Algorithms & Complexity
Lecture 5: Dynamic Programming

October 15, 2019

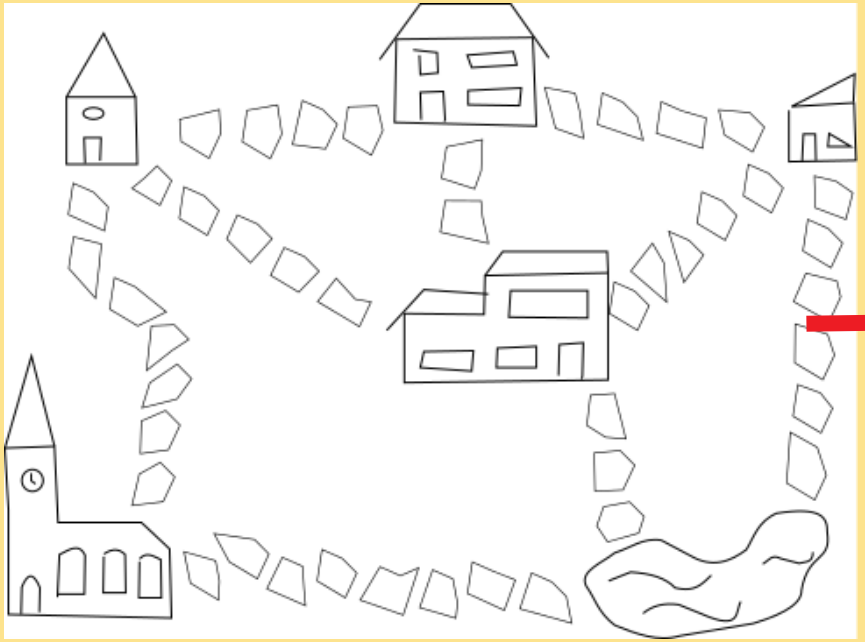
CentraleSupélec / ESSEC Business School

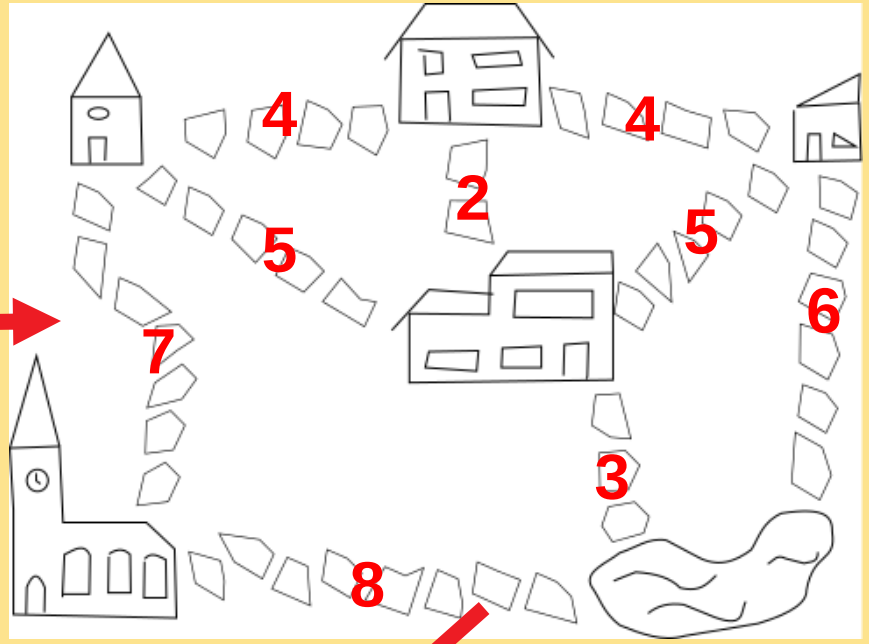
Dimo Brockhoff

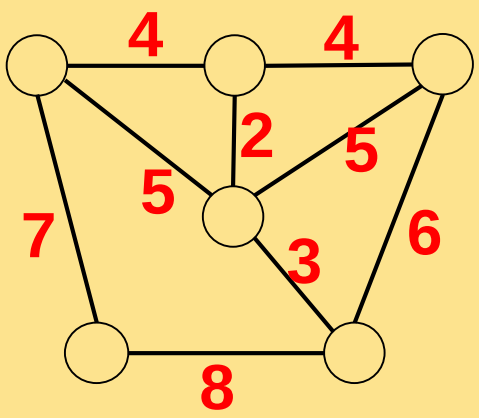
Inria Saclay – Ile-de-France

Discussion Home Exercise

Exercise 1: Little Slopy Village

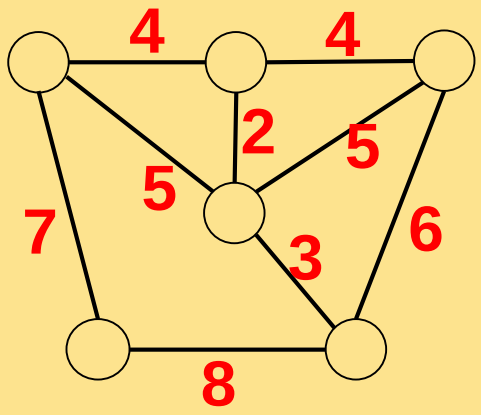






Discussion Home Exercise

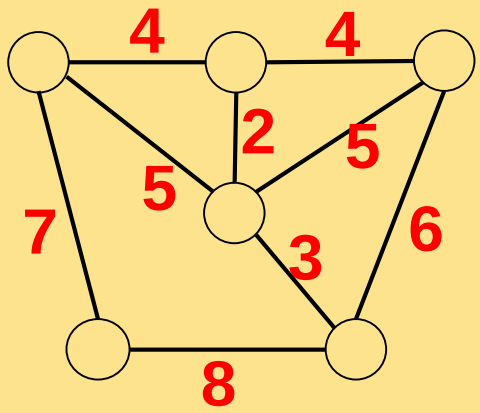
Exercise 1: Little Slopy Village

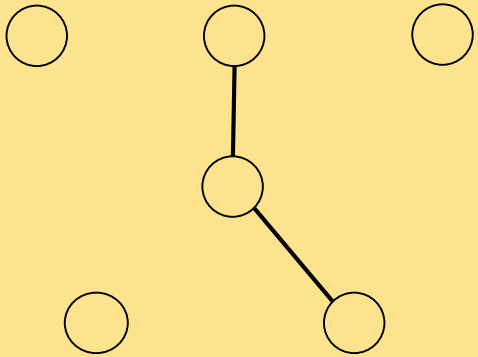


Algorithm of Kruskal:
take always the shortest edge that does not produce a cycle

Discussion Home Exercise

Exercise 1: Little Slopy Village



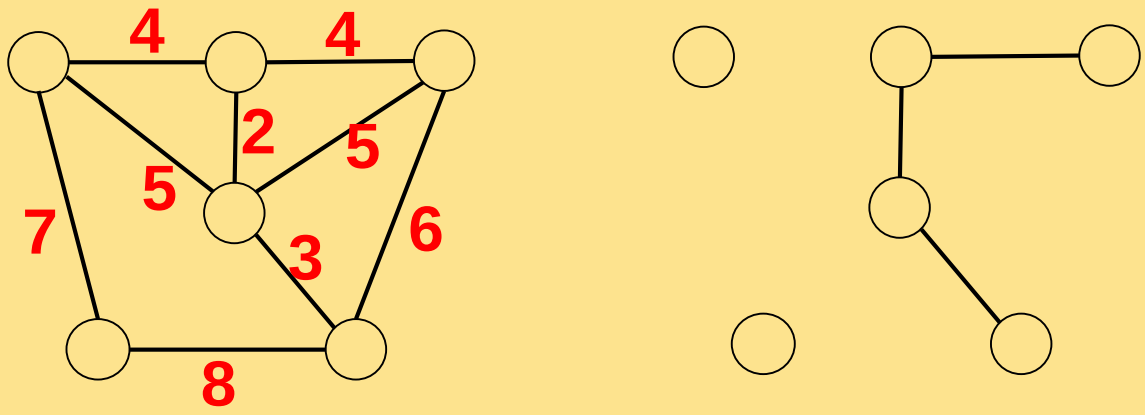


Algorithm of Kruskal:

take always the shortest edge that does not produce a cycle

Discussion Home Exercise

Exercise 1: Little Slopy Village

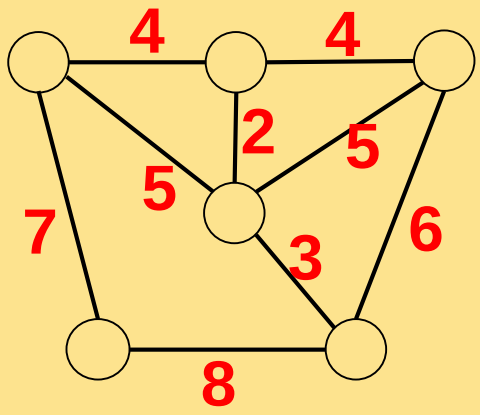


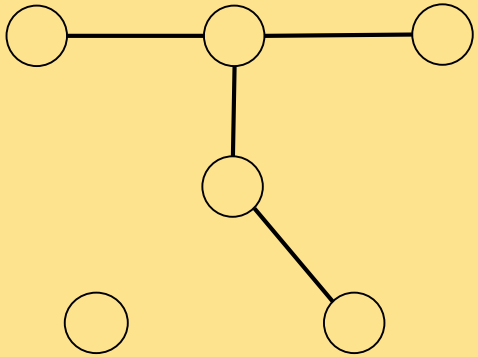
Algorithm of Kruskal:
take always the shortest edge that does not produce a cycle

! (way to the) solution not unique!

Discussion Home Exercise

Exercise 1: Little Slopy Village



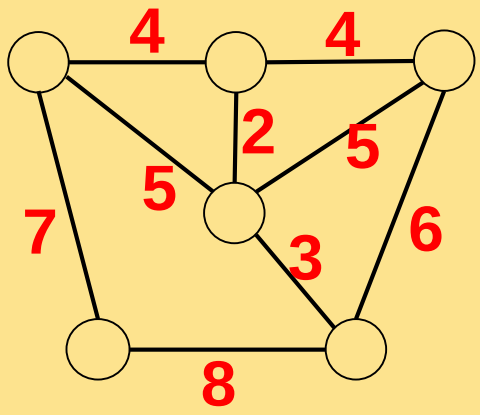


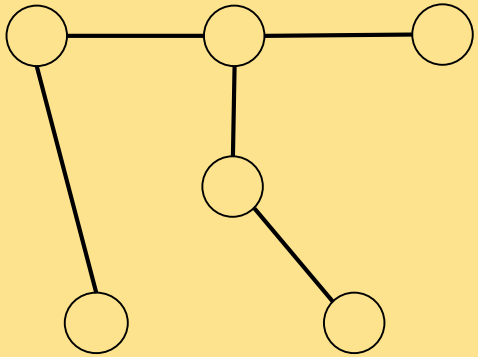
Algorithm of Kruskal:
take always the shortest edge that does not produce a cycle

! (way to the) solution not unique!

Discussion Home Exercise

Exercise 1: Little Slopy Village





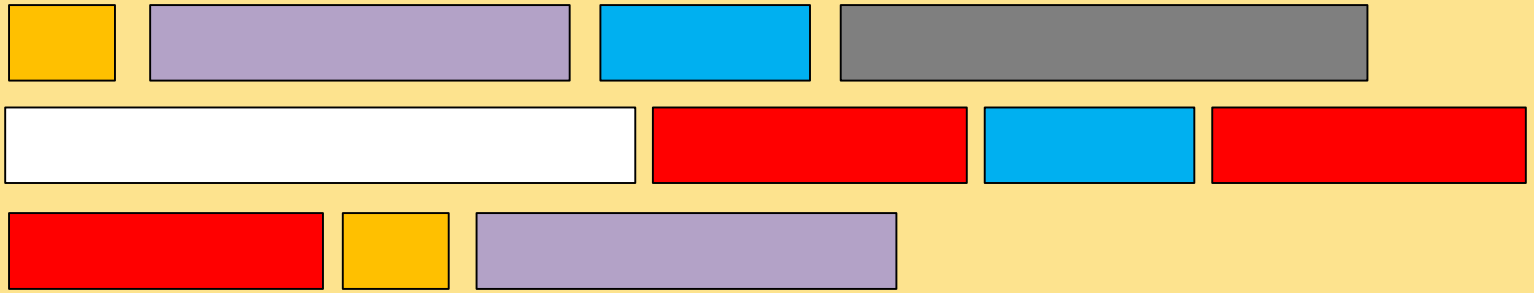
Algorithm of Kruskal:
take always the shortest edge that does not produce a cycle

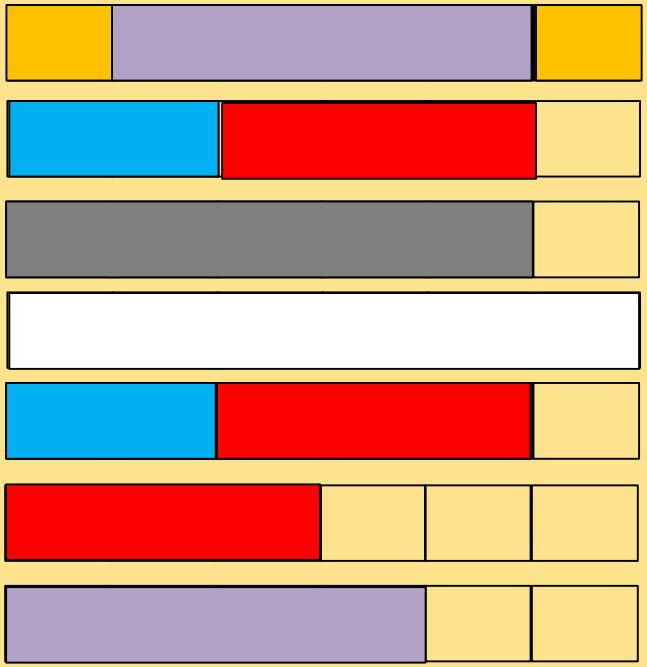
! (way to the) solution not unique!

Discussion Home Exercise

Exercise 2: Bin Packing

- items of size 1, 4, 2, 5, 6, 3, 2, 3, 3, 1, 4
- bin size: 6, first fit strategy

