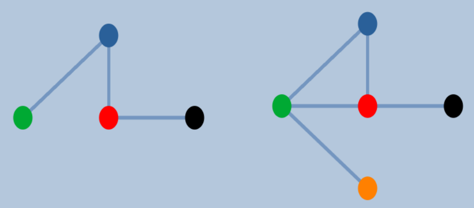
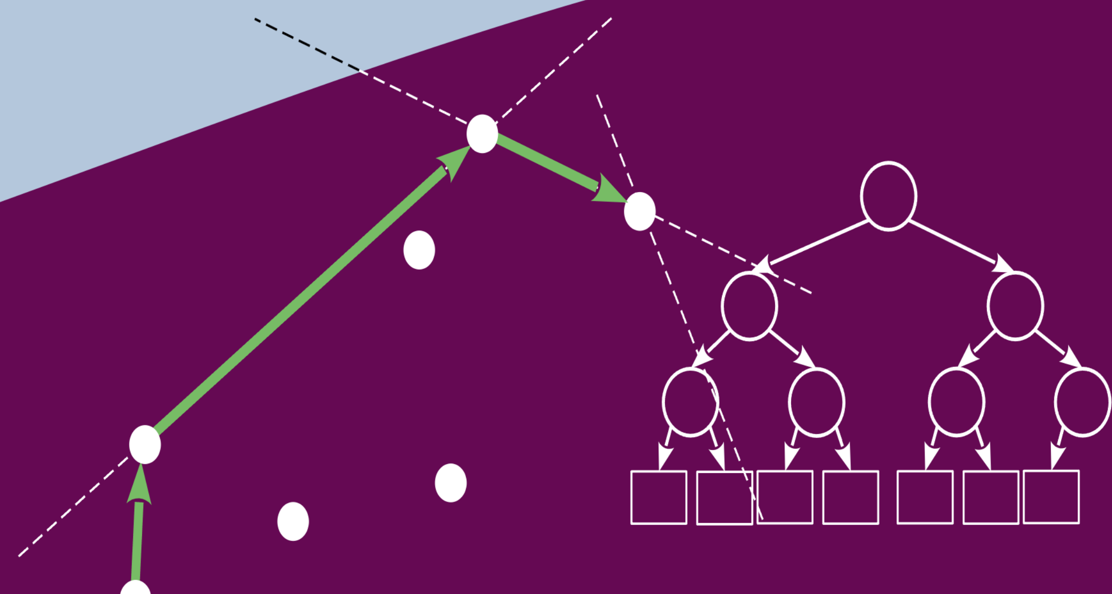
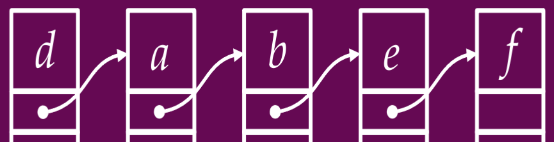
Napredni algoritmi i strukture podataka

Tjedan 6: Dinamičko programiranje







Creative Commons

slobodno smijete:

dijeliti — umnožavati, distribuirati i javnosti priopćavati djelo

prerađivati djelo

pod sljedećim uvjetima:

imenovanje: morate priznati i označiti autorstvo djela na način kako je specificirao autor ili davatelj licence (ali ne način koji bi sugerirao da Vi ili Vaše korištenje njegova djela imate njegovu izravnu podršku).

nekomercijalno: ovo djelo ne smijete koristiti u komercijalne svrhe.

dijeli pod istim uvjetima: ako ovo djelo izmijenite, preoblikujete ili stvarate koristeći ga, preradu možete distribuirati samo pod licencom koja je ista ili slična ovoj.

U slučaju daljnjeg korištenja ili distribuiranja morate drugima jasno dati do znanja licencne uvjete ovog djela.
Od svakog od gornjih uvjeta moguće je odstupiti, ako dobijete dopuštenje nositelja autorskog prava.
Ništa u ovoj licenci ne narušava ili ograničava autorova moralna prava. *Tekst licence preuzet je s http://creativecommons.org/*

Uvod (1)

- **Dinamičko programiranje** (*Dynamic Programing*)
  - Metoda (strategija) 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

Uvod (2)

- “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*

Uvod (3)

- 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)

Uvod (4)

- Zbog postupne izgradnje konačnog rješenja korištenjem rješenja pod ... podproblema, dinamičko programiranje primjenjivo je samo na probleme rekurzivnog karaktera

- Vrlo slično pohlepnoj (*greedy*) strategiji
  - Ova strategija traženja rješenja problema biti će detaljno objašnjena u sljedećim predavanjima…

Uvod (5)

- Zaključno: dinamičko programiranje rješava probleme kombinacijom rješavanja podproblema.
  - Za razliku od pristupa „divide and conquer” dinamičko programiranje primjenjuje se kod problema čiji podproblemi nisu međusobno nezavisni, već imaju neke zajedničke pod .. podprobleme.
  - U takvim slučajevima „divide and conquer” algoritmi nepotrebno bi više puta rješavali iste podprobleme.
  - Tipično algoritmi razvijeni na principima dinamičkog programiranja rješavaju svaki podproblem samo jednom i čuvaju njegovo rješenje u tablici (u privremenoj memoriji; implementacija na računalima).
  - Izbjegava se ponovno rješavanje istog podproblema više puta

Razvoj i primjena u praksi (1)

- Dinamičko programiranje (DP) je razvijeno sa svrhom optimalizacije velikih i složenih tehnoloških sustava za proizvodnju raznih industrijskih proizvoda ili komponenti te energetiku, a koji se mogu podijeliti na podsustave („podjeli pa vladaj”).

- Tijekom dinamičkog programiranja, pri postupnoj optimalizaciji podsustava uzimaju se u obzir njihova uzajamna djelovanja.

