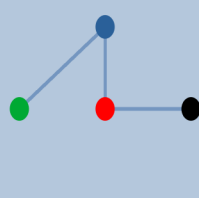
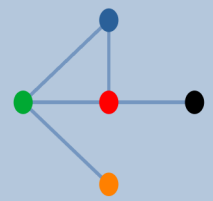
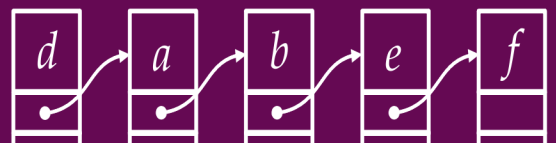
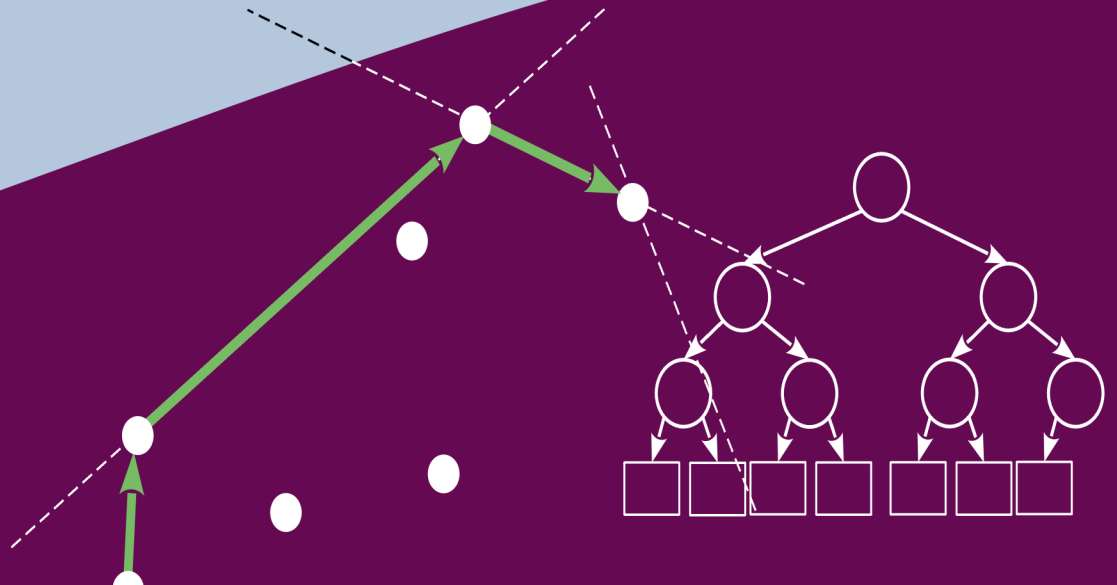
Advanced algorithms and data structures

Week 6: Dynamic programming



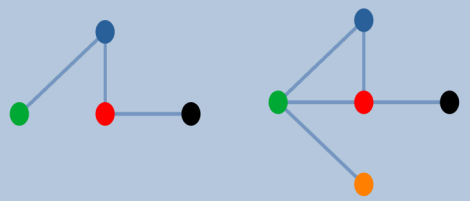


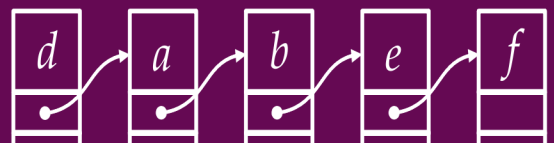


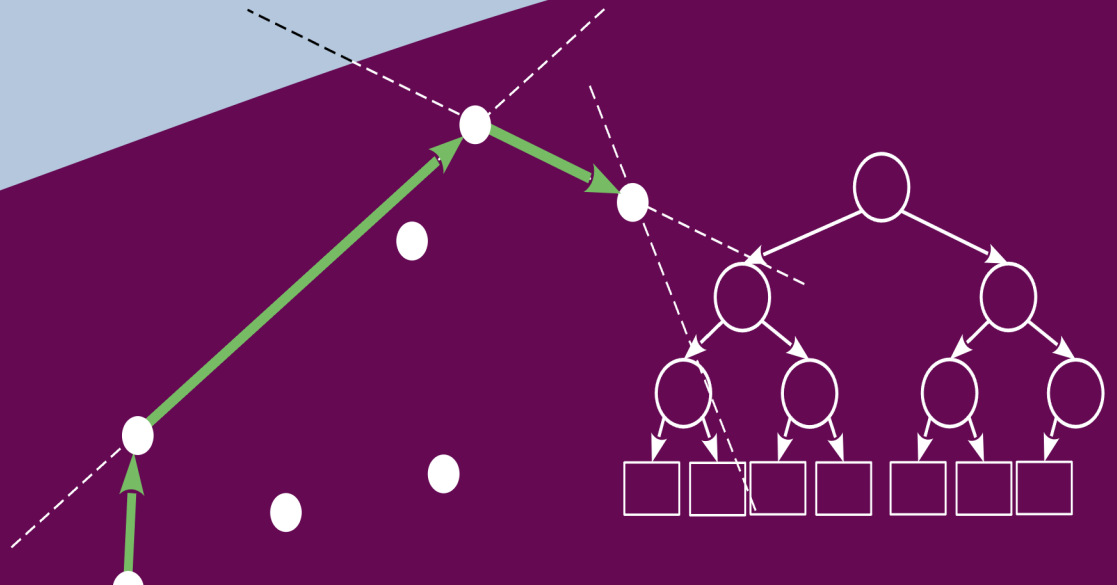


Napredni algoritmi i strukture podataka

Tjedan 6: Dinamičko programiranje







Creative Commons

you are free to:

to share — to reproduce, distribute and communicate the work to the public, to

adapt the work

under the following conditions:

Attribution: You must acknowledge and attribute the authorship of the work in a way specified by the author or licensor (but not in a way that suggests that you or your use of their work has their direct endorsement).

non-commercial: You may not use this work for commercial purposes.

share under the same conditions: if you modify, transform, or create using this work, you may distribute the adaptation only under a license that is the same or similar to this one.

In the case of further use or distribution, you must make clear to others the license terms of this work. Any of the above conditions may be waived with the permission of the copyright holder.
Nothing in this license infringes or limits the author's moral rights.

The text of the license is taken from http://creativecommons.org/

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/*

- **Dynamic programming**(*Dynamic Programming*)
  - A method (strategy) whose basic principle is to gradually build a solution of a complex problem using solutions of similar less complex problems (*bottom-up*access)
  - Applicable when subproblems "overlap" (*overlapping subproblems*)
  - Unlike divide and rule (*divide and conquer*) strategies that problem solves*top-down*approach, it does not repeat the work already done because it makes the most of the results of the previous steps

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

Introduction (2)

- "Programming" in the context of dynamic programming does not mean programming in the usual sense, it is just a name for **strategy** (planned procedure)

- In implementation, the table method is the most common; *memoization*

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*

Introduction (3)

- A typical application is in optimization problems in which the final solution is reached only after a series of decisions, where after each decision the problem remains the same, only less complex, and the final solution is obtained on the basis of the optimal solutions of the subproblems that arise after each individual decision and whose the best possible solutions considering the state achieved until then (Bellman's principle of optimality;Bellman's Principle of Optimality)

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)

Introduction (4)

- Due to the step-by-step construction of the final solution using solutions under ... subproblems, dynamic programming is only applicable to recursive problems

- Very similar to greedy (*greedy*) strategy
  - This problem-solving strategy will be explained in detail in the following lectures...

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…

Introduction (5)

- In conclusion: dynamic programming solves problems by combining solving subproblems.
  - Unlike the "divide and conquer" approach, dynamic programming is applied to problems whose subproblems are not mutually independent, but have some common sub.. subproblems.
  - In such cases, "divide and conquer" algorithms would needlessly solve the same subproblems multiple times.
  - Typically, algorithms developed on the principles of dynamic programming solve each subproblem only once and store its solution in a table (in temporary memory; implementation on computers).
  - Resolving the same subproblem multiple times is avoided

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

Development and application in practice (1)

- Dynamic programming (DP) was developed with the purpose of optimizing large and complex technological systems for the production of various industrial products or components and energy, which can be divided into subsystems ("divide and conquer").

- During dynamic programming, their interactions are taken into account during the gradual optimization of subsystems.

- The choice made in each step (stage) contributes to the optimum of the system - the combination of the optimum of the subsystems.