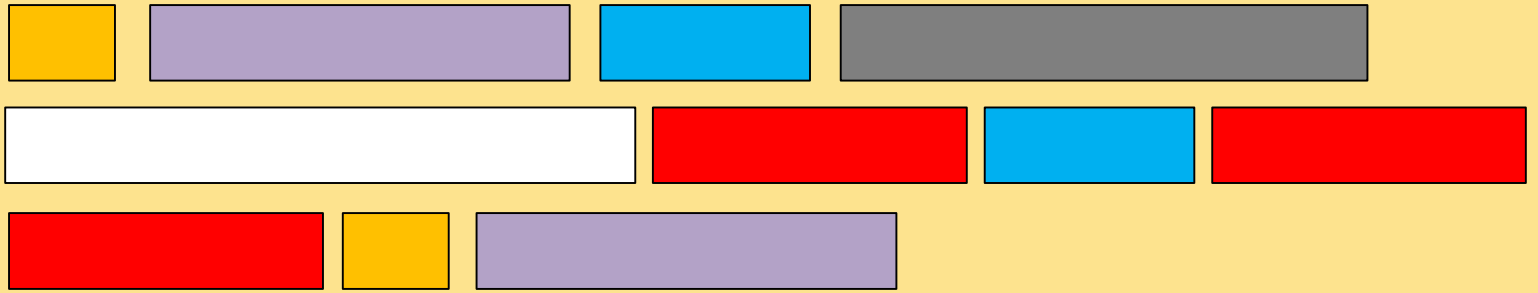




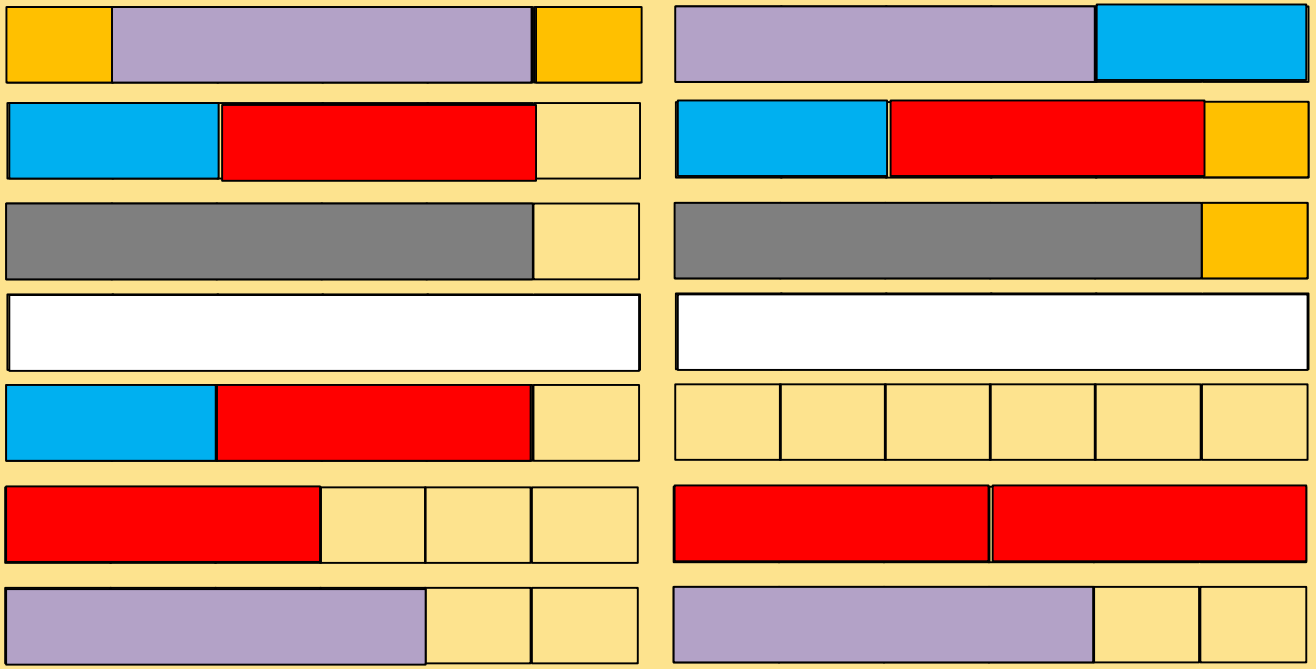
Discussion Home Exercise

Exercise 2: Bin Packing

- items of size 1, 4, 2, 5, 6, 3, 2, 3, 3, 1, 4
- bin size: 6, first fit strategy



Optimal?



Discussion Home Exercise

Exercise 3: Assisting in a Robbery

- n items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, max. load W
- calls for the knapsack problem

a) Potential Greedy Algorithm:

- take items according to their value-per-weight ratio v_i/w_i until total weight W is reached

b) Analysis

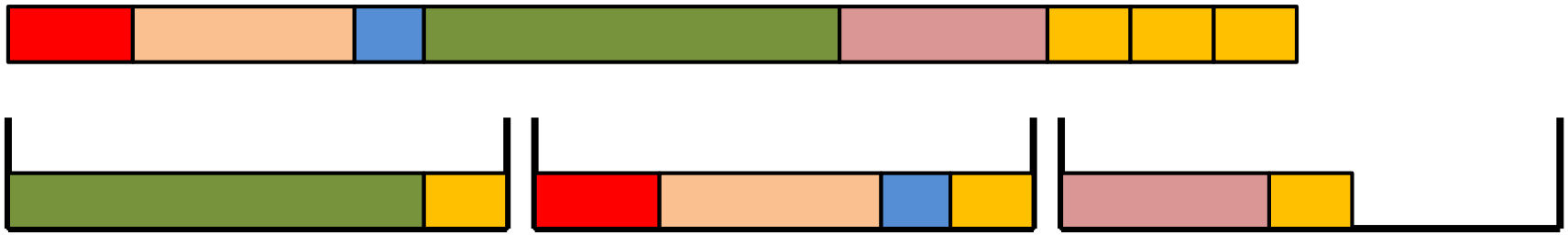
- Choice is easy (i.e. $O(n)$) as soon as we know the sorted ratios
- Ratio computing takes $O(n)$ divisions
- Sorting takes $O(n \log n)$ comparisons
- Overall, therefore: runtime of $O(n \log n)$

back to
greedy algorithms and Bin Packing

Example 3: Bin Packing (BP)

Bin Packing Problem

Given a set of n items with sizes a_1, a_2, ..., a_n. Find an assignment of the a_i's to bins of size V such that the number of bins is minimal and the sum of the sizes of all items assigned to each bin is $\leq V$.



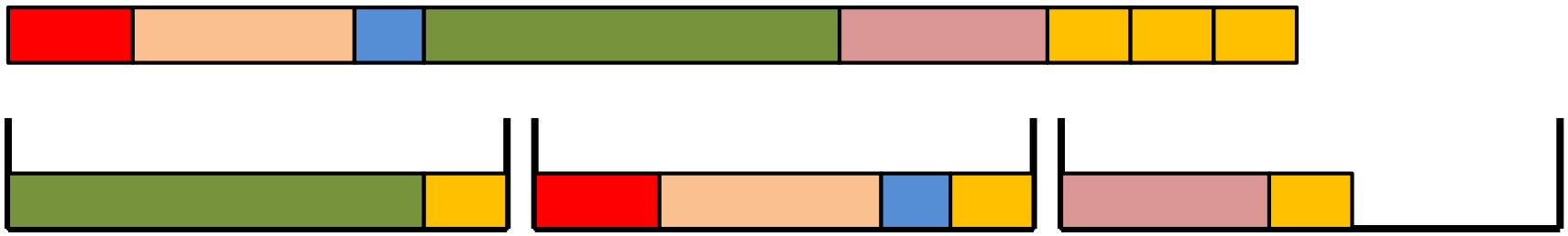
Applications

similar to multiprocessor scheduling of n jobs to m processors

Example 3: Bin Packing (BP)

Bin Packing Problem

Given a set of n items with sizes a_1, a_2, ..., a_n. Find an assignment of the a_i's to bins of size V such that the number of bins is minimal and the sum of the sizes of all items assigned to each bin is $\leq V$.



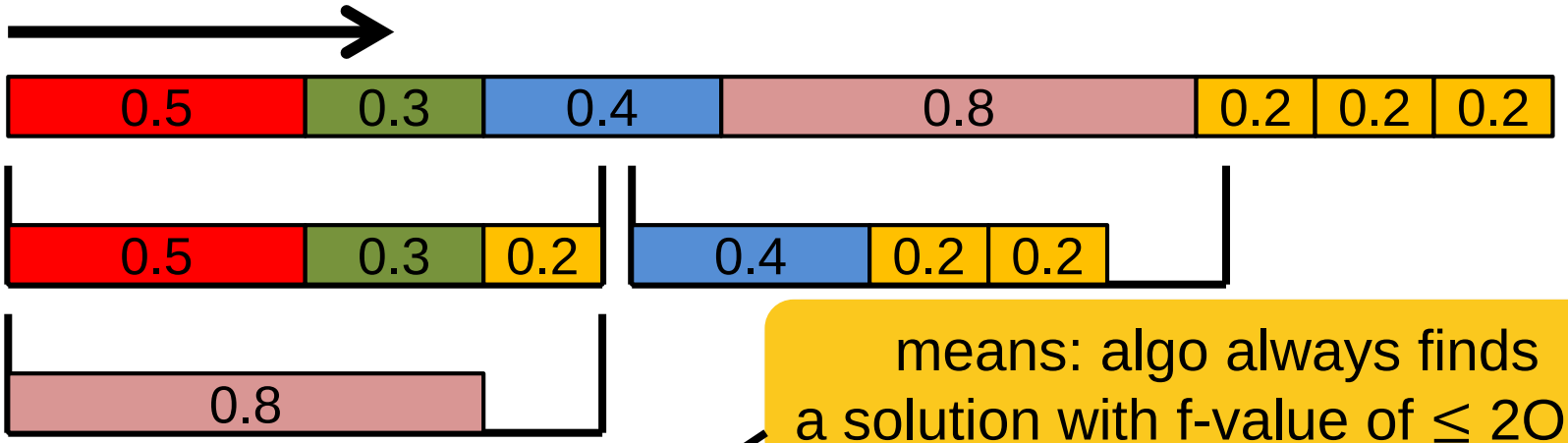
Known Facts

- no optimization algorithm reaches a better than 3/2 approximation in polynomial time (not shown here)
- greedy first-fit approach already yields an approximation algorithm with approximation ratio of 2