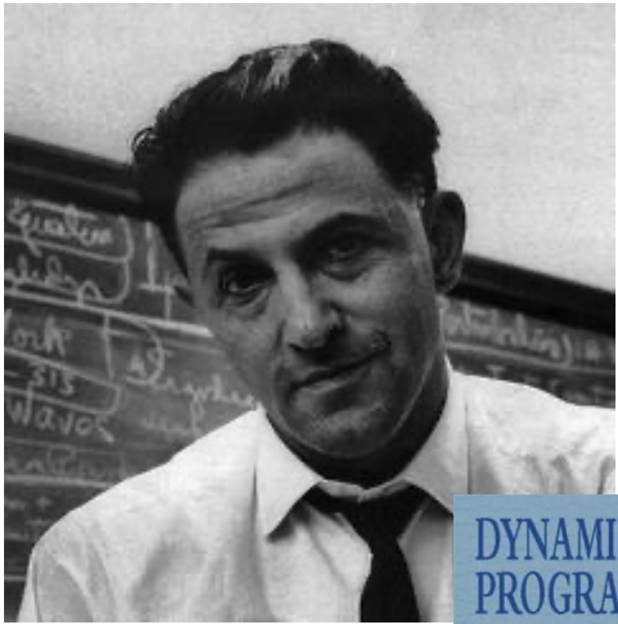
- Izbor izvršen u svakom koraku (stupnju) doprinosi optimumu sustava – kombinaciji optimuma podsustava.

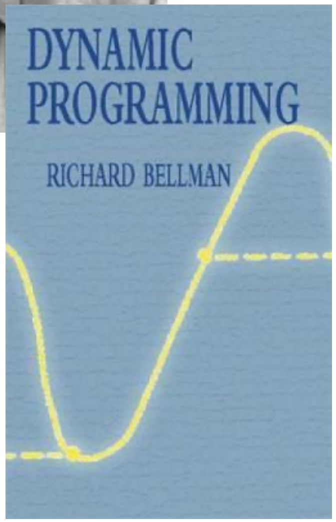
Razvoj i primjena u praksi (2)

- Dinamičkog programiranje prvi je detaljno obradio **Richard E Bellman** 1957.

- Od tada se dinamičko programiranje koristi u matematici, znanosti, inženjerstvu, biomatematici, medicini, ekonomiji, informatici i umjetnoj inteligenciji.

- Primjena dinamičkog programiranja se širi s razvojem metoda i postupaka ANN, dubinske analize podataka, otkrivanja znanja (*data mining*), *soft computing* i drugim područjima umjetne inteligencije.





Razvoj i primjena u praksi (3)

- Sljedeća četiri formalna postupaka dinamičkog programiranja često se primjenjuju u praksi za rješavanje raznih problema:
  1. jednodimenzionalna raspodjela
  2. dvodimenzionalna raspodjela
  3. najkraći put
  4. dinamika zamjena opreme

Značajke problema (1)

- Da bi bio rješiv po načelu dinamičkog programiranja, problem mora zadovoljavati sljedeća dva uvjeta:
  - 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 "pohlepne" (*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 ⇔ optimalna rješenja dvaju nezavisnih podproblema

Značajke problema (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)
  - svi pod ... pod ... podproblemi i dalje moraju biti nezavisni

Optimalna podstruktura

Preklopljenost podproblema

Rješavanje problema (1)

- 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

Rješavanje problema (2)

3. Izračunati optimalnu vrijednost konačnog rješenja koristeći rješenja manjih podproblema (korištenje *bottom-up* pristup i *memoization* postupka)

4. Izgraditi (konstruirati) konačno rješenje (odrediti optimalni skup odluka)
   - do ovog koraka problem je 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

Rješavanje problema (3)

1 Uočiti strukturu optimalnog rješenja

2 Postaviti rekurzivnu formulu za izračunavanje konačnog rješenja

3 Izračunati vrijednost konačnog rješenja manjih podproblema

4 Izgraditi konačno rješenje

Ne izvršava se uvijek

Memoization (1)

- Primjer Fibonaccijev niz:
  - $F_1 = F_2 = 1$; $F_n = F_{n-1} + F_{n-2}, n > 2$
- Naivna implementacija:

```
if n ≤ 2 then
    return 1;
end
return (Fib(n − 1) + Fib(n − 2));
```

Funkcija $Fib(n)$ za računanje n-tog Fibonaccijevog broja

- Za izračun člana F_n funkcije $Fib(m)$ za $m < n$ se pozivaju veći broj puta, pa tako za $n > 5$ vrijedi:

$$
\begin{aligned}
Fib(5) &= Fib(4) + Fib(3) \\
&= (Fib(3) + Fib(2)) + (Fib(2) + Fib(1)) \\
&= ((Fib(2) + Fib(1)) + Fib(2)) + (Fib(2) + Fib(1))
\end{aligned}
$$

Memoization (2)

- Korištenjem postupka memoizacije nije potrebno ulaziti u rekurziju i računati članove niza koji su već izračunati (i poznati).
  - Na početku inicijaliziramo sve elemente niza na -1 (označavamo ih kao neizračunate)

if $fib[n] \neq -1$ **then**
    return $fib[n]$;
end
if $n \leq 2$ **then**
    $fib[n] = 1$;
end
$fib[n] = Fib(n-1) + Fib(n-2)$;
return $fib[n]$;

Funkcija $Fib(n)$ za računanje n-tog Fibonaccijevog broja korištenjem postupka memoizacije

Primjer 1

- Odrediti faktorijel broja 3 (faktorijel broja n matematička je funkcija kojom se sukcesivno izračunava proizvod brojeva 1…n), uz korištenje rekurzije funkcije.

- Rješenje: $n! = 1 \circ 2 \circ \ldots \circ (n-1) \circ n$

  1. korak: $1! = 1$
  2. korak: $2! = 2 \circ 1! = 2 \circ 1 = 2$
  3. korak: $3! = 3 \circ 2! = 3 \circ 2 = 6$

- Treba uočiti da se u svakom koraku koristi rezultat prethodnog koraka – rekurzija funkcije (*strelice*). Opći matematički opis rješavanja bio bi:

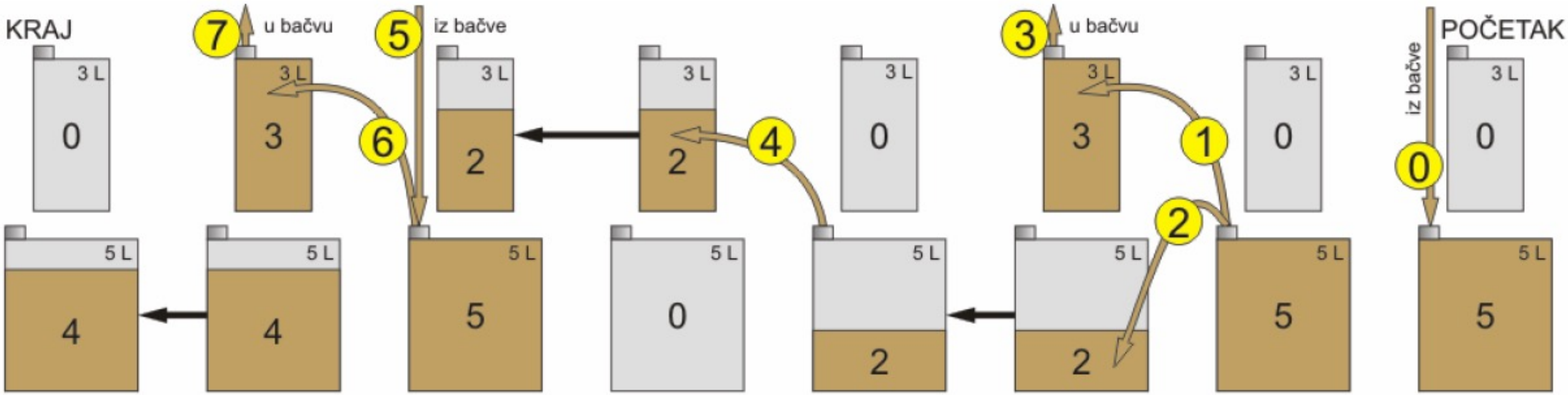
  1. korak: $f_{1,1} = 1$ (*temeljni slučaj*)
  2. korak: $f_{2,2} = 2 \circ f_{1,1}$ (*pravilo rekurzije*)

  $f_{k,j} = j \circ f_{(k-1),(j-1)} \quad j = 1,2, \ldots n$

  $j_{max} = 3$

Primjer 2 (1)

- Specijalno sintetičko mazivo ulje, pakirano u bačve od 50 L (litra), izdaje se iz skladišta tvornice na litre. Radnik održavanja dolazi u skladište s kantom od 5 L uzeti 4 L ulja, a skladištar ima kantu od 3 L. Volumen ulja možemo mjeriti s potpuno punom ili praznom kantom.



Primjer 2 (2)

- Rješenje:

- Problem se rješava u koraku m – 1 (rješavanje počinje s kraja problema) odlijevanjem 1 L ulja iz „velike“ kante s 5 L ulja u „malu“ kantu s 2 L ulja. U zadnjem se koraku (m) 3 L ulja vraća iz male kante u bačvu i odnosi velika kanta od 5 L s 4 L ulja (preostala nakon odlijevanja).

- Analizom unazad dolazi se do prvog koraka (i = 1), u kome se velika kanta puni s 5 L ulja, potom se mala kanta napuni s 3 L te velikoj kanti ostaje 2 L. Iz male se kante ulje vraća u bačvu ….

Problem naprtnjače

(*Knapsack problem*)

0-1 problem naprtnjače (*Knapsack problem*)

- Neka imamo spremnik nekog konačnog kapaciteta i skup elemenata različite težine i vrijednosti. Potrebno je odabrati podskup predmeta tako da je njihova ukupna vrijednost maksimalna, a ukupna težina manja ili jednaka kapacitetu spremnika.

- Koji je to podskup?
  - Koji su to predmeti koje stanu u naprtnjaču (koji je odabir najbolji mogući, tj. optimalan)?

- 0-1 ili *integer* inačica problema naprtnjače

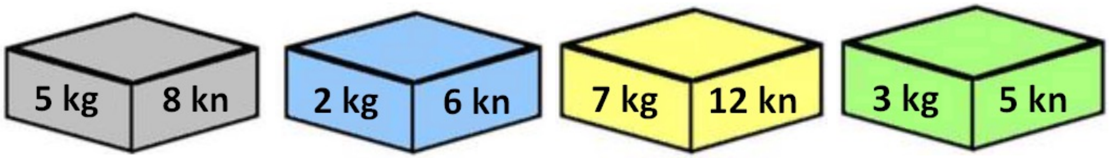
Vrste Knapsack problema

razlučiti (zasebni) problemi kombinatorne optimizacije

- 0-1 ili *integer* Knapsack problem
  - u literaturi se još navodi kao „0,1 knapsack” ili „0-1 knapsack”
  - predmeti s kojima se raspolaže u problemu mogu samo odabrati ili ne odabrati, te je neki predmet moguće odabrati samo jednom
- inačica 0-1 knapsack s ponavljanjem
  - ograničeni Knapsack problem (*bounded knapsack problem*, BKP)
    - svaki predmet ima ograničen broj identičnih kopija
      - povećanjem broja kopija predmeta otežava se problem
  - neograničeni Knapsack problem (*unbounded knapsack problem*, UKP)
    - broj kopija svakog predmeta je neograničen
      - zbog neograničenog broja kopija predmeta, UKP je još složeniji i teži za riješiti
- *fractional* inačica
  - rješava se *greedy* algoritmom
- višedimenzionalni Knapsack problem (*multidimensional Knapsack problem*, d-KP)
- višestruki Knapsack problem (*multiple knapsack problem*, MKP
- kvadratni Knapsack problem (*quadratic knapsack problem*, QKP)

Primjer: problem naprtnjače (*Knapsack problem*)

- Lopov ima samo jednu vreću kapaciteta (ili volumena) C=12 kg u koju ne stanu svi predmeti koji su mu dostupni. Kolika je najveća ukupna vrijednost koju može ukrasti ako sve što uzme mora stati u vreću?
  - Koji su to predmeti (koji je odabir najbolji mogući, tj. optimalan)?



