvari. Tako je uzeo prvu i treću stvar, ukupne vrijednosti v = 20 i otišao jer mu više ništa nije stalo u vreću. Tek je kasnije ustanovio da je mogao obaviti i bolji "posao"... ☹

Drugi se "u slobodno vrijeme" bavi programiranjem pa je znao algoritam kojim će sigurno naći najbolje rješenje – isprobavao je sve kombinacije stvari i izračunavao njihovu ukupnu vrijednost i volumen. Međutim, kombinacija je bilo koliko i podskupova u skupu od N stvari, dakle 2^N. Računanje je trajalo (pre)dugo i toliko ga je zaokupilo da je zaboravio kako je u tuđoj kući, a policija na putu ... ☹

Poučen iskustvom svojih prethodnika, ovaj je lopov unaprijed razradio prilično brz algoritam – dinamičkim programiranjem.

1. uočiti strukturu optimalnog (konačnog) rješenja
   - Recimo da znamo najbolje moguće rješenje kad promatramo k stvari i cijeli raspoloživi kapacitet. Označimo skup stvari koje čine najbolji izbor s Ω.
   - Ključno je primijetiti da ako iz skupa Ω uklonimo samo jednu (bilo koju) stvar, preostale stvari sigurno čine najbolji mogući izbor iz skupa od k–1 stvari, ali za kapacitet vreće umanjen za volumen izdvojene stvari.

   Dokaz: kontradikcija. Recimo da nakon izdvajanja jedne stvari (označimo ju s A) iz skupa Ω preostale stvari čine skup S i nisu najbolji mogući izbor za preostali kapacitet. To znači da se iz skupa od k–1 stvari može odabrati skup T nekih drugih stvari koje će ukupno imati veću vrijednost nego one iz S i pritom neće zauzeti više od preostalog kapaciteta vreće. No, tada skup {T, A} ukupno daje veću vrijednost nego Ω kada se promatra svih k stvari i cijeli raspoloživi kapacitet, a to je protivno pretpostavci da je Ω najbolje moguće rješenje. ■

Zaključak: najbolje moguće rješenje većeg problema se sastoji od najboljih mogućih rješenja manjih istovrsnih problema $\Rightarrow$ optimalna podstruktura.
Rješenje za k stvari se dobiva koristeći rješenja za k–1 stvari, ono za k–1 stvari iz rješenja za k–2 stvari itd.. $\Rightarrow$ podproblemi se preklapaju.

2. postaviti rekurzivnu formulu za izračunavanje konačnog rješenja, tj. *optimalne vrijednosti ciljne funkcije*

- Promatranjem bilo koje stvari, recimo k-te, raspoloživog skupa S, uočavamo da konačno rješenje može biti samo dvojako – ono ili uključuje ili ne uključuje promatranu stvar.
- Ako ju ne uključuje, onda je rješenje problema za zadanu cijenu c i cijeli skup S jednako optimalnom rješenju za cijenu c i skup bez k-te stvari $S\backslash\{k\}$. Simbolički, $v_k(c) = v_{-k}(c)$, gdje $v_{-k}(c)$ označava najveću moguću vrijednost za cijenu c kada promatramo skup bez k-te stvari $S\backslash\{k\}$.
- Ako ju uključuje, onda je najveća ostvariva vrijednost za cijenu c i skup s k-tom stvari jednaka optimalnom rješenju za skup bez k-te stvari i najveću dozvoljenu cijenu umanjenu za cijenu k-te stvari, tj. za cijenu c–cost(k), uvećana za vrijednost k-te stvari. Simbolički, $v_k(c) = v_{-k}[c\text{–cost}(k)] + \text{value}(k)$.
- Iz prethodnih razmatranja slijedi da k-ta stvar ulazi u najbolji izbor ako je $[v_{k-1}(c\text{–cost}(k)) + \text{value}(k)] > v_{k-1}(c)$.