First-Fit Approach

First-Fit Algorithm

- without sorting the items do:
  - put each item into the first bin where it fits
  - if it does not fit anywhere, open a new bin



means: algo always finds a solution with f-value of ≤ 2OPT

Theorem: First-Fit algorithm is a 2-approximation algorithm

Proof: Assume First Fit uses m bins. Then, at least m-1 bins are more than half full (otherwise, move items).

$$\mathrm{OPT} > \frac{m-1}{2} \iff 2\mathrm{OPT} > m-1 \implies 2\mathrm{OPT} \geq m$$

↑because m and OPT are integer

Conclusion Greedy Algorithms I

What we have seen so far:

- three problems where a greedy algorithm was optimal
  - money change
  - circle packing
  - minimum spanning tree (Kruskal's algorithm)
- but also: greedy not always optimal
  - see the example of bin packing
  - this is true in particular for so-called NP-hard problems

Obvious Question: when is greedy good?

Answer: if the problem is a matroid (not covered here)

From Wikipedia: [...] a matroid is a structure that captures and generalizes the notion of linear independence in vector spaces. There are many equivalent ways to define a matroid, the most significant being in terms of independent sets, bases, circuits, closed sets or flats, closure operators, and rank functions.

Conclusions Greedy Algorithms II

I hope it became clear...

...what a greedy algorithm is
...that it not always results in the optimal solution
...but that it does if and only if the problem is a matroid

Course Overview

Thu		Topic
Thu, 12.09.2019	PM	Introduction, Combinatorics, O-notation, data structures
Tue, 24.09.2019	PM	Sorting algorithms I
Tue, 1.10.2019	PM	Sorting algorithms II, recursive algorithms
Tue, 8.10.2019	PM	Recursive and Greedy Algorithms
Tue, 15.10.2019	PM	Dynamic programming
Thu, 31.10.2019	AM	Randomized Algorithms and Blackbox Optimization
Tue, 5.11.2019	PM	Complexity theory I
Tue, 26.11.2019	PM	Complexity theory II
Tue, 17.12.2019	AM	Exam (written)

Dynamic Programming

Wikipedia:

> "[...] **dynamic programming** is a method for solving a complex problem by breaking it down into a collection of simpler subproblems."

But that's not all:

- dynamic programming also makes sure that the subproblems are not solved too often but only once by keeping the solutions of simpler subproblems in memory ("trading space vs. time")
- it is an exact method, i.e. in comparison to the greedy approach, it always solves a problem to optimality

Two Properties Needed

Optimal Substructure

A solution can be constructed efficiently from optimal solutions of sub-problems

Overlapping Subproblems

Wikipedia: “[...] a problem is said to have **overlapping subproblems** if the problem can be broken down into subproblems which are reused several times or [if] a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.”

Note: in case of optimal substructure but independent subproblems, often greedy algorithms are a good choice; in this case, dynamic programming is often called “divide and conquer” instead

Main Idea Behind Dynamic Programming

Main idea: solve larger subproblems by breaking them down to smaller, easier subproblems in a recursive manner

Typical Algorithm Design:

1. decompose the problem into subproblems and think about how to solve a larger problem with the solutions of its subproblems
2. specify how you compute the value of a larger problem recursively with the help of the optimal values of its subproblems ("Bellman equation")
3. bottom-up solving of the subproblems (i.e. computing their optimal value), starting from the smallest by using a table structure to store the optimal values and the Bellman equality

   (top-down approach also possible, but less common)
4. eventually construct the final solution (can be omitted if only the value of an optimal solution is sought)

Bellman Equation (aka “Principle of Optimality”)

- introduced by R. Bellman as “Principle of Optimality” in 1957
- the basic equation underlying dynamic programming
- necessary condition for optimality

citing Wikipedia:

> “Richard Bellman showed that a dynamic optimization problem in discrete time can be stated in a recursive, step-by-step form known as backward induction by writing down the relationship between the value function in one period and the value function in the next period. The relationship between these two value functions is called the "Bellman equation".”
>
> - The value function here is the objective function.
> - The Bellman equation exactly formalizes how to compute the optimal function value for a larger subproblem from the optimal function value of smaller subproblems.

we will see examples later today...

Historical Note

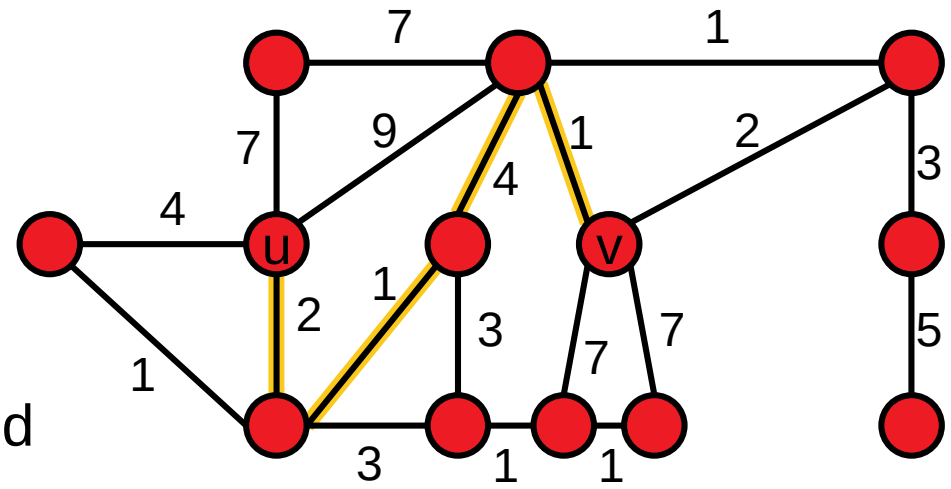
Why is it called “dynamic” and why “programming”?

- R. Bellman worked at the time, when he “invented” the idea, at the RAND Corporation who were strongly connected with the Air Force
- In order to avoid conflicts with the head of the Air Force at this time, R. Bellman decided against using terms like “mathematical” and he liked the word *dynamic* because it “has an absolutely precise meaning” and cannot be used “in a pejorative sense”
- in addition, it had the right meaning: “I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying.”
- Citing Wikipedia: “The word *programming* referred to the use of the method to find an optimal *program*, in the sense of a military schedule for training or logistics.”

A First Example: Shortest Path Problem

Shortest Path problem:

Given a graph G=(V,E) with edge weights w_i for each edge e_i. Find the shortest path from a vertex v to a vertex u, i.e., the path (v, e_1={v, v_1}, v_1, ..., v_k, e_k={v_k,u}, u) such that w_1 + ... + w_k is minimized.