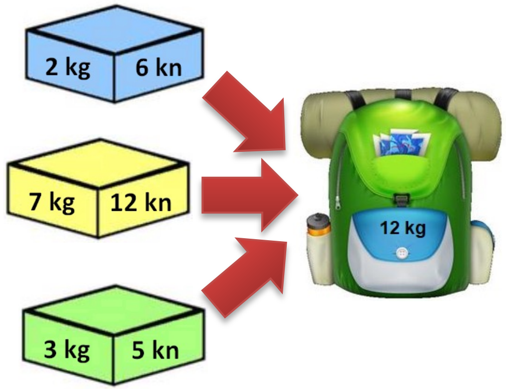
	Predmet 1	Predmet 2	Predmet 3	Predmet 4
Vrijednost	8	6	12	5
Volumen („cijena”)	5	2	7	3

- Dakle, problem je optimizirati odabir ukradenih predmeta; tražimo maksimalnu vrijednost za zadani volumen vreće, pri čemu je „cijena” svakog predmeta (odluke) prostor koji ona zauzima.

Primjer: problem naprtnjače (*Knapsack problem*)

- Za ovako mali broj predmeta problem je dovoljno jednostavan da ga možemo riješiti napamet i lako nalazimo:
  - Najveća vrijednost koju lopov može ukrasti $v_{max} = 23$
  - Ukupna cijena (volumen) predmeta koje čine najbolji odabir je točno $c = 12$ (prema tome, u ovom slučaju lopov može potpuno iskoristiti vreću)
  - Najbolji odabir: **drugi**, **treći** i **četvrti** predmet

	Predmet 1	Predmet 2	Predmet 3	Predmet 4
Vrijednost	8	**6**	**12**	**5**
Volumen („cijena”)	5	2	7	3



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

Primjer: problem naprtnjače (*Knapsack problem*)

- 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 najviše što je mogao, a drugi je „zaglavio”. Bilo je to ovako…

- Prvi lopov je bio pohlepan – uzimao je redom samo najvrijednije predmete.
  - Tako je uzeo prvi i treći predmet, ukupne vrijednosti $v = 20$ i otišao jer mu više nije moglo stati u vreću. Tek je kasnije ustanovio da je mogao obaviti i bolji „posao”…

Primjer: problem naprtnjače (*Knapsack problem*)

- Drugi se „u slobodno vrijeme” bavi programiranjem pa je znao algoritam koji će sigurno naći najbolje rješenje – isprobavao je sve kombinacije predmeta i izračunao njihovu ukupnu vrijednost i volumen. Međutim, kombinacija je bilo koliko i podskupova u skupu od N predmeta, 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.

Primjer: problem naprtnjače (*Knapsack problem*)

- Uočiti strukturu optimalnog (konačnog) rješenja:
  - Recimo da znamo najbolje moguće rješenje kad promatramo *k* predmeta i cijeli raspoloživi kapacitet.
  - Označimo skup predmeta koji čine najbolji izbor s Ω
  - Ključno je primijetiti da ako iz skupa Ω uklonimo samo jedan (bilo koji) predmet, preostali predmeti sigurno čine najbolji mogući izbor iz skupa od $k-1$ predmeta, ali za kapacitet vreće umanjen za volumen izdvojenog predmeta.

Primjer: problem naprtnjače (*Knapsack problem*)

- Dokaz: kontradikcija. Recimo da nakon izdvajanja jednog predmeta (označimo ga s A) iz skupa Ω preostali predmeti čine skup S i nisu najbolji mogući izbor za preostali kapacitet. To znači da se iz skupa od $k-1$ predmeta može odabrati skup T nekih drugih predmeta koji 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 predmeta 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 predmeta se dobiva koristeći rješenja za $k-1$ predmeta, ono za $k-1$ predmeta iz rješenja za $k-2$ predmeta itd. $\Rightarrow$ **podproblemi se preklapaju**.

Primjer: problem naprtnjače (*Knapsack problem*)

- Postaviti rekurzivnu formulu za izračunavanje konačnog rješenja, tj. optimalne vrijednosti ciljne funkcije

  - Promatranjem bilo kog predmeta, recimo k-tog, raspoloživog skupa S, uočavamo da konačno rješenje može biti samo dvojako: ono ili uključuje ili ne uključuje promatrani predmet

  - 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-tog predmeta $S\backslash\{k\}$. Simbolički, $v_k(c) = v_{-k}(c)$, gdje $v_{-k}(c)$ označava najveću moguću vrijednost za cijenu c kada promatramo skup bez k-tog predmeta $S\backslash\{k\}$.

Primjer: problem naprtnjače (*Knapsack problem*)

nastavak

- Ako ju uključuje, onda je najveća ostvariva vrijednost za cijenu c i skup s k-tim predmetom jednaka optimalnom rješenju za skup bez k-tog predmeta i najveću dozvoljenu cijenu umanjenu za cijenu k-tog predmeta, tj. za cijenu $c - cost(k)$, uvećana za vrijednost k-tog predmeta. Simbolički, $v_k(c) = v_{-k}[c - cost(k)] + value(k)$.

- Iz prethodnih razmatranja slijedi da *k*-ti predmet ulazi u najbolji izbor ako je $[v_{k-1}\big(c - cost(k)\big) + value(k)] > v_{k-1}(c)$.

- Tražena rekurzivna formula glasi:

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

Primjer: problem naprtnjače (*Knapsack problem*)

- Izračunati vrijednost konačnog rješenja koristeći rješenja manjih podproblema (*bottom-up* pristup + *memoization*)
  - Uzimati u razmatranje jedan po jedan predmeti i donositi odluke primjenom rekurzivne formule
  - Algoritam se ubrzava pohranom prethodnih rješenja u tablicu (puni se po stupcima, tj. predmetima ili stvarima):
    - Redci tablice = cijene (zauzeti volumeni)
    - Stupci tablice = predmeti ili stvari