Razvoj i primjena u praksi (1)

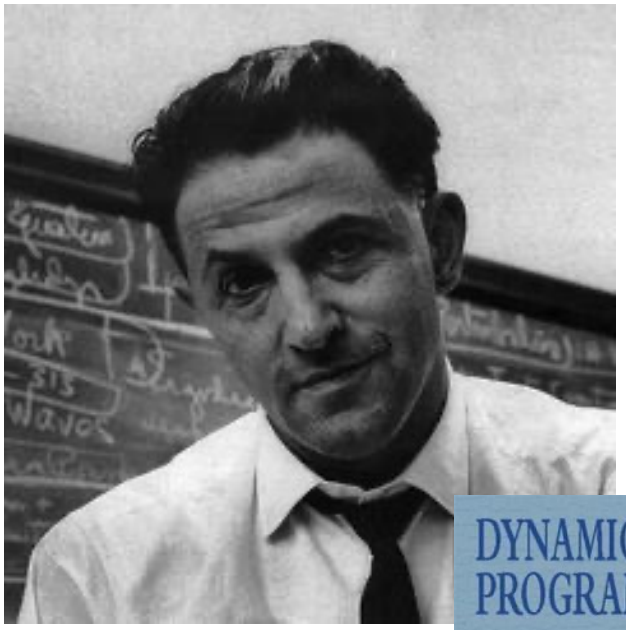
- Dinamičko programiranje (DP) je razvijeno sa svrhom optimalizacije velikih i složenih tehnoloških sustava za proizvodnju raznih industrijskih proizvoda ili komponenti te energetiku, a koji se mogu podijeliti na podsustave („podjeli pa vladaj”).

- 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.

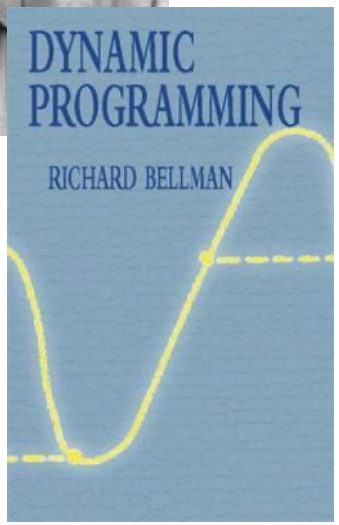
Development and application in practice (2)

- Dynamic programming was the first to be treated in detail **Richard E Bellman** in 1957



- Since then, dynamic programming has been used in mathematics, science, engineering, biomathematics, medicine, economics, informatics and artificial intelligence.

- The application of dynamic programming is expanding with the development of ANN methods and procedures, deep data analysis, knowledge discovery (*data mining*), *soft computing* and other areas of artificial intelligence.

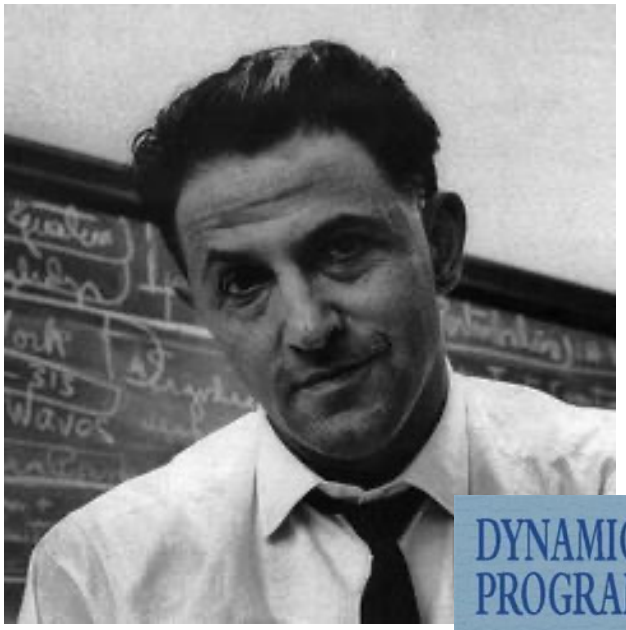


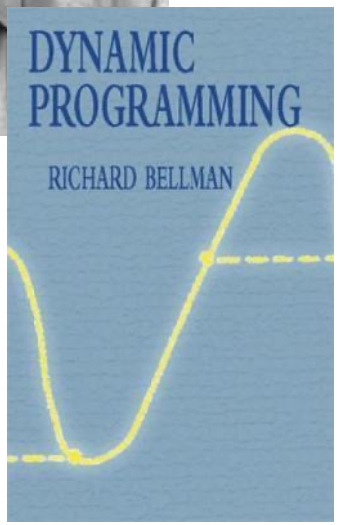
Razvoj i primjena u praksi (2)

- 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.





Development and application in practice (3)

- The following four formal procedures of dynamic programming are often applied in practice to solve various problems:
  1. one-dimensional distribution
  2. two-dimensional distribution
  3. the shortest path
  4. dynamics of equipment replacement

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

Features of the problem (1)

- In order to be solvable according to the principle of dynamic programming, the problem must satisfy the followingtwo conditions :

  - Optimal substructure (*optimal substructure*)
    - property of the problem that it contains the optimal solutionoptimal solutions independentsubproblems (consists of)

      - this in itself is not enough because otherwise it is also a property that points to the application of "greedy" (*greedy*) strategies

      - a good example is the problem of searching for the shortest path (Dijkstra's algorithm, lakoma strategy); the shortest path between two vertices consists of the shortest path from the starting vertex to an intermediate vertex and the shortest path from the intermediate vertex to the final vertex⇔ optimal solutions of two independent subproblems

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

Features of the problem (2)

- Overlap of subproblems (*overlapping subproblems*)
  - the property of the problem that its solution requires (leads to) repeated solving of identical sub ... subproblems, so previous results can be used to solve later steps faster (example: Fibonacci numbers)
  - all sub ... sub ... subproblems must still be independent

Optimal substructure

Overlapping subproblems

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

Troubleshooting (1)

! Solving problems using dynamic programming involves four basic steps:

1. Observe the structure of the optimal solution

   And) Does the problem satisfy the optimal substructure condition?

   b) Are subproblems overlapped?

   - this is the step in which we evaluate the solvability of the problem by dynamic programming
   - it depends entirely on intuition

2. Set a recursive formula for calculating the value of the final solution

   - "value" is the quantity (function) that is optimized, i.e. whose minimum or maximum is sought

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

Troubleshooting (2)

3. Calculate the optimal value of the final solution using solutions of smaller subproblems (use *bottom-up* approach i *memoization* procedure)

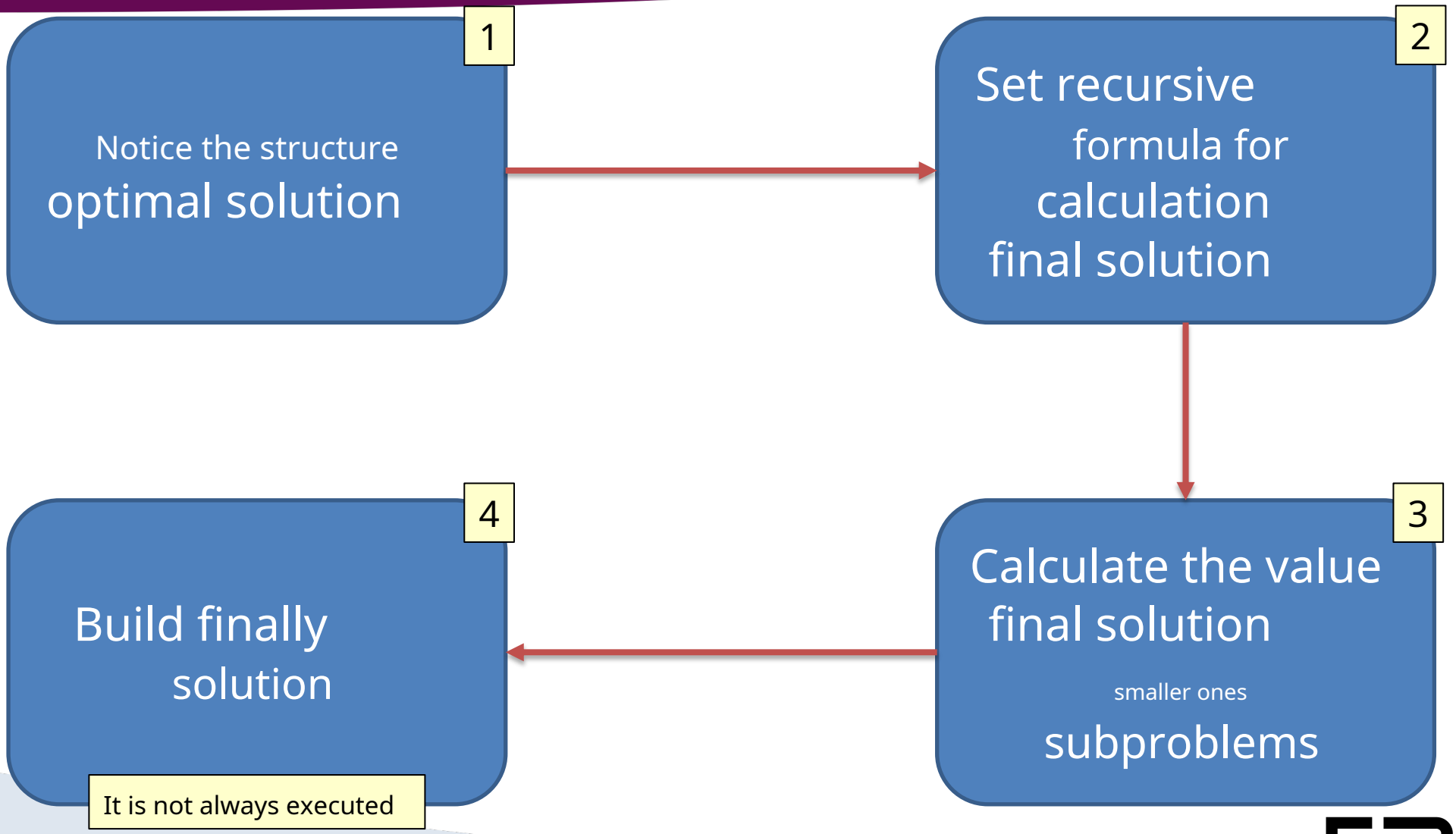
4. Build (construct) the final solution (determine the optimal set of decisions)
   - by this step, the problem has already been solved (we know the optimal value), so this step is not always performed, because in order to achieve it, it is usually necessary to store additional information during the previous three steps