Tražena rekurzivna formula glasi:

$$v_k(c) = \max\{v_{k-1}(c),\ v_{k-1}[c\text{–cost}(k)] + \text{value}(k)\} \quad ; \quad v_0(\cdot) = 0\ .$$

3. izračunati *vrijednost* konačnog rješenja koristeći rješenja manjih podproblema (*bottom-up* pristup + *memoization*)
   - uzimati u razmatranje jednu po jednu stvar i donositi odluke primjenom rekurzivne formule
   - algoritam se ubrzava pohranom prethodnih rješenja u tablicu (puni se po stupcima, tj. stvarima); redci = cijene (zauzeti volumeni), stupci = stvari

	stvar 1	stvar 2	stvar 3	stvar 4
1	$v_1(1) = 0$	$v_2(1) = 0$	0	0
2	$v_1(2) = 0$	$v_2(2) = 6$	6	6
3	0	6	6	6
4	0	6	6	6
5	(+8) = 8	8	8	11
6	8	8	8	11
7	8	14	14	14
8	8	14	14	14
9	8	14	18	18
10	8	14	18	19
11	8	14	18	19
12	8	14	20	23

	1	2	3	4
v	8	6	12	5
c	5	2	7	3

$v_0(\cdot) = 0$

$v_k(c) = \max\{v_{k-1}(c),$
$v_{k-1}[c - \text{cost}(k)] + \text{value}(k)\}$

Svako polje tablice sadrži najbolje rješenje za cijenu određenu indeksom redka i broj stvari određen indeksom stupca!

4. izgraditi (konstruirati) konačno rješenje (odrediti optimalni skup odluka)
   - najbolji odabir stvari lako se može očitati iz tablice; redak sa zadanom cijenom (kapacitetom vreće) pregledava se s desna na lijevo; najveća ostvariva vrijednost je u zadnjem stupcu

	stvar 1	stvar 2	stvar 3	stvar 4
1	$v_1(1) = 0$	$v_2(1) = 0$	0	0
2	$v_1(2) = 0$	$v_2(2) = 6$	6	6
3	0	6	6	6
4	0	6	6	6
5	(+8) = 8	8	8	11
6	8	8	8	11
7	8	14	14	14
8	8	14	14	14
9	8	14	18	18
10	8	14	18	19
11	8	14	18	19
12	8	14	20	23

- ako je postignuta dodavanjem zadnje stvari, onda je različita od vrijednosti u polju s lijeva i zadnja stvar ulazi u najbolji odabir; sljedeće polje je ono iz kojeg dolazi crvena strelica
- ako zadnja stvar nije u najboljem odabiru, vrijednost u polju s lijeva je jednaka onoj u promatranom polju; prelazi se u polje s lijeva (plava strelica) i ponavlja razmatranje

- programsko određivanje optimalnog odabira u složenijim problemima zahtijeva održavanje popisa s trenutačnim odabirom za svako polje tablice, dakle još jednu tablicu veličine $c_{\max} \times n$
- u ovom primjeru dovoljno je samo upisivati oznake je li u nekom koraku trenutačno promatrana stvar ušla u izbor ili ne (drugim riječima, oznake vrste strelica; npr. =*true* za kosu strelicu i =*false* za vodoravnu)
- po završetku algoritma, krene se od zadnjeg polja tablice, indeksa $[c_{\max}, n]$, i pogleda je li u tom koraku zadnja stvar ušla u najbolji izbor (oznaka *true*) ili nije (oznaka *false*), a u oba slučaja prelazi se u polje iz kojeg se došlo tijekom popunjavanja tablice i ponavlja razmatranje
- ako je zadnja stvar ušla u izbor, sljedeće polje bit će polje indeksa $[c_{\max}–\text{cost}(n), n–1]$
- ako zadnja stvar nije ušla u izbor, sljedeće polje bit će prvo polje s lijeva, dakle indeksa $[c_{\max}, n–1]$