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	–	–	–
2	–	–	–	–
3	–	–	–	–
4	–	–	–	–
5	–	–	–	–
6	–	–	–	–
7	–	–	–	–
8	–	–	–	–
9	–	–	–	–
10	–	–	–	–
11	–	–	–	–
12	–	–	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	–	–	–
2	$v_1(2) = 0$	–	–	–
3	–	–	–	–
4	–	–	–	–
5	–	–	–	–
6	–	–	–	–
7	–	–	–	–
8	–	–	–	–
9	–	–	–	–
10	–	–	–	–
11	–	–	–	–
12	–	–	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	–	–	–
2	$v_1(2) = 0$	–	–	–
3	0	–	–	–
4	0	–	–	–
5	(+8)=8	–	–	–
6	–	–	–	–
7	–	–	–	–
8	–	–	–	–
9	–	–	–	–
10	–	–	–	–
11	–	–	–	–
12	–	–	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	–	–	–
2	$v_1(2) = 0$	–	–	–
3	0	–	–	–
4	0	–	–	–
5	(+8)=8	–	–	–
6	8	–	–	–
7	8	–	–	–
8	8	–	–	–
9	8	–	–	–
10	8	–	–	–
11	8	–	–	–
12	8	–	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	$v_2(1) = 0$	–	–
2	$v_1(2) = 0$	–	–	–
3	0	–	–	–
4	0	–	–	–
5	(+8)=8	–	–	–
6	8	–	–	–
7	8	–	–	–
8	8	–	–	–
9	8	–	–	–
10	8	–	–	–
11	8	–	–	–
12	8	–	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	$v_2(1) = 0$	–	–
2	$v_1(2) = 0$	$v_2(2) = 6$	–	–
3	0	–	–	–
4	0	–	–	–
5	(+8)=8	–	–	–
6	8	–	–	–
7	8	–	–	–
8	8	–	–	–
9	8	–	–	–
10	8	–	–	–
11	8	–	–	–
12	8	–	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	$v_2(1) = 0$	–	–
2	$v_1(2) = 0$	$v_2(2) = 6$	–	–
3	0	6	–	–
4	0	6	–	–
5	(+8)=8	8	–	–
6	8	–	–	–
7	8	–	–	–
8	8	–	–	–
9	8	–	–	–
10	8	–	–	–
11	8	–	–	–
12	8	–	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	$v_2(1) = 0$	–	–
2	$v_1(2) = 0$	$v_2(2) = 6$	–	–
3	0	6	–	–
4	0	6	–	–
5	(+8)=8	8	–	–
6	8	8	–	–
7	8	14	–	–
8	8	–	–	–
9	8	–	–	–
10	8	–	–	–
11	8	–	–	–
12	8	–	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	$v_2(1) = 0$	–	–
2	$v_1(2) = 0$	$v_2(2) = 6$	–	–
3	0	6	–	–
4	0	6	–	–
5	(+8)=8	8	–	–
6	8	8	–	–
7	8	14	–	–
8	8	14	–	–
9	8	14	–	–
10	8	14	–	–
11	8	14	–	–
12	8	14	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	$v_2(1) = 0$	0	–
2	$v_1(2) = 0$	$v_2(2) = 6$	–	–
3	0	6	–	–
4	0	6	–	–
5	(+8)=8	8	–	–
6	8	8	–	–
7	8	14	–	–
8	8	14	–	–
9	8	14	–	–
10	8	14	–	–
11	8	14	–	–
12	8	14	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	$v_2(1) = 0$	0	–
2	$v_1(2) = 0$	$v_2(2) = 6$	6	–
3	0	6	–	–
4	0	6	–	–
5	(+8)=8	8	–	–
6	8	8	–	–
7	8	14	–	–
8	8	14	–	–
9	8	14	–	–
10	8	14	–	–
11	8	14	–	–
12	8	14	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	$v_2(1) = 0$	0	–
2	$v_1(2) = 0$	$v_2(2) = 6$	6	–
3	0	6	6	–
4	0	6	6	–
5	(+8)=8	8	8	–
6	8	8	–	–
7	8	14	–	–
8	8	14	–	–
9	8	14	–	–
10	8	14	–	–
11	8	14	–	–
12	8	14	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	$v_2(1) = 0$	0	–
2	$v_1(2) = 0$	$v_2(2) = 6$	6	–
3	0	6	6	–
4	0	6	6	–
5	(+8)=8	8	8	–
6	8	8	8	–
7	8	14	14	–
8	8	14	–	–
9	8	14	–	–
10	8	14	–	–
11	8	14	–	–
12	8	14	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	Stvar 1	Stvar 2	Stvar 3	Stvar 4
1	$v_1(1) = 0$	$v_2(1) = 0$	0	–
2	$v_1(2) = 0$	$v_2(2) = 6$	6	–
3	0	6	6	–
4	0	6	6	–
5	(+8)=8	8	8	–
6	8	8	8	–
7	8	14	14	–
8	8	14	14	–
9	8	14	18	–
10	8	14	–	–
11	8	14	–	–
12	8	14	–	–

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

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

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

	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

Stvari

		1	2	3	4
Vrijednost	v	8	6	12	5
Volumen („cijena”)	c	5	2	7	3

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

Primjer: problem naprtnjače (*Knapsack problem*)

- 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
  - 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

Primjer: problem naprtnjače (*Knapsack problem*)

	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

0⇒STOP

6-value(Stvar2)=6-6=0

18-value(Stvar3)=18-12=6

23-value(Stvar4)=23-5=18

Primjer: problem naprtnjače (*Knapsack problem*)

- 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)

Primjer: problem naprtnjače (*Knapsack problem*)

- 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 f$alse$), 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 $[c_{max} - cost(n), n - 1]$

- ako zadnja stvar nije ušla u izbor, sljedeće polje bit će prvo polje s lijeva, dakle indeksa $[c_{max}, n - 1]$

Pseudo-kod rješenja Knapsack problema dinamičkim programiranjem

```
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
    }
```

Složenost:
$O(Cmax \cdot N)$.

Primjer programskog kôda za rješavanje Knapsack problema

- 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);
```

Primjer programskog kôda za rješavanje Knapsack problema

- 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 (*backtracing?*). 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.

Pseudokod rekurzivnog rješenja Knapsack problema

```
KnapsackRec (c,k):
if ( c > 0 && k > 0 )                          //Ako je to element za razmatranje …
{  kNo = KnapsackRec(c,k-1);
   if ( c >= cost[k])                          //Ako je preostali maksimum cijene dovoljan …
     kYes= KnapsackRec(c-cost[k],k-1) + value[k];
   else
     kYes = kNo;
   if (kYes > kNo )
   {   w[c,k] = kYes;                          // Upis ostvarive vrijednosti u tablicu W(x,k)
       decisions[c,k] = true;                  // k-ta ulazi u najbolji izbor
     return kYes; }
   else
   { w[c,k] = kNo;                             // Upis ostvarive vrijednosti u tablicu W(c,k) .
     decisions[c,k] = false;                   //k-ta ne ulazi u najbolji izbor
     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	-	-	-
3	…6	-	-	-
5	…6	-	-	-
7	…6	-	-	-
8	…6	-	-	-
9	…6	-	-	-
10	…6	-	-	-
12	…6	-	-	-