Rješavanje problema (2)

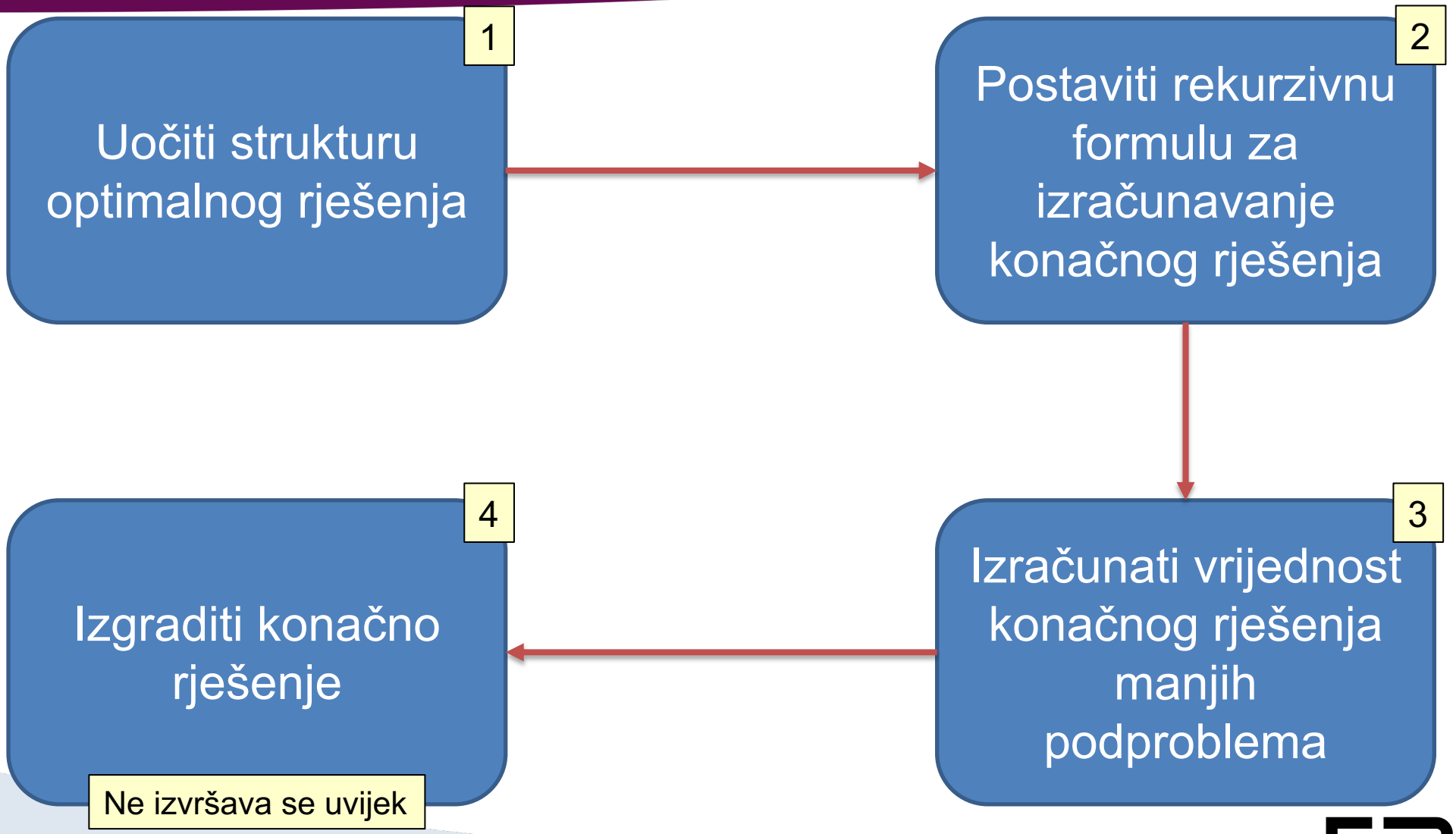
3. Izračunati optimalnu vrijednost konačnog rješenja koristeći rješenja manjih podproblema (korištenje *bottom-up* pristup i *memoization* postupka)

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

Troubleshooting (3)



Rješavanje problema (3)



Memoization (1)

- Example Fibonacci sequence:

  ! != "= 1; #= #$!+ #$", > 2

- Naive implementation:

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

Function ( )for calculating the nth Fibonacci number

- To calculate the member #functions ( )for < are called a greater number of times, so for > 5

  valid for:

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

Memoization (1)

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

- Naivna implementacija:

```
1 if n ≤ 2 then
2     return 1;
3 end
4 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)

- By using the memoization procedure, it is not necessary to go into recursion and count the members of the array that are already calculated (and known).
  - At the beginning, we initialize all elements of the array to -1 (we mark them as uncalculated)

```
1 if fib[n] ≠ −1 then
2     return fib[n];
3 end
4 if n ≤ 2 then
5     fib[n] = 1;
6 end
7 fib[n] = Fib(n − 1) + Fib(n − 2);
8 return fib[n];
```

Function ( )to calculate the nth Fibonacci number using the memoization procedure

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)