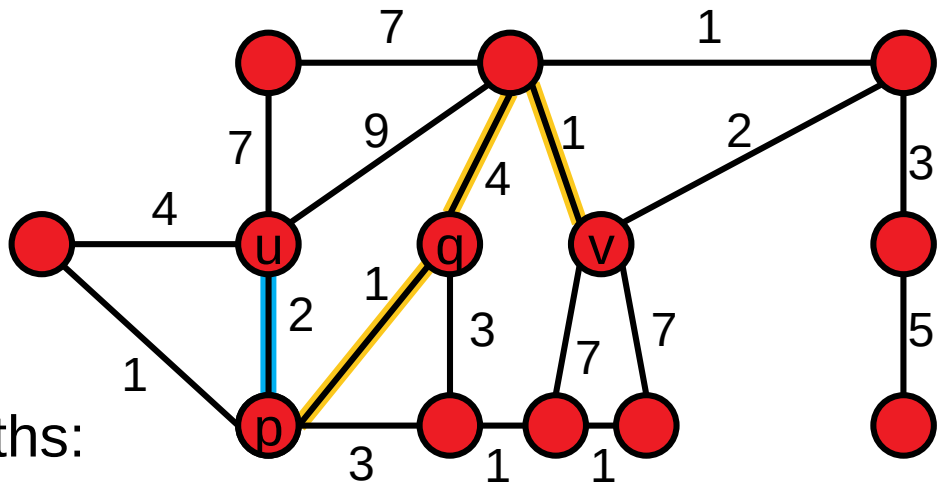
Note:

We can often assume that the edge weights are stored in a distance matrix D of dimension |V|x|V| where an entry $D_{i,j}$ gives the weight between nodes i and j and “non-edges” are assigned a value of ∞

Opt. Substructure and Overlapping Subproblems

Optimal Substructure

The optimal path from u to v, if it contains another vertex p can be constructed by simply joining the optimal path from u to p with the optimal path from p to v.



Overlapping Subproblems

Optimal shortest sub-paths can be reused when computing longer paths: e.g. the optimal path from u to p is contained in the optimal path from u to q and in the optimal path from u to v.

The Algorithm of E. Dijkstra (1956)

Basic Idea:

- distinguish between visited and unvisited nodes
- in each step visit only one new node
- How?
  - choose the one with smallest distance to the current set of nodes
  - update all shortest path lengths of the new point's neighbors
  - keep track of second-to-last node on those shortest paths

The Algorithm of E. Dijkstra (1956)

ShortestPathDijkstra(G, D, source, target):

Initialization:

- dist(source) = 0 and for all $v \in V$: dist(v)= $D_{source,v}$
- for all $v \in V$:
  if $D_{source,v}$ finite: prev(v) = source # predecessors on opt. path
  else: prev(v) = None
- U = V \ {source} # U: unexplored vertices

Unless U empty do:

- newNode = $\text{argmin}_{u \in U}$ {dist(u)}
- remove newNode from U
- for each neighbor v of newNode do:
  - alternativeDist = dist(newNode) + $D_{newNode,v}$
  - if alternativeDist < dist(v):
    - dist(v) = alternativeDist
    - prev(v) = u

The Algorithm of R. Floyd (1962)

Idea:

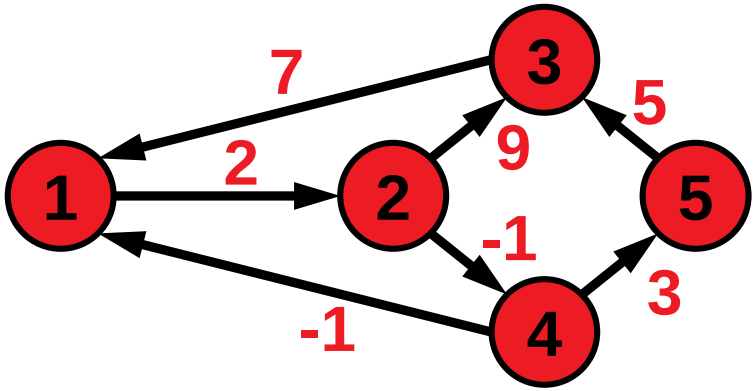
- if we knew that the shortest path between source and target goes through node v, we would be able to construct the optimal path from the shorter paths "source→v" and "v→target"
- subproblem P(k): compute all shortest paths where the intermediate nodes can be chosen from v_1, ..., v_k

ShortestPathFloyd(G, D, source, target) [= AllPairsShortestPath(G)]

- Init: for all $1 \le i,j \le |V|$: $dist(i,j) = D_{i,j}$
- For k = 1 to $|V|$ # solve subproblems P(k)
  - for all pairs of nodes (i.e. $1 \le i,j \le |V|$):
    - $dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$

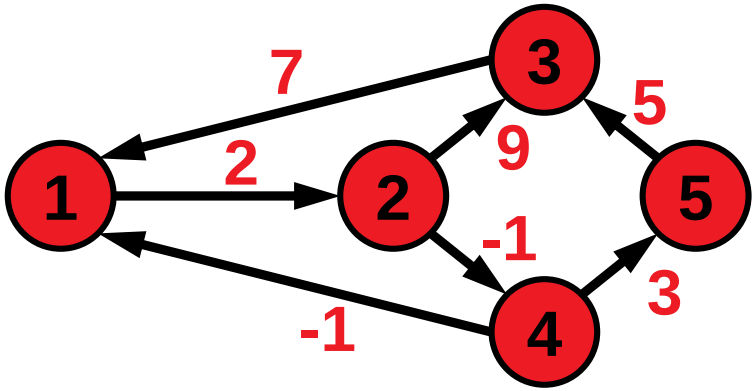
Note: This algorithm has the advantage that it can handle negative weights as long as no cycle with negative total weight exists

Example



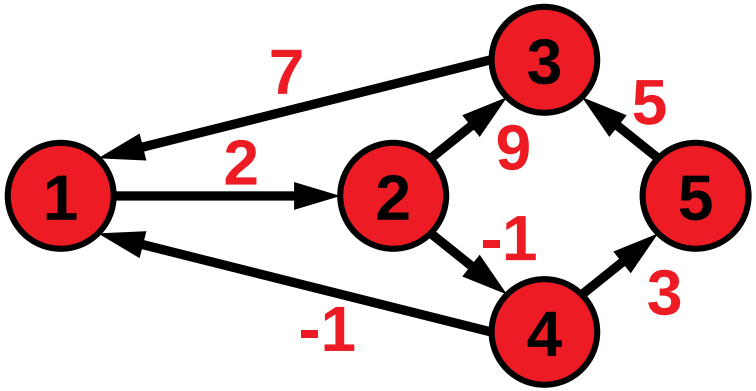
k=0	1	2	3	4	5
1					
2					
3					
4					
5					

Example



k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

Example

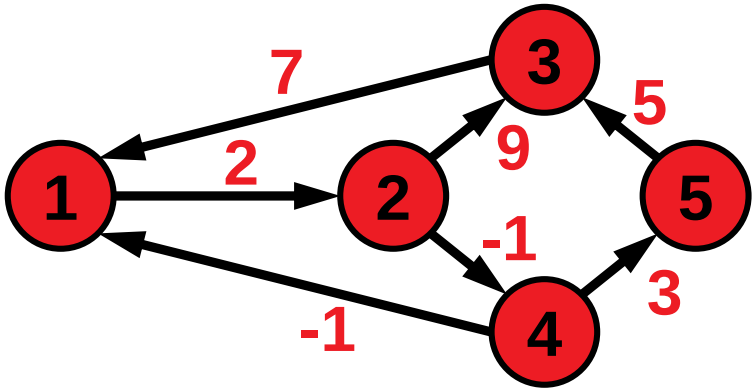


allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1					
2					
3					
4					
5					

Example

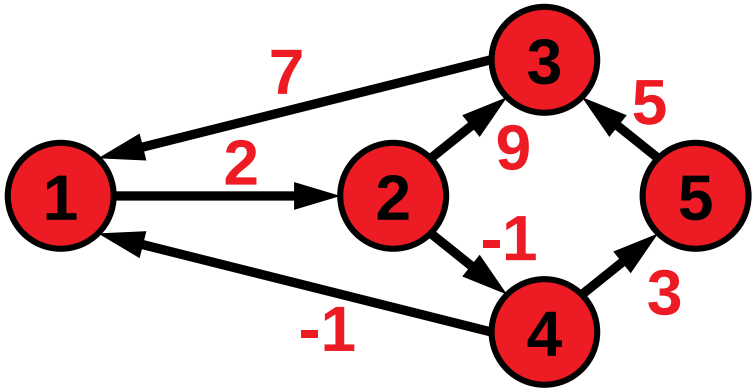


allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1					
2					
3					
4					
5					