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

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	-	-
3	…6	6	-	-
5	…6	11	-	-
7	…6	…11	-	-
8	…6	…11	-	-
9	…6	…11	-	-
10	…6	…11	-	-
12	…6	…11	-	-

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

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	-
3	…6	6	6	-
5	…6	11	11	-
7	…6	…11	14	-
8	…6	…11	14	-
9	…6	…11	14	-
10	…6	…11	19	-
12	…6	…11	…19	-

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

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

$v_0(\cdot) = 0$

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

Knapsack problem – dodatna literatura i vizualizacija

- Više o problemu naprtnjače: https://rosettacode.org/wiki/Knapsack_problem

- Razni zadaci:
  - https://www.spoj.com/problems/KNAPSACK/
  - http://codeforces.com/problemset/problem/632/E

- Vizualizacije:
  - https://www.cs.usfca.edu/~galles/visualization/DPFib.html
  - https://www.cs.usfca.edu/~galles/visualization/DPChange.html
  - https://www.cs.usfca.edu/~galles/visualization/DPLCS.html

Knapsack problem – programski resursi

- Knapsack Problem in Python With 3 Unique Ways to Solve, https://www.pythonpool.com/knapsack-problem-python/
- Knapsack Problem | Dynamic Programming, https://www.codesdope.com/course/algorithms-knapsack-problem/
- 0-1 Knapsack Problem using Dynamic Programming, https://pencilprogrammer.com/algorithms/0-1-knapsack-problem-dynamic-programming/

Primjeri za rješavanje

Primjer

- Dinamičko programiranje
- Zadana je kvadratna matrica koja se sastoji od prirodnih brojeva
- Pijun se nalazi u gornjem lijevom kutu te se može kretati jedno polje dolje ili jedno polje dijagonalno dolje-desno
- Cilj je doći do donjeg retka tako da je suma brojeva na putu **maksimalna**

5	X	X	X
2	4	X	X
7	9	2	X
7	7	6	7

Rješenje (1)

5	X	X	X
2	4	X	X
7	9	2	X
7	7	6	7

- Rekurzivno rješenje dinamičkim programiranjem:
- Osnovna ideja: na slici desno, da li put označen plavom bojom ikada bude dio optimalnog rješenja? Zašto?
- Definirajmo funkciju cijena(r, s) kao vrijednost najboljeg puta od gornjeg lijevog ruba do pozicije (r, s).
- cijena(r, s) = max{ cijena(r-1, s), cijena(r-1, s-1) } + A[r][s]
- Rješenje je: max{ cijena(n-1, i) za 0 <= i < n }

Rješenje (2)

- Rekurzivno rješenje:

```
int mem[MXN][MXN]; // inicijalizirano na -1

int cijena(int r, int s) {
   if (r == 0) return A[r][s];
   if (mem[r][s] != -1) return mem[r][s];
   int best = cijena(r - 1, s);
   if (s > 0) best = max(best, cijena(r - 1, s - 1));
   return mem[r][s] = best + A[r][s];
}

int sol = 0; // krajnje rješenje
for (int i = 0; i < n; i++)
   sol = max(sol, cijena(n - 1, i));
```

Rješenje (3)

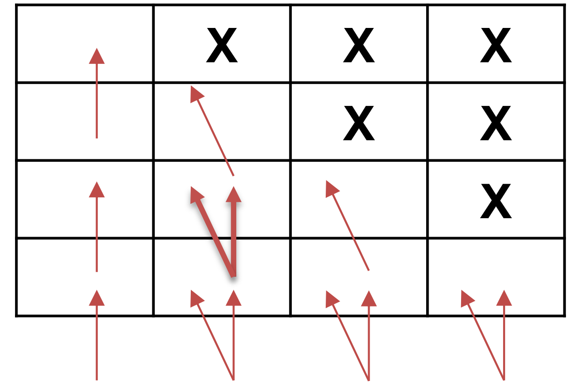
- Iterativno rješenje tablicom:
- Kojim redoslijedom ćemo izračunavati vrijednosti funkcije cijena(r, s), tj. kojim redoslijedom rekurzija obilazi stanja?
- Prije nego što pokušamo izračunati cijena(r, s) trebamo biti sigurni da su izračunate vrijednosti za cijena(r-1, s) i cijena(r-1, s-1) (ako postoje).
- Nemamo više funkciju cijena, već samo matricu dp[r][s] koja ima isto značenje
- Zapravo direktno popunjavamo memoizacijsku matricu

Rješenje (4)

- Međusobne ovisnosti (preduvjete) potrebno je zapisati u tabličnom formatu.

- Možemo prvo s lijeva na desno izračunati vrijednosti matrice dp u prvom retku, zatim opet slijeva nadesno u drugom retku, i tako dalje dok ne dođemo do n-tog retka.

- Svi preduvjeti se nalaze u prethodnom retku pa je jasno da su svi izračunati.



Rješenje (5)

- Iterativno rješenje:

```
dp[0][0] = A[0][0];
for (int r = 1; r < n; r++) {
   dp[r][0] = dp[r-1][0] + A[r][0];
   for (int c = 1; c <= r; c++) {
      dp[r][c] = max(dp[r-1][c], dp[r-1][c-1]);
      dp[r][c] += A[r][c];
   }
}

int best = 0;
for (int i = 0; i < n; i++)
   best = max(best, dp[n-1][i]);
```

Primjer

- Dinamičko programiranje

- Svaka riječ se može rastaviti na palindrome (banana – b anana, abbabbaab – abba bb aa b, ...). Na koliko se najmanje dijelova mora podijeliti zadana riječ, a da je svaki dio palindrom? Neka je A oznaka za zadanu riječ.

- Neka je P(l, r) najmanji broj dijelova na koji se može podijeliti zadana podriječ koja počinje znakom A[l] i završava s A[r].

Rješenje (1)

- Koje situacije postoje?