```
1 if fib[n] ≠ −1 then
2     return fib[n];
3 end
4 if n ≤ 2 then
5     fib[n] = 1;
6 end
7 fib[n] = Fib(n − 1) + Fib(n − 2);
8 return fib[n];
```

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

Example 1

- Determine the factorial of the number 3 (the factorial of the number n is a mathematical function that successively calculates the product of the numbers 1...n), using function recursion.

- Solution:

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

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

- It should be noted that in each step the result of the previous step is used - function recursion(*arrows*). The general mathematical description of the solution would be:

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

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

$$j_{max} = 3$$

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:

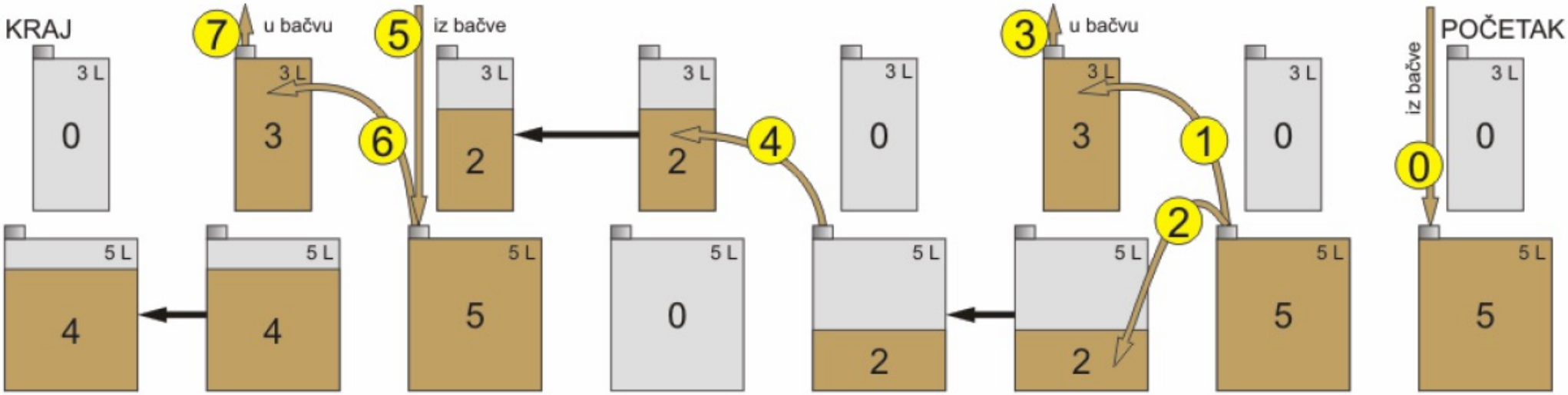
1. korak: $f_{1,1} = 1$ (*temeljni slučaj*)
2. korak: $f_{2,2} = 2 \circ f_{1,1}$ (*pravilo rekurzije*)

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

$$j_{max} = 3$$

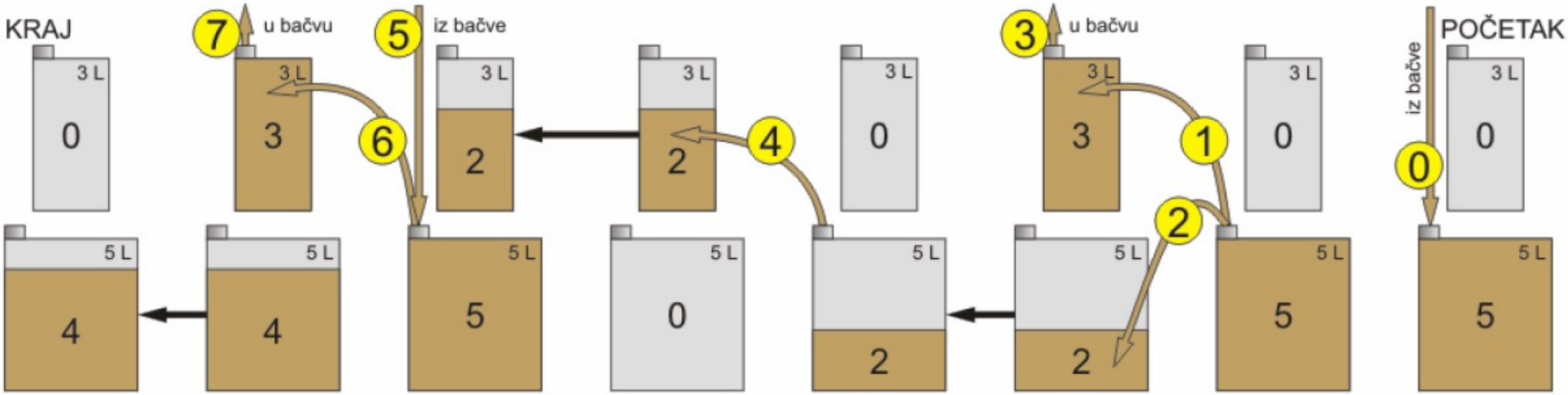
Example 2 (1)

- Special synthetic lubricating oil, packed in 50 L (liter) barrels, is issued from the factory's warehouse by the liter. The maintenance worker comes to the warehouse with a 5 L bucket to take 4 L of oil, and the warehouse worker has a 3 L bucket. We can measure the volume of oil with a completely full or empty bucket.



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.



Example 2 (2)

- Solution:

- The problem is solved in step m – 1 (solving starts from the end of the problem) by pouring 1 L of oil from a "big" bucket with 5 L of oil into a "small" bucket with 2 L of oil. In the last step (m), 3 L of oil is returned from the small bucket to the barrel and a large 5 L bucket with 4 L of oil (remaining after pouring) is taken.

- Backward analysis leads to the first step (i = 1), in which the large bucket is filled with 5 L of oil, then the small bucket is filled with 3 L, leaving 2 L in the large bucket. From the small bucket, the oil returns to the barrel....

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 ….

Backpack problem

(*Knapsack problem*)

Problem naprtnjače

(*Knapsack problem*)

0-1 knapsack problem (*Knapsack problem*)

- Let us have a container of some finite capacity and a set of elements of different weights and values. It is necessary to select a subset of items so that their total value is maximum and the total weight is less than or equal to the capacity of the container.

- What subset is that?
  - What are the items that fit in the backpack (which selection is the best possible, i.e. optimal)?

- 0-1 or *integer* version of the 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

Types of Knapsack problems

different (separate) combinatorial optimization problems

- 0-1 or *integer* Knapsack problem
  - in the literature it is also referred to as "0.1 knapsack" or "0-1 knapsack"
  - the items available in the problem can only be selected or not selected, and a certain item can only be selected once
- version 0-1 knapsack with repetition
  - the restricted Knapsack problem (*bounded knapsack problem*, BKP)
    - each item has a limited number of identical copies
      - by increasing the number of copies of the object, the problem becomes more
  - difficult, the unbounded Knapsack problem (*unbounded knapsack problem*, UKP)
    - the number of copies of each item is unlimited
      - due to the unlimited number of copies of the case, UKP is even more complex and difficult to solve
- *fractional* version
  - is resolved *greedy* algorithm
- the multidimensional Knapsack problem (*multidimensional Knapsack problem*, d-KP)
- multiple Knapsack problem (*multiple knapsack problem*, the FEM quadratic Knapsack
- problem (*quadratic knapsack problem*, QKP)

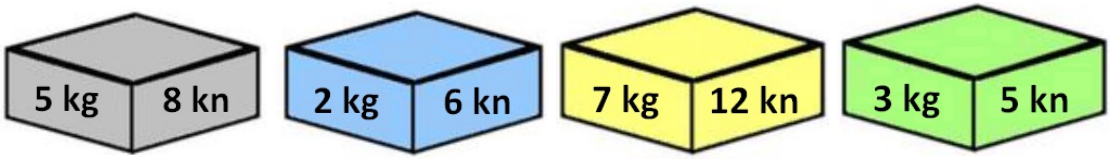
Vrste Knapsack problema

različiti (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)

Example: knapsack problem (*Knapsack problem*)

- A thief has only one bag of capacity (or volume) =12 kg, which does not fit all the items available to him. What is the maximum total value he can steal if everything he takes has to fit in a sack?
  - What are the subjects (which selection is the best possible, i.e. optimal)?

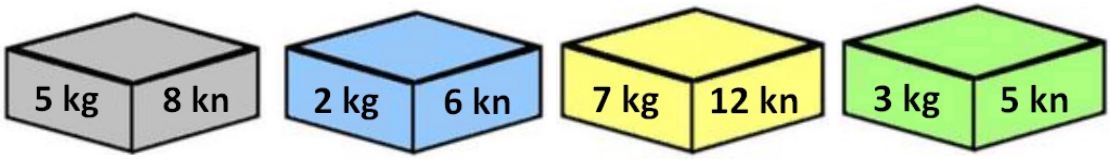


	Subject 1	Subject 2	Subject 3	Subject 4
Value	8	6	12	5
Volume ("price")	5	2	7	3

- So the problem is to optimize the selection of stolen items; we are looking for the maximum value for the given volume of the bag, where the "price" of each object (decision) is the space it occupies.

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)?



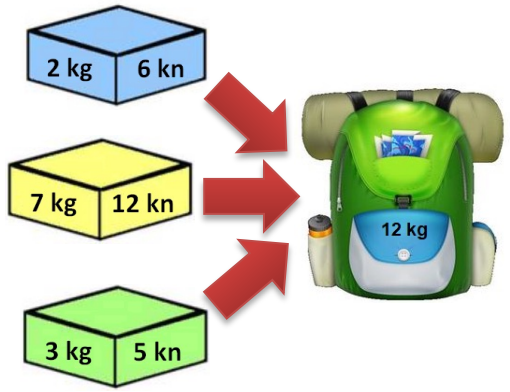
	Predmet 1	Predmet 2	Predmet 3	Predmet 4
Vrijednost	8	6	12	5
Volumen („cijena”)	5	2	7	3

- 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.

Example: knapsack problem (*Knapsack problem*)

- For such a small number of subjects, the problem is simple enough that we can solve it by heart and easily find:
  - The greatest value a thief can steal %&'= 23
  - The total price (volume) of the items that make up the best selection is correct = 12
    (therefore, in this case, the thief can completely use the bag)
  - Best choice: **other**, **the third** and **fourth** case

	Subject 1	Subject 2	Subject 3	Subject 4
Value	8	**6**	**12**	**5**
Volume ("price")	5	2	7	3

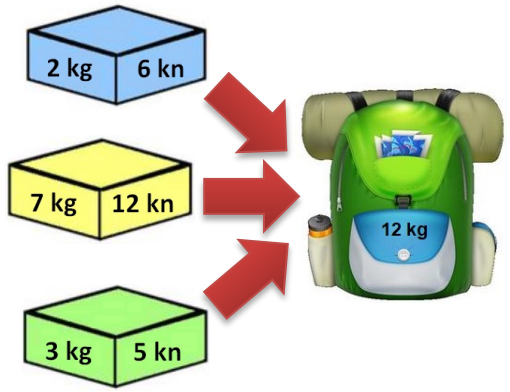


- For a large number of subjects, intuitive solving becomes impossible and we need an algorithm that will surely lead us to the best possible solution!

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!

Example: knapsack problem (*Knapsack problem*)

- But it's not just about finding a solution - it's about stealing the most

  yes, the thief would like to escape before the police arrive!

  - Two of his "colleagues" had already attempted the same theft before him, but only one later found out that he didn't steal as much as he could, and the other got "stuck". It was like this...

- The first thief was greedy - he took only the most valuable things in turn

  subjects.

  - So he took the first and third items, total value = 20and

    left because it could no longer fit in his bag. Only later did he find out that he could have done a better "job"...

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”…

Example: knapsack problem (*Knapsack problem*)

- The other "in his free time" deals with programming, so he knew the algorithm that would surely find the best solution - he tried all combinations of items and calculated their total value and volume. However, there were as many combinations as there were subsets in the set of subject, therefore2'. The calculation took (too) long and he was so busy that he forgot that he was in someone else's house, and the police were on their way...

- Taught by the experience of his predecessors, this thief worked out a fairly fast algorithm in advance - dynamic programming.

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.

Example: knapsack problem (*Knapsack problem*)

- Observe the structure of the optimal (final) solution:
  - Let's say we know the best possible solution when we observe k subjects i the entire available capacity.
  - Let us denote the set of items that make up the best choice with W
  - It is crucial to note that if from a set W remove only one (any) subject, the remaining subjects surely make the best possible choice from the set of -1 items, but for the capacity of the bag minus the volume of the separated item.

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.

Example: knapsack problem (*Knapsack problem*)

- Proof: contradiction. Let's say that after extracting one item