Pseudo-kod (C# sintaksa) rješenja Knapsack problema dinamičkim programiranjem; složenost $O(C_{max} \cdot N)$.

```
Knapsack (items, Cmax, value[], cost[])
form table w[Cmax, items] and table decisions[Cmax, items];
initialisation: w[0, *] = 0 and w[*, 0] = 0; //nulti redak i stupac
initialisation: decisions[*,*] = false;
for (k = 1; k<=items; ++k)                  //Za sve stvari ...
   for (c = 1; c<=Cmax; ++c)                //Za sve cijene ...
   {  kNo = w[c,k-1];      //Vrijednost bez k-te, za istu cijenu.
      if (c >= cost[k])    //Ako je dozvoljena cijena (preostali kapacitet)...
         kYes = w[c-cost[k],k-1] + value[k];//Vrijednost s k-tom.
      else                 //Ako je k-ta preskupa već sama po sebi,
         kYes = kNo;       //najveća vrijednost s k-tom = ona bez k-te.
      if (kYes > kNo)
      {  w[c,k] = kYes;    //k-ta ulazi u najbolji izbor
         decisions[c,k] = true;  }
      else
         w[c,k] = kNo;      //k-ta ne ulazi u najbolji izbor
   }
```

Ispis (unazad) odabranih elemenata (C# sintaksa).

Ispis (bool[,] decisions)

```
int c = maxcost, k = items;
//'maxcost' i 'items' moraju biti vidljive ovoj funkciji ili
//ih treba proslijediti kao ulazne argumente
do
{  if (decisions[c, k] == true)
   {    Console.Out.Write("{0,4}", k);
        c -= costs[k];     }
   --k;
} while (c > 0 && k > 0);
```

Izravna primjena rekurzivne formule bila bi primjer “klasične” podijeli pa vladaj strategije i takvo bi rješenje bilo osjetno sporije od tabličnog jer izračunavanje najboljih odabira za vreće većeg kapaciteta iznova zahtijeva obradu istih podproblema. Na primjer, oba kapaciteta 3 i 4, svaki za sebe, zahtijevaju obradu kapaciteta 1 i 2, što znači da bi se manji kapaciteti rješavali više puta pa bi program bio izrazito “redundantan”.

Nedostatci obične rekurzije u ovom i drugim problemima koji se mogu rješavati dinamičkim programiranjem (znači tablično) izravna su posljedica preklapanja podproblema koje treba riješiti da se nađe konačno rješenje.

Dobra vježba: analiza programa za izračunavanje Fibbonacievih brojeva iz predmeta “Algoritmi i strukture podataka” (poglavlje o rekurzivnom programiranju).

- analizirajte koliko puta se ponavlja ista obrada

Pseudo-kod (C# sintaksa) rekurzivnog rješenja Knapsack problema.

```
KnapsackRec (c,k)
if (c > 0 && k > 0)        //Ako to je element za razmatranje ...
{
   kNo = KnapsackRec(c,k-1);
   if (c >= cost[k])   //Ako je preostali maksimum cijene dovoljan...
      kYes = KnapsackRec(k-1, c-cost[k]) + value[k];
   else
       kYes = kNo;
   if (kYes > kNo)
   {  w[c,k] = kYes;    //Upis ostvarive vrijednosti u tablicu W(c,k).
      decisions[c] = true;       //k-ta ulazi u najbolji izbor
      return kYes;  }
   else
   {  w[c,k] = kNo;      //Upis ostvarive vrijednosti u tablicu W(c,k).
      decisions[c] = false;    //k-ta ispada iz najboljeg izbora
      return kNo;   }
}
else
   return 0;
```

Dodatak: skraćenje postupka

- pogodno za ljude, relativno nespretno za programiranje

	1	2	3	4
v	8	6	12	5
c	5	2	7	3

sort po c →

	2	4	1	3
v	6	5	8	12
c	2	3	5	7

	stvar 2	stvar 4	stvar 1	stvar 3
2	6	6	6	6
3	... 6	6	6	6
5	... 6	11	11	11
7	... 6	... 11	14	14
8	... 6	... 11	14	14
9	... 6	... 11	14	18
10	... 6	... 11	19	19
12	... 6	... 11	... 19	23

$v_0(\cdot) = 0$

$v_k(c) = \max\{v_{k-1}(c),\ v_{k-1}[c - \text{cost}(k)] + \text{value}(k)\}$