- Ako je A palindrom (što se lako provjeri) onda je rezultat 1, a ako nije onda riječ možemo podijeliti na dva dijela i za ta dva dijela izračunati optimalan rastav i sumirati ih. Znači da ovaj problem znamo podijeliti na 2 manja problema, ali istog tipa. Npr. Riječ ‘banana’ možemo rastaviti na ‘ban’ i ‘ana’ i onda izračunati optimalan rastav za ‘ban’ i ‘ana’. Još samo treba provjeriti koji od rastava riječi na dvije je najbolji, jer je b i anana bolji od ban i ana.

- $P(l, r) = 1$ ako je podriječ $A[l \ldots r]$ palindrom
- $P(l, r) = \min\{P(l, k) + P(k+1, r); l \leq k < r\}$

Rješenje (2)

- Iterativno rješenje:

```
int mem[MXN][MXN]; // inicijalizirano na -1

int func (int l, int r) {
   if (palindrom(l, r)) return 1;
   if (mem[l][r] != -1) return mem[l][r];
   int ret = r - l + 1;
   for (int i = l; i < r; i++) {
      ret = min(ret, f(l, i) + f(i + 1, r));
   }
   return mem[l][r] = ret;
}
```

Primjer

- Knapsack
- Ivan ima ruksak u koji stane maksimalno N kilograma prije nego pukne. Na raspolaganju ima M predmeta od kojih svaki ima svoju težinu T_i i vrijednost V_i. **Svakog predmeta ima beskonačno mnogo kopija.**
- Ruksak je prazan te u njega treba smjestiti predmete tako da zbroj vrijednosti svih predmeta bude maksimalan.
- Primjer :
  - 6 2 (u ruksak stane 6 kila, a na raspolaganju imamo 2 predmeta)
  - 3 4 (predmet težak 3 kila, vrijednosti 4) predmet #1
  - 5 7 (predmet težak 5 kila, vrijednosti 7) predmet #2
- Potrebno je ispisati najveći zbroj vrijednosti u ruksaku
- Rješenje: 8 (više se isplati uzeti dva predmeta #1, nego jedan #2)

Rješenje (1)

- Intuitivno će se mnogi sjetiti pohlepnog (*greedy*) rješenja:
    - stavi najviše vrijedan predmet u ruksak koji stane
    - ponovi (sa prostorom ruksaka umanjenim za stavljeni predmet)
- …ili varijaciju tog rješenja koja računa omjer cijene i težine

- Važno je primijetiti da takva rješenja nisu točna, kao što se vidi već iz priloženog jednostavnog test primjera.

- Za rješavanje ovog zadatka treba razmišljati drugačije, problem se treba svesti na jednostavniji oblik

Rješenje (2)

- Zamislimo da imamo ruksak u koji stane X kilograma.
- Na raspolaganju imamo tri predmeta Y1 (9kg, 10kn), Y2 (12kg, 13kn) i Y3 (16kg, 18kn)

- Definirajmo funkciju f koja nam za $f(a)$ vraća najveću vrijednost koju možemo spremiti u ruksak veličine a.
- Ako stavimo u ruksak prvi predmet, znači da će vrijednost u ruksaku biti 10 + idealno rješenje za ruksak u koji stane "X-9" kila.
  - Zapisano preko funkcije: $f(x) = 10 + f(x-9)$
- Isto možemo napisati i za predmete 2 i 3.
  - $f(x) = 13 + f(x-12)$, $f(x) = 18 + f(x-16)$

- Pod pretpostavkom da znamo točno izračunati $f(x-9)$, $f(x-12)$ i $f(x-16)$ koja od ove tri formule izračuna pravu vrijednost $f(x)$?

- Naravno najveća, jer tražimo maksimalni zbroj vrijednosti u ruksaku

Rješenje (3)

- Cijelo rješenje se temelji na tome da znamo izračunati rješenje za neki $f(x)$, tako da ga zapišemo kao

- $f(x-a) + b$, gdje je a težina predmeta koji dodamo u ruksak, a b vrijednost tog predmeta. Smisao je u tome da se parametar u $f(x)$ stalno smanjuje do broja na kojem je rješenje očito. Koliko je rješenje za $f(0)$?

- $f(0) = 0$, jer svaki predmet ima težinu

- Koliko je rješenje za $f(1)$, a za $f(2)$? Računa se preko iste jednadžbe.

- Jedino moramo paziti da ne pokušamo staviti predmet koji je teži od preostalog mjesta u ruksaku.

Rješenje (4)

Iterativno rješenje:

```
// Prva petlja računa f(x), za brojeve od 1 do n
for ( int i=1; i<=n; ++i ) {
   // Druga petlja prolazi kroz sve predmete i pokušava ih ubaciti u ruksak

   for ( int j=0; j<m; ++j ){
      // Provjeravamo stane li predmet "j" u ruksak veličine "i"
      if ( t[j] <= i ){
         f[i] = max ( f[i], v[ j ] + f[ i - t[ j ] ] );
      }
   }
}
printf ("%d\n", f[ n ] );
```

Rješenje (5)

Iterativno rješenje:

```
int f[ 10000 ];
int t[ 10000 ],v[ 10000 ];

int main () {
   int n,m;
   scanf ("%d %d", &n, &m);
   for ( int i=0; i<m; ++i ) {
      scanf ("%d%d",&t[i],&v[i]);
   }

   // Prva petlja racuna f(x), za brojeve od 1 do n
   for ( int i=1; i<=n; ++i ) {
      // Druga petlja prolazi kroz sve predmete i pokusava ih ubaciti u ruksak
      for ( int j=0; j<m; ++j ) {
         // Provjeravamo stane li predmet "j" u ruksak velicine "i"
         if ( t[j] <= i ) {
            f[i] = max ( f[i], v[ j ] + f[ i - t[ j ] ] );
         }
      }
   }
   printf ("%d\n", f[ n ] );
   return 0;
}
```

Rješenje (6)

Rekurzivno rješenje:

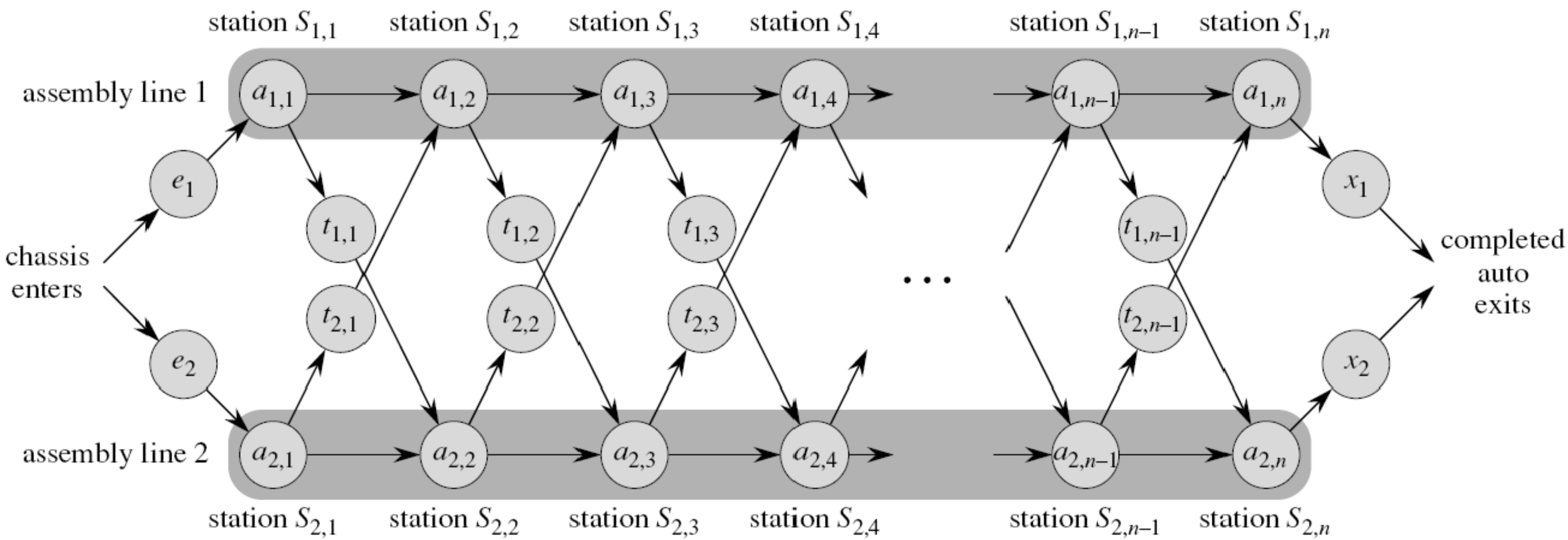
```
int n, m;
int memo[ 10000 ];
bool bio[ 10000 ];
int t[ 10000 ], v[ 10000 ];

int func (int x) {
   if (x == 0) return 0;
   if (bio[ x ] == 1) return memo[ x ];
   for (int i=0; i<m; ++i) {
      if (t[i] <= x) memo[ x ] = max(memo[ x ], v[i] + f( x - t[i] ));
   }
   bio[ x ] = 1;
   return memo[ x ];
}

int main () {
   scanf ("%d %d", &n, &m);
   for (int i=0; i<m; ++i) {
      scanf ("%d %d", &t[i], &v[i]);
   }
   printf ("%d\n", func(n));
   return 0;
}
```

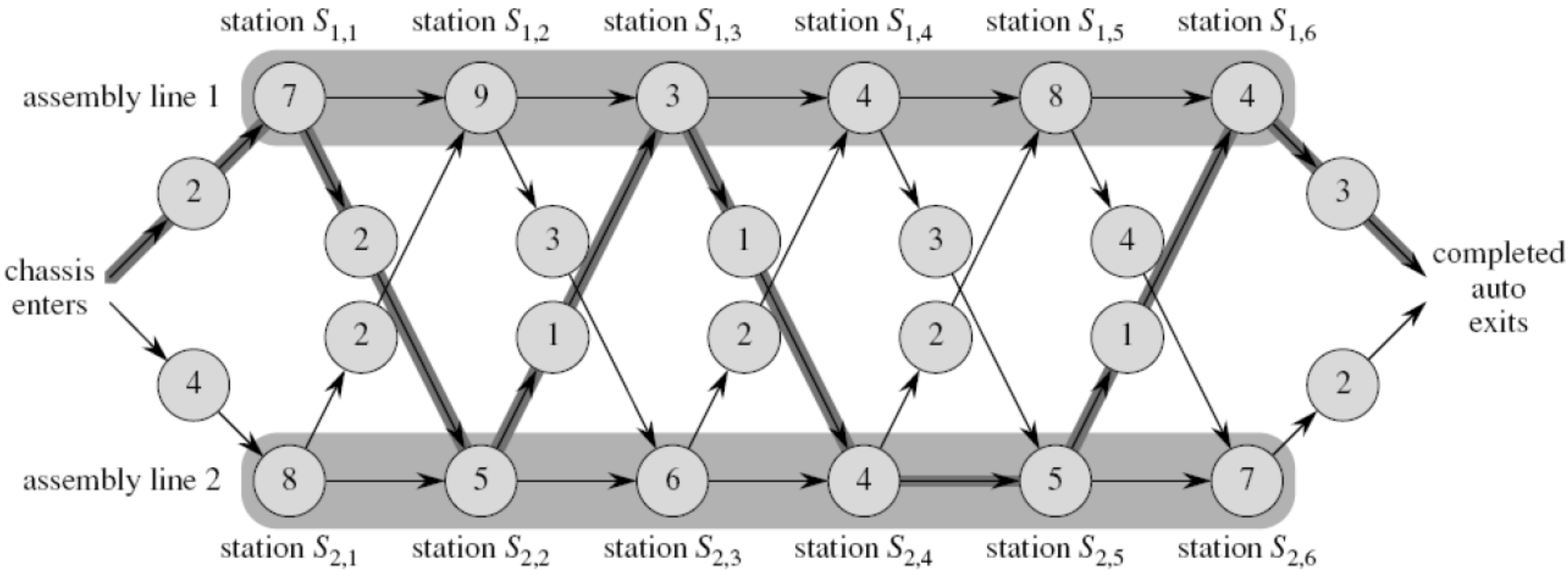
Primjer (1)

- Organiziranje proizvodne linije u tvornici automobila:



- Problem pronalaženja najbržeg puta u proizvodnji korištenjem dinamičkog programiranja

Primjer (2)



(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)