  (let's mark it with)from the set Ω the remaining items form the set and are not the best possible choice for remaining capacity. This means that from the set of − 1 of subjects can choose a set some other items that will have a higher value in total than those from and will not occupy more than the remaining capacity of the bag. But then the set { , }overall gives a higher value than Ω when considering all subjects and the entire available capacity, which is contrary to the assumption that Ω is the best possible solution.■

! Conclusion: the best possible solution to a larger problem consists of the best possible solutions to smaller problems of the same typeÞ**optimal substructure**. Solution for subject is obtained using solutions for − 1subject, that for − 1items from the solution for − 2subjects etc.Þ **subproblems overlap**.

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**.

Example: knapsack problem (*Knapsack problem*)

- Set the recursive formula for calculating the final solution, i.e.

  optimal values of the objective function

  - By observing any object, for example -of that, available set ,

    we observe that the final solution can only be twofold: it either includes or does not include the observed object

  - If it does not include it, then the solution to the problem is for the given price and whole

    expensive equal to the optimal solution for the price and set without -of that subject \{ }.Symbolically, ( ) =)(( ),where)(( )indicates the highest possible value for the price when we observe the set without -of that subject \{ }.

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 $S\backslash\{k\}$.

Example: knapsack problem (*Knapsack problem*)

continuation

- If it includes it, then it is the highest achievable value for the price and set with - by that subject is equal to the optimal solution for the set without -of that item and the highest allowed price minus the price -of that item, i.e. for the price − ( ),increased by value -of that subject. ( ) Symbolically, ((=)) [− + ( ).

- From the previous considerations it follows that*k*-the subject is included in the best choice if it is

[()* −(+ ( )]) > ()*( ).

! The required recursive formula reads:

(=(m)ax{ ()* , ()* −( ) + [( ) } ; +A(=)0]. ( )

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 $[v_{k-1}(c - cost(k)) + value(k)] > v_{k-1}(c)$.

- Tražena rekurzivna formula glasi:

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

Example: knapsack problem (*Knapsack problem*)

- Calculate the value of the final solution using smaller solutions subproblem (*bottom-up*access +*memoization*)
  - Take into consideration one subject at a time and make decisions by applying it recursive formulas
  - The algorithm is accelerated by storing previous solutions in a table (filled by columns, i.e. objects or things):
    - Rows of the table = prices (occupied volumes)
    - Table columns = objects or things

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∈ 0	-	-	-
2	-	-	-	-
3	-	-	-	-
4	-	-	-	-
5	-	-	-	-
6	-	-	-	-
7	-	-	-	-
8	-	-	-	-
9	-	-	-	-
10	-	-	-	-
11	-	-	-	-
12	-	-	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∈ 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∈ 0	-	-	-
2	!2 ∈ 0	-	-	-
3	-	-	-	-
4	-	-	-	-
5	-	-	-	-
6	-	-	-	-
7	-	-	-	-
8	-	-	-	-
9	-	-	-	-
10	-	-	-	-
11	-	-	-	-
12	-	-	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" = 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	-	-	-
2	!2 ∉ 0	-	-	-
3	0	-	-	-
4	0	-	-	-
5	(+8)=8	-	-	-
6	-	-	-	-
7	-	-	-	-
8	-	-	-	-
9	-	-	-	-
10	-	-	-	-
11	-	-	-	-
12	-	-	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	-	-	-
2	!2 ∉ 0	-	-	-
3	0	-	-	-
4	0	-	-	-
5	(+8)=8	-	-	-
6	8	-	-	-
7	8	-	-	-
8	8	-	-	-
9	8	-	-	-
10	8	-	-	-
11	8	-	-	-
12	8	-	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	"1 ∉ 0	-	-
2	!2 ∉ 0	-	-	-
3	0	-	-	-
4	0	-	-	-
5	(+8)=8	-	-	-
6	8	-	-	-
7	8	-	-	-
8	8	-	-	-
9	8	-	-	-
10	8	-	-	-
11	8	-	-	-
12	8	-	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	"1 ∉ 0	-	-
2	!2 ∉ 0	"2 ∉ 6	-	-
3	0	-	-	-
4	0	-	-	-
5	(+8)=8	-	-	-
6	8	-	-	-
7	8	-	-	-
8	8	-	-	-
9	8	-	-	-
10	8	-	-	-
11	8	-	-	-
12	8	-	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" ∉ max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	"1 ∉ 0	-	-
2	!2 ∉ 0	"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	-	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	"1 ∉ 0	-	-
2	!2 ∉ 0	"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	-	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∈ 0	"1 ∈ 0	-	-
2	!2 ∈ 0	"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	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∈ 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	"1 ∉ 0	0	-
2	!2 ∉ 0	"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	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	"1 ∉ 0	0	-
2	!2 ∉ 0	"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	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" ∉ max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	"1 ∉ 0	0	-
2	!2 ∉ 0	"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	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	"1 ∉ 0	0	-
2	!2 ∉ 0	"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	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	"1 ∉ 0	0	-
2	!2 ∉ 0	"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	-	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	"1 ∉ 0	0	-
2	!2 ∉ 0	"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	-

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" = max{ "#$, ( )

"#$ − [+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2	Thing 3	Thing 4
1	!1 ∉ 0	"1 ∉ 0	0	0
2	!2 ∉ 0	"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

Things

		1	2	3	4
Value	c	8	6	12	5
Volume ("price")	c	5	2	7	3

!" ∉ 0

" ∉ max{ "#$, ( )

"#$ −[+ (( ))]

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

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

Example: knapsack problem (*Knapsack problem*)

- Build (construct) the final solution (determine the optimal set decision)
  - The best selection of things can easily be read from the table; line with the default price (bag capacity) is viewed from right to left; the highest achievable value is in the last column
  - If it is reached by adding the last thing, then it is different from the value in the field to the left of i the last thing goes into the best selection; the next field is the one from which the red arrow comes
  - If the last item is not in the best selection, the value in the field on the left is equal to that in the observed field; move to the field on the left (blue arrow) and repeat the consideration

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

Example: knapsack problem (*Knapsack problem*)

	Thing 1	Thing 2		
1	!1 ∉ 0	"1 ∉ 0		
2	!2 ∉ 0	"2 ∉ 6	6	6
3	0	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	
9	8	14	18	18
10	8	14	18	19
11	8	14	18	19
12	8	14	20	23

0⇒STOP

6-value(Thing2)=6-6=0

18-value(Thing3)=18-12=6

23-value(Item4)=23-5=18

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

Example: knapsack problem (*Knapsack problem*)

- Programmatic determination of optimal selection in more complex problems
  requires maintaining a list with the current selection for each table field, so another size table 456×

- In this example, it is enough to just enter the marks whether it is in a step the currently observed thing entered the selection or not (in other words, arrow type labels; e.g. = for the oblique arrow and = for horizontal)

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)

Example: knapsack problem (*Knapsack problem*)

- after the completion of the algorithm, it starts from the last field of the table, the index [456,],and looks to see if in that step the last thing entered the best choice (label)or not (label f),and in both cases, move to the field from which you came while filling out the table and repeat the consideration

- if the last thing entered the selection, the next field will be the index field [456- ( ), − 1]

- if the last item was not selected, the next field will be the first field from the left, so the index [456, − 1]

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-code solution of the Knapsack problem by dynamic programming