Example

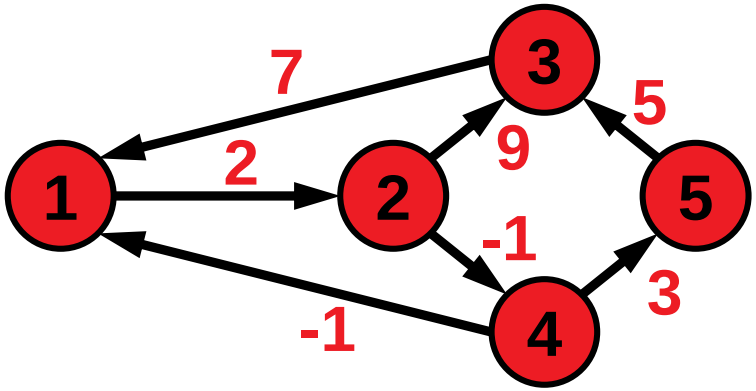


allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1					
2					
3					
4					
5					

Example

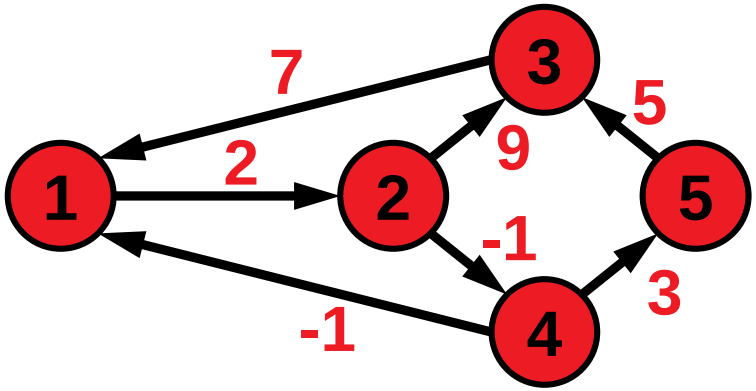


allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1					
2					
3		9			
4		1			
5					

Example

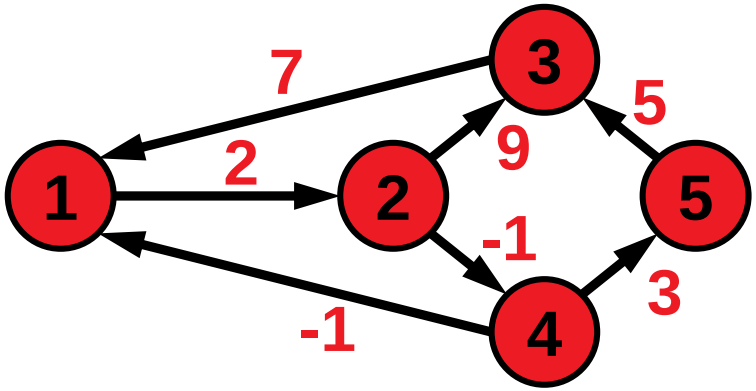


allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

Example

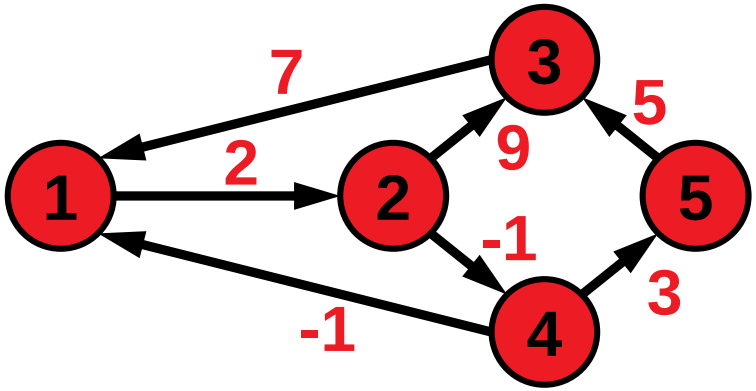


allow 1 & 2 as intermediate nodes

k=1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

k=2	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

Example

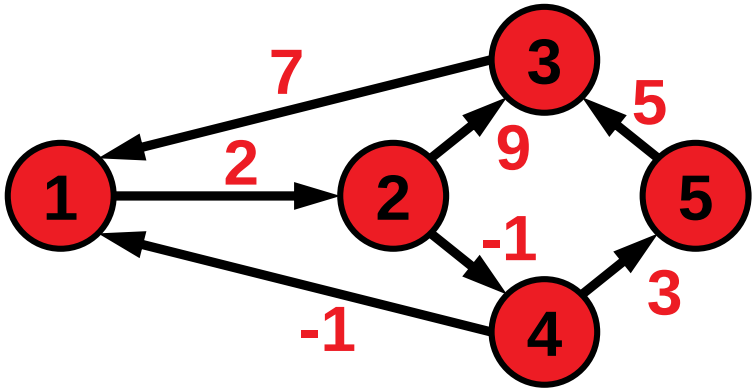


allow 1 & 2 as intermediate nodes

k=1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

k=2	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

Example

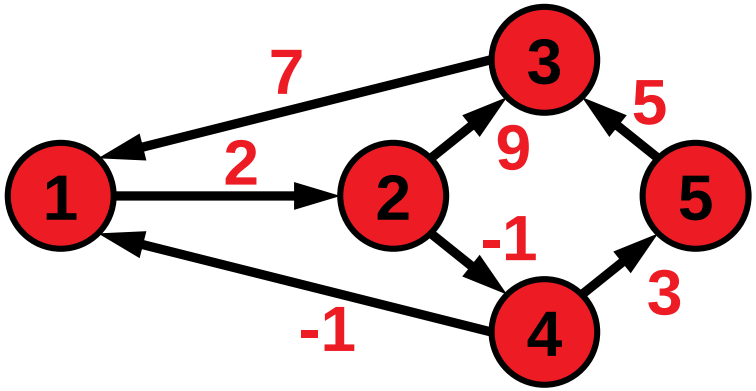


allow 1 & 2 as intermediate nodes

k=1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

k=2	1	2	3	4	5
1	∞	2	11	1	∞
2	∞	∞	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	∞	∞	5	∞	∞

Example

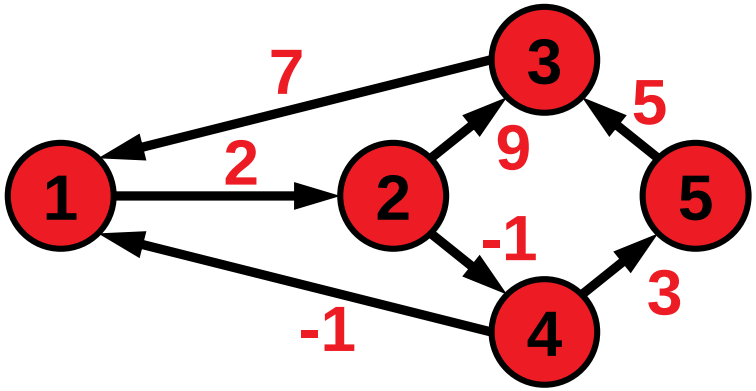


allow {1,2,3} as intermediate nodes

k=2	1	2	3	4	5
1	∞	2	11	1	∞
2	∞	∞	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	∞	∞	5	∞	∞

k=3	1	2	3	4	5
1	∞	2	11	1	∞
2	∞	∞	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	∞	∞	5	∞	∞