```
Knapsack (items, Cmax, value[], cost[]):
    form table w[Cmax, items] and table decisions[Cmax, items];
    initialization: w[0, *] = 0 and w[*, 0] = 0;          //zero row and column
    initialisation: decisions[*,*] = false;
    for (k = 1; k<=items; ++k)                            //For all things ... //
       for (c = 1; c<=Cmax; ++c)                          For all prices ...
      {
          kNo = w[c,k-1];                                 //Value without kit, for the same price.
          if (c >= cost[k])                               //If the price is allowed (remaining
              kYes = w[c-cost[k],k-1] + value[k];         capacity)... //Value with k.
              otherwise                                   //If the kth is already too expensive by itself, //the
              kYes = kNo;                                 highest value with the kth = the one without the kth.
          if (kYes > kNo)
              { w[c,k] = kYes;                            //k-ta enters the best choice
                decisions[c,k] = true; }
                else
                 w[c,k] = kNo;                            //k-ta is not included in the best choice
     }
```

Complexity:
$O(Cmax \cdot N)$.

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 ·N).

Example program code for solving the Knapsack problem

- Printing (backwards) selected elements (C# syntax)

```
Print (bool[,] decisions):
int c=maxcost, k = items;
// 'maxcost' and 'items' must be visible to this
function or // passed as input arguments
To
{ 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

- 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);
```

Example program code for solving the Knapsack problem

! A direct application of the recursive formula would be an example of a "classical" divide-and-conquer strategy, and such a solution would be significantly slower than a tabular one, since calculating the best selections for bags with a larger capacity requires processing the same subproblems again (*backtracing?*). For example, both capacities 3 and 4, each by itself, require the processing of capacities 1 and 2, which means that the smaller capacities would be solved more than once, so the program would be extremely "redundant".

!Disadvantages of ordinary recursion in this and other solvable problems

dynamic programming (meaning tabular) are a direct consequence of overlapping subproblems that need to be solved to find the final solution.

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.

Pseudocode of the recursive solution to the Knapsack problem

```
KnapsackRec (c,k):
if ( c > 0 && k > 0 )                                   //If it's an element to consider...
{ kNo = KnapsackRec(c,k-1);
   if ( c >= cost[k])                                   //If the remaining maximum price is enough...
      kYes= KnapsackRec(c-cost[k],k-1) + value[k];
      otherwise
      kYes = kNo;
   if (kYes > kNo )
   { w[c,k] = kYes;                                     // Entry of the achievable value in the table W(x,k) //
       decisions[c,k] = true;                           k-th enters the best choice
      return kYes; }
   otherwise
   { w[c,k] = kNo;                                      // Write the achievable value in the table W(c,k) . //k-ta is
     decisions[c,k] = false;                            not included in the best choice
     return kNo;}
} else
    return 0;
```

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;
```

Appendix: Shortening the procedure

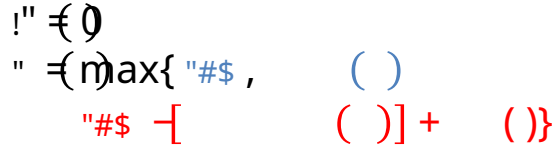
- Human-friendly, relatively awkward to program

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

sort by c →

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

	Thing 2	Thing 4	Thing 1	Thing 3
2	6	-	-	-
3	…6	-	-	-
5	…6	-	-	-
7	…6	-	-	-
8	…6	-	-	-
9	…6	-	-	-
10	…6	-	-	-
12	…6	-	-	-



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	-	-	-

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

Appendix: Shortening the procedure

- Human-friendly, relatively awkward to program

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

sort by c

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

	Thing 2	Thing 4	Thing 1	Thing 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	-	-

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	-	-

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

Appendix: Shortening the procedure

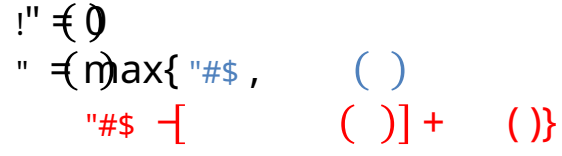
- Human-friendly, relatively awkward to program

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

sort by c →

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

	Thing 2	Thing 4	Thing 1	Thing 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	-



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	-

$v_0(\cdot) = 0$

$v_k(c) = \max\{v_{k-1}(c),$

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

Appendix: Shortening the procedure

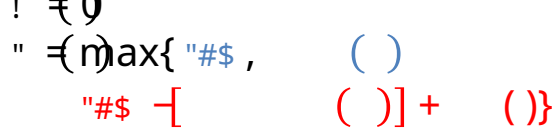
- Human-friendly, relatively awkward to program

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

sort by c

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

	Thing 2	Thing 4	Thing 1	Thing 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



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 - additional literature and visualization

- More on the knapsack problem:https://rosettacode.org/wiki/Knapsack_problem

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

- Visualizations:
  - 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 – 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

The Knapsack Problem - Program Resources

- 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-dynamicprogramming/

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/

Examples to solve

Primjeri za rješavanje

Example

- Dynamic programming
- Given is a square matrix consisting of natural numbers
- The pawn is located in the upper left corner and can move one square down or one square diagonally down to the right
- The goal is to get to the bottom row so that the sum of the numbers is on the way**maximum**

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

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

Solution (1)

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

- Recursive solution by dynamic programming:
- Basic idea: in the picture on the right, is the path marked in blue ever part of the optimal solution? Why?
- Let's define the function cost(r, s) as the value of the best path from the upper left edge to position (r, s).
- price(r, s) = max{ price(r-1, s), price(r-1, s-1) } + A[r][s]
- The solution is: max{ price(n-1, i) for 0 <= i < n }

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 }

Solution (2)

- Recursive solution:

```
int mem[MXN][MXN];// initialized to -1

intprice(int r,int s) {
      if(r==0)returnAND[r][with];
      if(mem[r][with] != -1)returnmem[r][with]; int
      best=price(r-1,with);
      if(with>0)the best=max(the best,price(r-1,with-1)); returnmem[
      r][with] =the best+AND[r][with];
}

int salt=0;// final solution for(int i=0;and
<n;and++)
      salt=max(salt,price(n-1,and));
```

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));
```

Solution (3)

- Iterative solution with a table:
- In what order will we calculate the values of the price function (r, s), i.e. in what order does the recursion go around the states?
- Before we try to calculate price(r, s) we need to make sure that the values for price(r-1, s) and price(r-1, s-1) have been calculated (if they exist).
- We no longer have the price function, but only the matrix dp[r][s] which has the same meaning
- In fact, we directly fill in the memoization matrix

Rješenje (3)

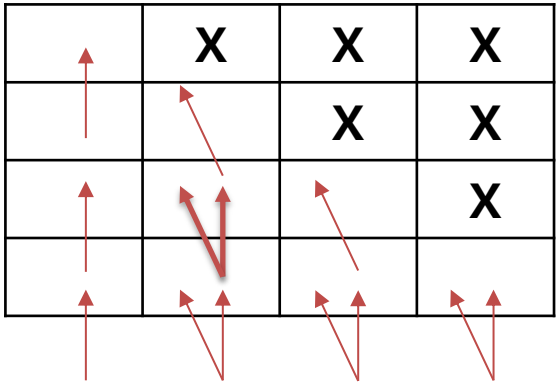
- Iterativno rješenje tablicom:
- Kojim redoslijedom ćemo izračunavati vrijednosti funkcije cijena(r, s), tj. kojim redoslijedom rekurzija obilazi stanja?
- Prije nego što pokušamo izračunati cijena(r, s) trebamo biti sigurni da su izračunate vrijednosti za cijena(r-1, s) i cijena(r-1, s-1) (ako postoje).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

Solution (4)

- Interdependencies (preconditions) should be written in tabular format.

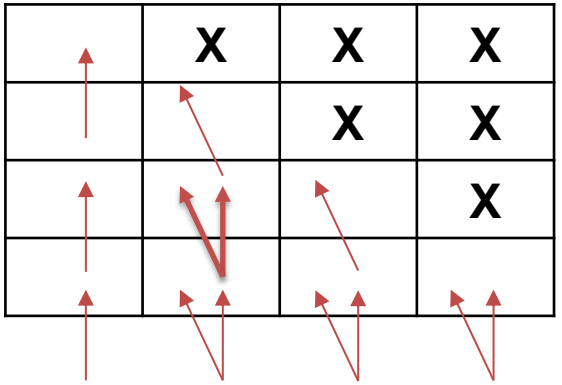
- We can first calculate the values of the matrix dp from left to right in the first row, then again from left to right in the second row, and so on until we reach the nth row.

- All prerequisites are in the previous line, so it is clear that they are all calculated.



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.



Solution (5)

- Iterative solution:

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

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

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]);
```

Example

- Dynamic programming

- Each word can be broken down into palindromes (banana – b anana, abbabbaab – abba bb aa b, ...). How many parts must the given word be divided into at least, if each part is a palindrome? Let A be the label for the given word.

- Let P(l, r) be the smallest number of parts into which the given subword starting with A[l] and ending with A[r] can be divided.

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].

Solution (1)

- What situations are there?

- If A is a palindrome (which is easy to check), then the result is 1, and if it is not, then we can divide the word into two parts and calculate the optimal separation for those two parts and sum them up. It means that we know how to divide this problem into 2 smaller problems, but of the same type. For example We can split the word 'banana' into 'ban' and 'ana' and then calculate the optimal split for 'ban' and 'ana'. You just need to check which of the two word splittings is the best, because bi anana is better than ban and ana.

- P(l, r) = 1 if the subword A[l...r] is a palindrome
- $P(l, r) = \min\{P(l, k) + P(k+1, r); l \leq k < r\}$

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

Solution (2)

- Iterative solution:

```
int mem[MXN][MXN];// initialized to -1

intfunc(int l,int r) {
      if(palindrome(l,r))return1; if(mem[l][r] != -1)
      returnmem[l][r]; int ret=r-l+1; for(int i=l;and<r;
      and++) {

            ret=min(ret,f(l,and) +f(and+1,r));
      }
      returnmem[l][r] =ret;
}
```

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;
}
```

Example

- Knapsack

- Ivan has a backpack that fits the maximum kilograms before it breaks. There are M items available, each of which has its own weight (and value (.**There are infinitely many copies of every object.**

- The backpack is empty and you need to place items in it so that the sum of the value of all items is maximum.

- Example:
  - 6 2 (the rucksack holds 6 kilos, and we have 2 items at our disposal)
  - 3 4 (object weighing 3 kilos, value 4) object #1
  - 5 7 (object weighing 5 kilos, value 7) object #2
- It is necessary to print the highest sum of values in the backpack

- Solution: 8 (it is more worthwhile to take two subjects #1 than one #2)

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)

Solution (1)

- Intuitively, many will remember greedy (*greedy*) solutions:
  - put the most valuable item in a backpack that fits
  - repeat (with the space of the backpack reduced by the placed object)
- ...or a variation of that solution that calculates the ratio of price and weight

- It is important to note that such solutions are not correct, as can already be seen from the attached simple test example.

- To solve this task, you need to think differently, the problem needs to be reduced to a simpler form

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

Solution (2)

- Let's imagine that we have a backpack that holds X kilograms.
- We have three items available: Y1 (9kg, HRK 10), Y2 (12kg, HRK 13) and Y3 (16kg, HRK 18).

- Let's define a function which for us ( )returns the most value we can store in a backpack size .
- If we put the first item in the backpack, it means that the value in the backpack will be 10 + the ideal solution for a backpack that fits "X-9" kilos.
  - Written via function: ( ) = 10 + (− 9)
- We can write the same for subjects 2 and 3.
  - ( ) = 13 + (− 12) , ( ) = 18 + (− 16)

- Assuming we know how to calculate correctly (− 9), (− 12)and (− 16)which of these three formulas calculates the true value ( )?

- Of course the largest, because we are looking for the maximum sum of values in the backpack

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

Solution (3)

- The whole solution is based on knowing how to calculate the solution for some ( ),so let's write it as
- (−) + ,where is the weight of the items we add to the backpack a the value of that item. The point is that the parameter u ( )constantly reduces to a number where the solution is obvious. How much is the solution for (0)?
- (0) = 0,because every object has weight
- How much is the solution for (1),and for (2)?It is calculated using the same equation.
- We only have to be careful not to try to put an object that is heavier than the remaining space in the backpack.

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.

Solution (4)

Iterative solution:

```
// The first loop calculates f(x), for numbers from 1 to n for(
int i=1;and<=n; ++and) {
    // The second loop goes through all the items and tries to put them in the backpack

    for(int j=0;j<m; ++j){
        // Check if object "j" fits in backpack of size "i" if(t[j] <=and){

            f[and] =max(f[and],c[j] +f[and-t[j] ] );
        }
    }
}
printf("%d\n",f[n] );
```

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 ] );
```

Solution (5)

Iterative solution:

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

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

    // First loop of calculation f(x), for numbers from 1 to n for(
    int i=1;and<=n; ++and) {
        // The second loop goes through all the items and tries to put them in the backpack for(
        int j=0;j<m; ++j) {
            // We check if object "j" fits in backpack of size "i" if(t[j] <=and) {

                f[and] =max(f[and],c[j] +f[and-t[j] ] );
            }
        }
    }
    printf("%d\n",f[n] ); return0;

}
```

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;
}
```

Solution (6)

Recursive solution:

```
int n,m;
int memo[10000];
bool was[10000];
int t[10000],c[10000];

intfunc(int x) {
    if(x==0)return0;
    if(was[x] ==1)returnmemo[x]; for(int i=0;and
    <m; ++and) {
        if(t[and] <=x)memo[x] =max(memo[x],c[and] +f(x-t[and] ));
    }
    was[x] =1;
    returnmemo[x];
}

intmain() {
    scanf("%d %d", &n, &m); for(int
    i=0;and<m; ++and) {
        scanf("%d %d", &t[and], &c[and]);
    }
    printf("%d\n",func(n)); return0;

}
```

Rješenje (6)

Rekurzivno rješenje:

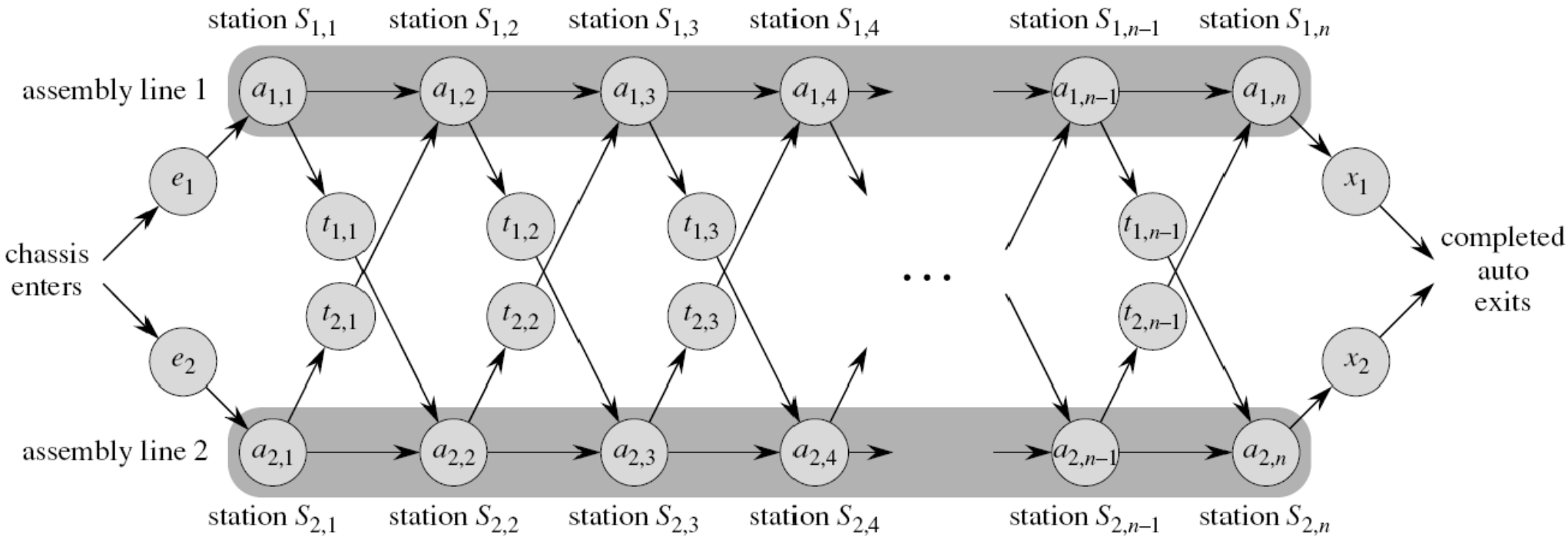
```
int n, m;
int memo[ 10000 ];
bool bio[ 10000 ];
int t[ 10000 ], v[ 10000 ];

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;
}
```

Example (1)