Example

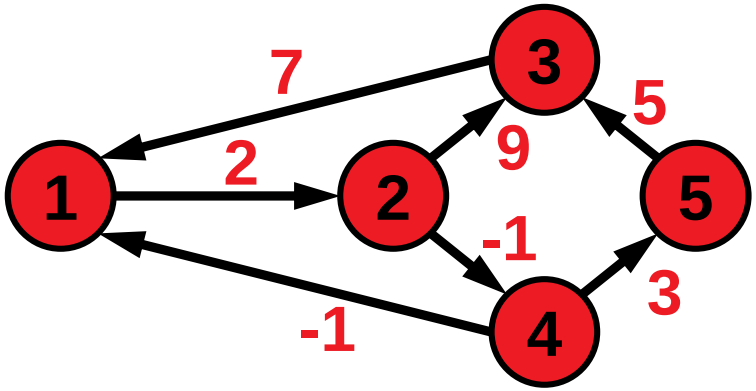


allow {1,2,3} as intermediate nodes

k=2	1	2	3	4	5
1	∞	2	11	1	∞
2	∞	∞	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	∞	∞	5	∞	∞

k=3	1	2	3	4	5
1			11		∞
2			9		∞
3	7	9	18	8	∞
4			10		3
5			5		∞

Example

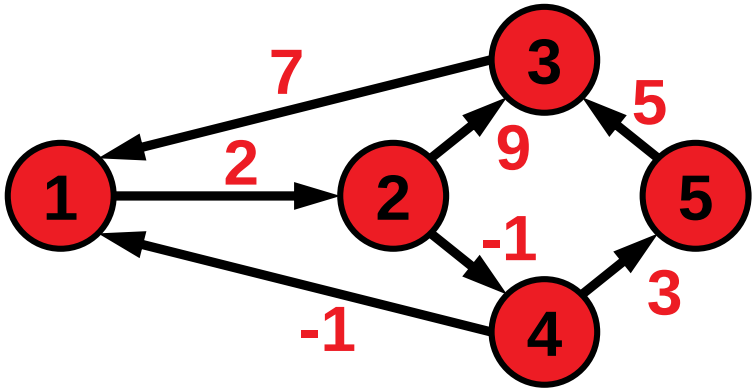


allow {1,2,3} as intermediate nodes

k=2	1	2	3	4	5
1	∞	2	11	1	∞
2	∞	∞	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	∞	∞	5	∞	∞

k=3	1	2	3	4	5
1	18	2	11	1	∞
2	16	18	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	12	14	5	13	∞

Example

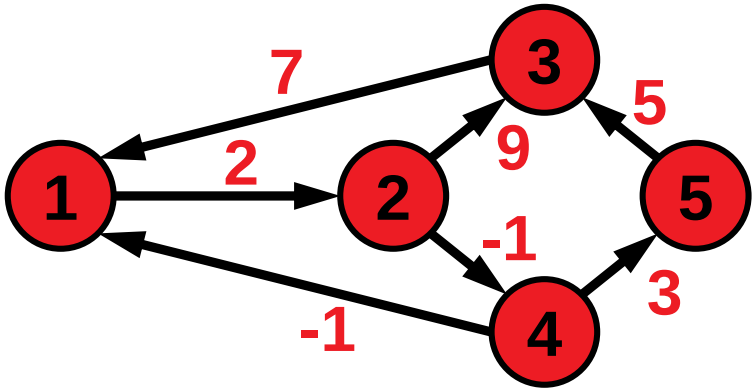


allow {1,2,3,4} as intermediate nodes

k=3	1	2	3	4	5
1	18	2	11	1	∞
2	16	18	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	12	14	5	13	∞

k=4	1	2	3	4	5
1	18	2	11	1	∞
2	16	18	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	12	14	5	13	∞

Example

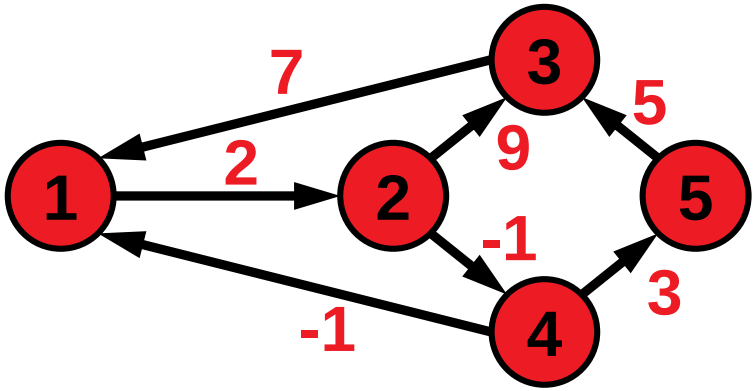


allow {1,2,3,4} as intermediate nodes

k=3	1	2	3	4	5
1	18	2	11	1	∞
2	16	18	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	12	14	5	13	∞

k=4	1	2	3	4	5
1				1	
2				-1	
3				8	
4	-1	1	10	0	3
5				13	

Example

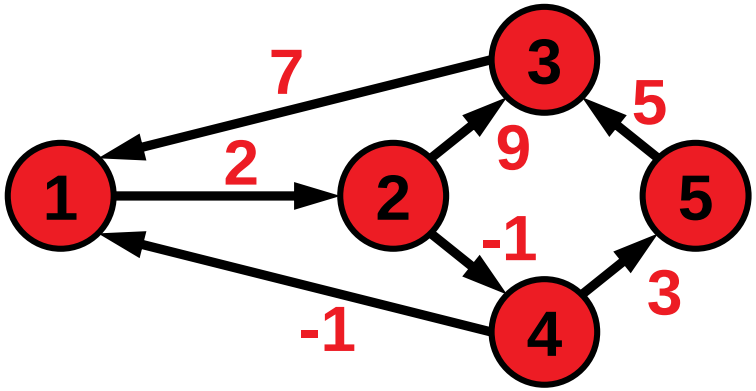


allow {1,2,3,4} as intermediate nodes

k=3	1	2	3	4	5
1	18	2	11	1	∞
2	16	18	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	12	14	5	13	∞

k=4	1	2	3	4	5
1	0	2	11	1	4
2	-2	0	9	-1	2
3	7	9	18	8	11
4	-1	1	10	0	3
5	12	14	5	13	16

Example

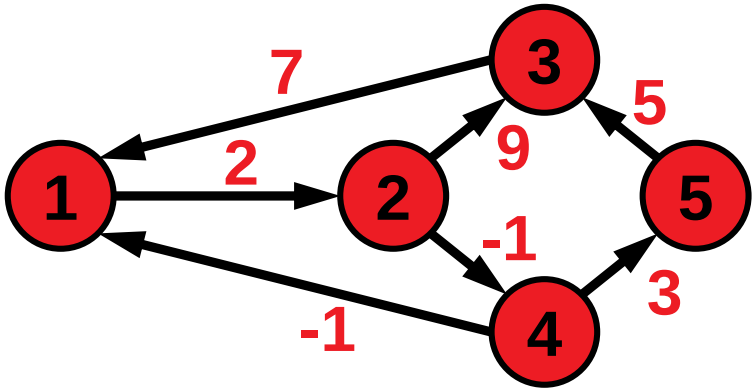


allow all nodes as intermediate nodes

k=4	1	2	3	4	5
1	0	2	11	1	4
2	-2	0	9	-1	2
3	7	9	18	8	11
4	-1	1	10	0	3
5	12	14	5	13	16

k=5	1	2	3	4	5
1	0	2	11	1	4
2	-2	0	9	-1	2
3	7	9	18	8	11
4	-1	1	10	0	3
5	12	14	5	13	16

Example



allow all nodes as intermediate nodes

k=4	1	2	3	4	5
1	0	2	11	1	4
2	-2	0	9	-1	2
3	7	9	18	8	11
4	-1	1	10	0	3
5	12	14	5	13	16

k=5	1	2	3	4	5
1	0	2	9	1	4
2	-2	0	7	-1	2
3	7	9	16	8	11
4	-1	1	8	0	3
5	12	14	5	13	16

Runtime Considerations and Correctness

$O(|V|^3)$ easy to show

- $O(|V|^2)$ many distances need to be updated $O(|V|)$ times

Correctness

- given by the Bellman equation

  $$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$$

- only correct if cycles do not have negative total weight (can be checked in final distance matrix if diagonal elements are negative)

But How Can We Actually Construct the Paths?