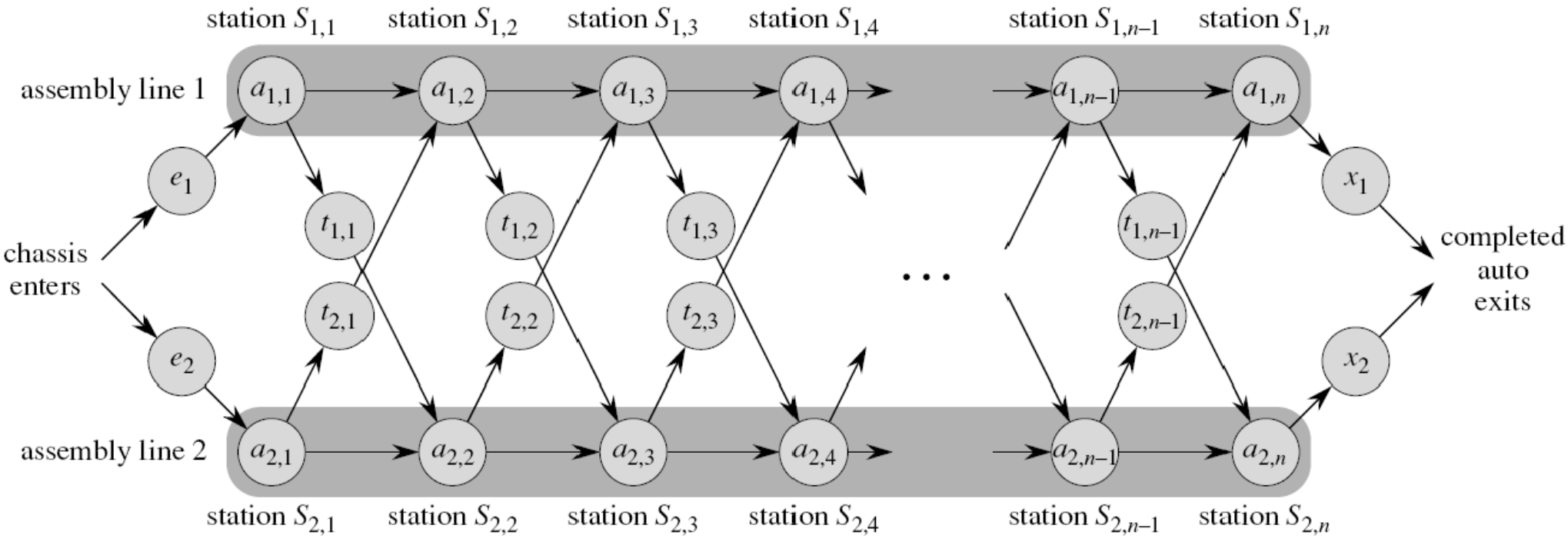
- Organizing a production line in a car factory:



- The problem of finding the fastest path in production using dynamic programming

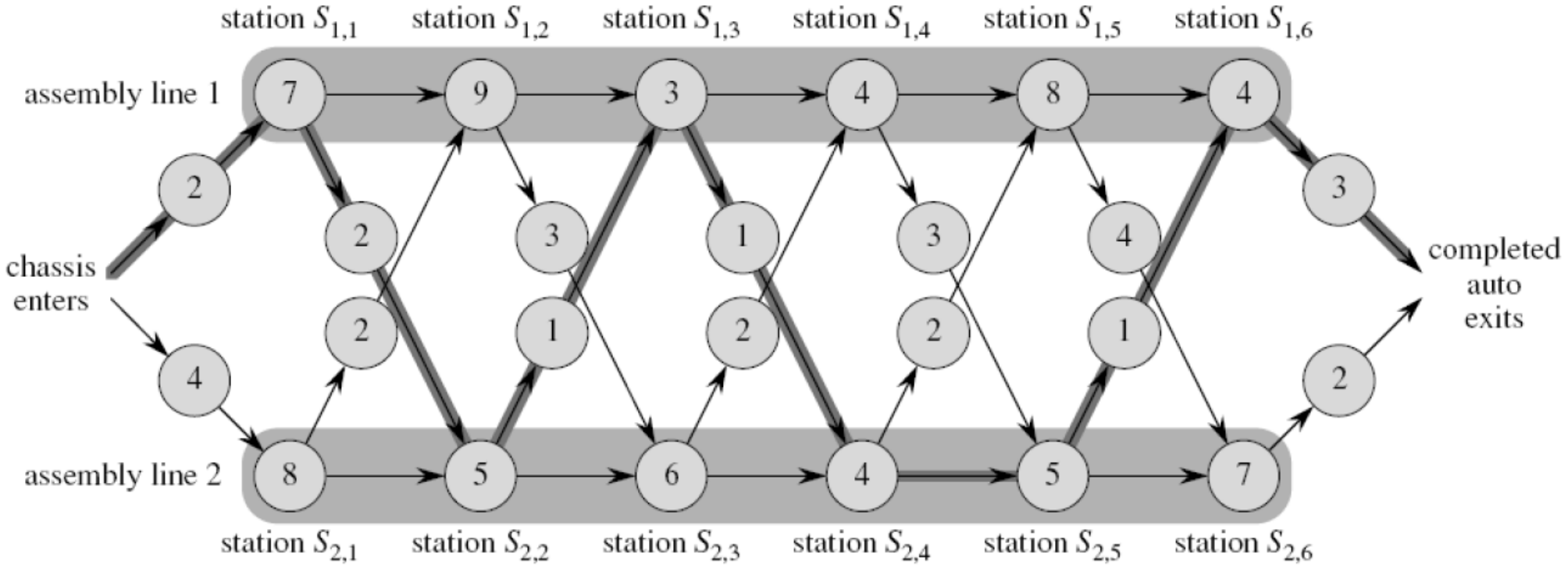
Primjer (1)

- Organiziranje proizvodne linije u tvornici automobila:



- Problem pronalaženja najbržeg puta u proizvodnji korištenjem dinamičkog programiranja

Example (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

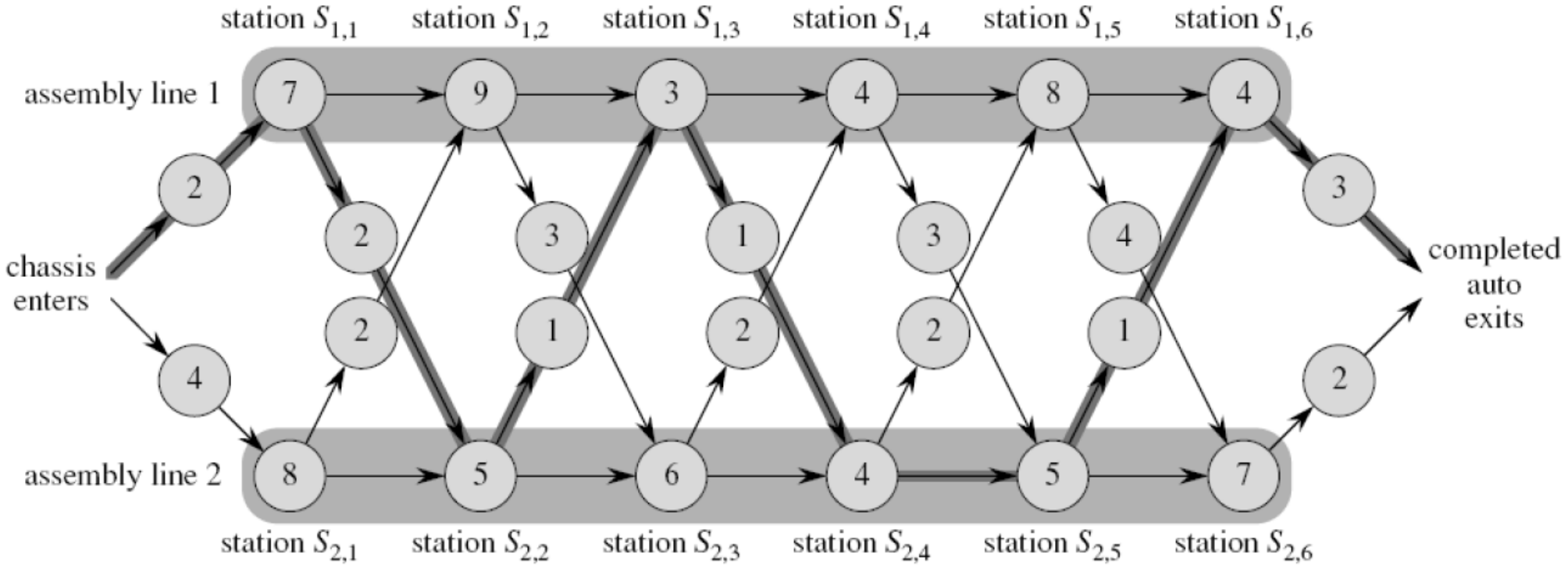
$f^* = 38$

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)

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)