- Construct matrix of predecessors P alongside distance matrix
- $P_{i,j}(k)$ = predecessor of node j on path from i to j (at algo. step k)
- no extra costs (asymptotically)

$$P_{i,j}(0) = \begin{cases} 0 & \text{if } i = j \text{ or } d_{i,j} = \infty \\ i & \text{in all other cases} \end{cases}$$

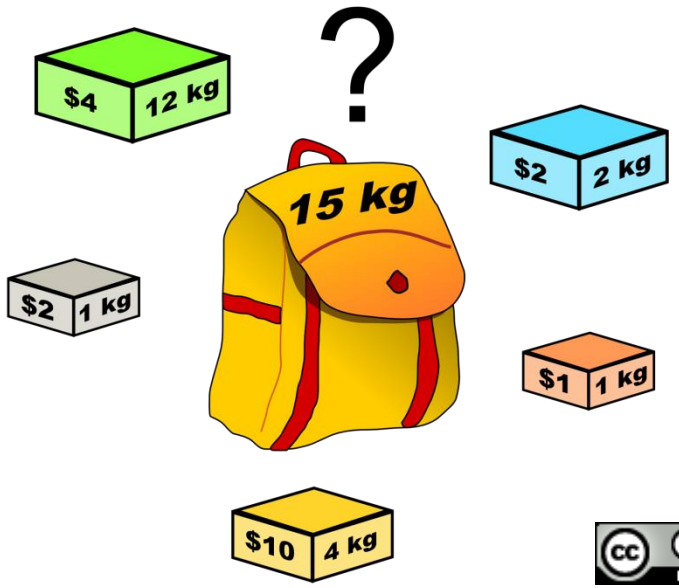
$$P_{i,j}(k) = \begin{cases} P_{i,j}(k-1) & \text{if } \mathrm{dist}(i,j) \leq \mathrm{dist}(i,k) + \mathrm{dist}(k,j) \\ P_{k,j}(k-1) & \text{if } \mathrm{dist}(i,j) > \mathrm{dist}(i,k) + \mathrm{dist}(k,j) \end{cases}$$

A Second Example: The 0-1 Knapsack Problem

0-1 Knapsack Problem (KP)

$$\max. \sum_{j=1}^{n} p_j x_j \text{ with } x_j \in \{0,1\}$$

$$\text{s.t.} \sum_{j=1}^{n} w_j x_j \leq W$$



Dake

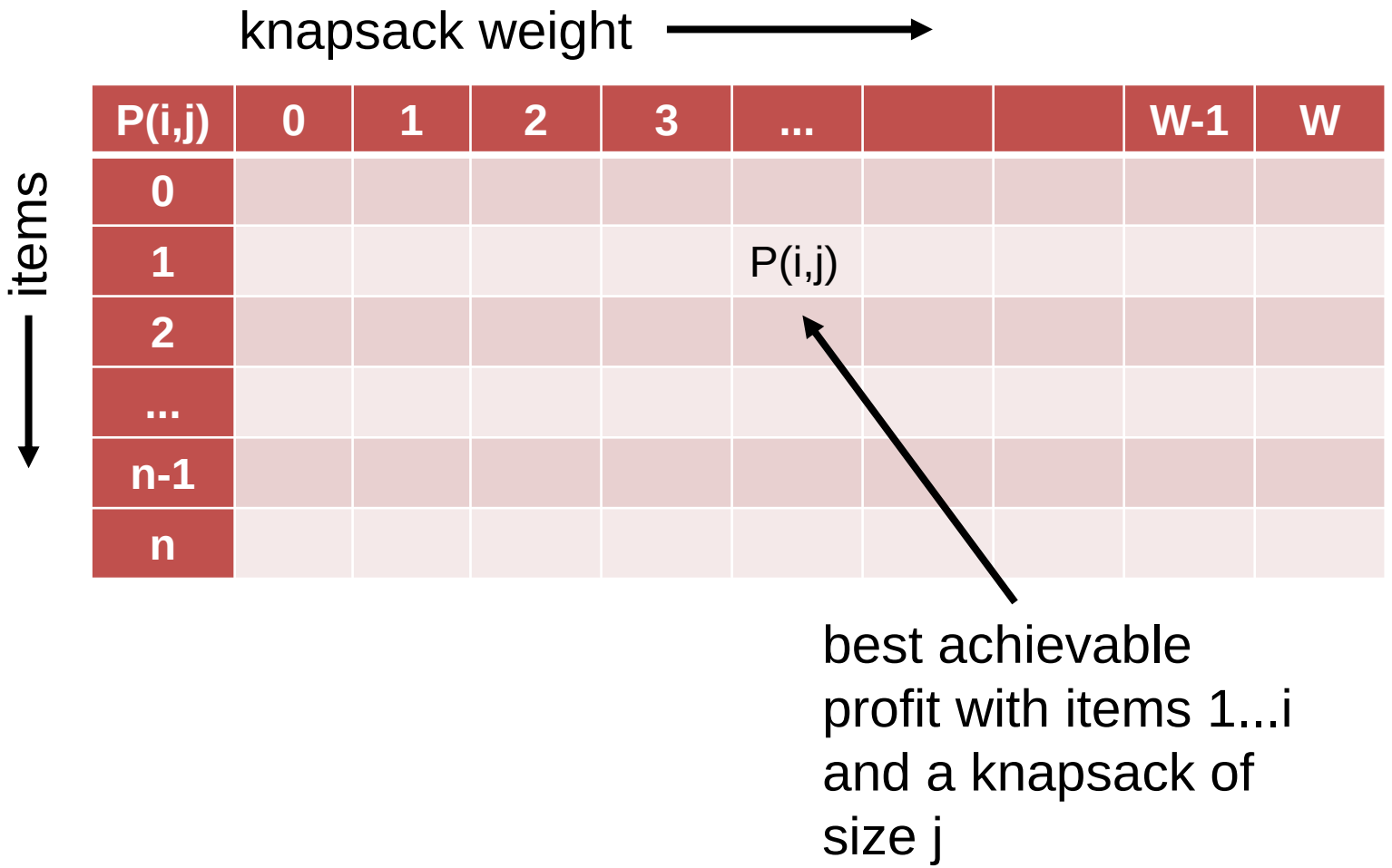
Goal: a dynamic programming algorithm for KP

Questions:

a) what could be subproblems?
b) how to solve subproblems with the help of smaller ones?
c) how to solve the smallest subproblems exactly?

Dynamic Programming Approach to the KP

To circumvent computing the subproblems more than once, we can store their results (in a matrix for example)...



P(i,j)	0	1	2	3	...			W-1	W
0									
1					P(i,j)				
2									
...									
n-1									
n									

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is W=11.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0												
1												
2												
3												
4												
5												

initialization:

$P(i,j) = 0$ if $i = 0$ or $j = 0$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is W=11.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0											
2	0											
3	0											
4	0											
5	0											

initialization:

$P(i,j) = 0$ if $i = 0$ or $j = 0$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0											
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0										
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0									
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0							
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4						
2	0											
3	0											
4	0											
5	0											

$+p_1 (= 4)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4					
2	0											
3	0											
4	0											
5	0											

$+p_1(=4)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4					
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10				
3	0											
4	0											
5	0											

$+p_2 (= 10)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3							
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4						
4	0											
5	0											

$+p_3(=3)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4					
4	0											
5	0											

$+p_3(=3)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	etc.			
4	0											
5	0											

$+p_3 (= 3)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	10	13	13	13
4	0	0	3	3	5	5	8	10	10	13	13	15
5	0	0	3	3	5	6	8	10	10	13	13	15

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	10	13	13	13
4	0	0	3	3	5	5	8	10	10	13	13	15
5	0	0	3	3	5	6	8	10	10	13	13	**15**

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Question: How to obtain the actual packing?

Answer: we just need to remember where the max came from!

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	10	13	13	13
4	0	0	3	3	5	5	8	10	10	13	13	15
5	0	0	3	3	5	6	8	10	10	13	13	15

$x_1 = 0$, $x_2 = 1$, $x_3 = 0$, $x_4 = 1$, $x_5 = 0$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Runtime Considerations

- If we try all possible combinations, we can solve the KP in time $O(2^n)$
- With the dynamic programming approach, we can do it in $O(nW)$
- For small enough weights (of the knapsack), this is quicker
- We might come back to this in the lectures on computational complexity…

Conclusions

I hope it became clear...

...what the algorithm design idea of dynamic programming is
...for which problem types it is supposed to be suitable
...and how to apply the idea to the knapsack problem