COMP20007 Workshop Week 8: Dynamic Programming

1	DP: efficient implementation for recursive algorithms • Warshall's Algorithm: Questions 8.1 & 8.2 • Floyd's Algorithm: Question 8.3
2	DP: Typical Application for Optimization Tasks Question 8.4 (Baked Bean Bundles)
LAB	baked bean bundles (homework implementation)

Dynamic Programming: great technique, fancy name

Dynamic Programming = **'remembering solved sub-problems in a table and re-use whenever possible'**

Programming= planning/table filling

Dynamic= multi-stage, walking-around

Key-words in DP:

What kind of problems can be solved with DP?

Dynamic Programming: great technique, fancy name

Dynamic Programming = **'remembering solved sub-problems in a table and re-use whenever possible'**

Programming= planning/table filling

Dynamic= multi-stage, walking-around

Key-words in DP:

- memorization, table
- problem, sub-problem, parameters
- bottom-up de-recursion

What kind of problems can be solved with DP?

problems that

- have overlapping sub-problems?
- can be divided into smaller, simpler sub-problems?
- are recursive in nature?
- are optimization problem just like the knapsack?

- Problem: finding fib(5), ie. fib(n) when n=5
- Sub-problem: finding fib(k) where k<n
- Recursive soln is a top-down soln
  - fib(5) = fib(4)+fib(3)
  - fib(4) = fib(3)+fib(2)
  - fib(3) = …
  - …
- Using DP we work bottom-up and store soln of sub-problems in a table for re-use
  - fib(1)= 1 → DP_1 = 1
  - fib(2)= 1 → DP_2 =
  - fib(3)= fib(2)+fib(1)= DP_2+DP_1 → DP_3 =
  - fib(4)= fib(3)+fib(2)= DP_3+DP_2 → DP_4 =
  - fib(5)= fib(4)+fib(3)= DP_4+DP_3 → DP_5 =
  - and DP5 is the soln

DP – efficient de-recursion for non-optimization tasks

Notes:

- DP is often used for optimisation tasks, but can also be used for non-optimisation tasks.
- Optimisation Tasks: finding the best solution to some problems. Examples: the knapsack problem.
- Non-optimisation Tasks: just find a (most of the times, the unique) solution. Recursive Implementation is expensive, and DP could be a more efficient one.

Examples of recursive algorithms for non-optimisation tasks:

- computing n!
- computing fib(n)
- computing the transitive closure of a graph

DP: how?

The most important & difficult steps in DP are to w

- the **problem, sub-problems and parameters**
  `fib(n)`, `fib(k)` and `k`
- the **relationship** amongst sub-problems
  `fib(k)= fib(k–1)+fib(k–2)`
- the base case(s)
  `fib(1)=1, fib(2)= 1`

- Problem: finding fib(5), ie. fib(n) when n=5
- Sub-problem: finding fib(k) where k<n
  - fib(1)= 1 → DP_1 = 1
  - fib(2)= 1 → DP_2 = 1
  - fib(3)= fib(2)+fib(1)= DP_2+DP_1 → DP_3 = 2
  - fib(4)= fib(3)+fib(2)= DP_3+DP_2 → DP_4 = 3
  - fib(5)= fib(4)+fib(3)= DP_4+DP_3 → DP_5 = 5
  - and DP5 is the soln

Implementation is often straightforward, in a non-recursive, bottom-up manner:

- build a table to store the solution for `k= 0..n` where `n` is the problem size
- fill the table in the bottom-up manner, starting from base cases

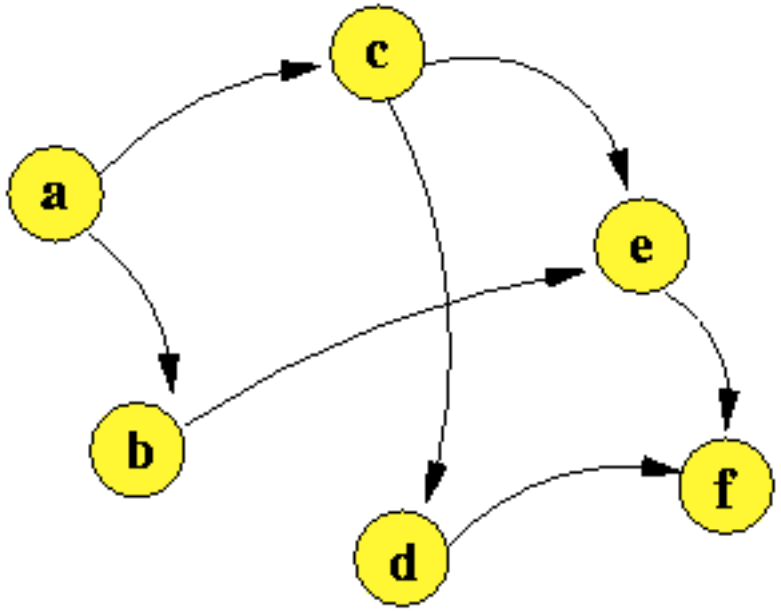
```
function fib(n)
   F[1..n], F[1]= 1, F[2]= 1
   for k:= 3 to n
      F[k]= F[k–1] + F[k–2]
   return F[n]
```

we can save memory by using just 3 cells instead of F

```
function fib(n)
   F1=1, F2=1, Fk= 1
   for k:= 3 to n
      Fk= F1 + F2
      F2= Fk, F1= F2
   return Fk
```

Transitive Closure of digraphs

Understanding
Transitive Closure



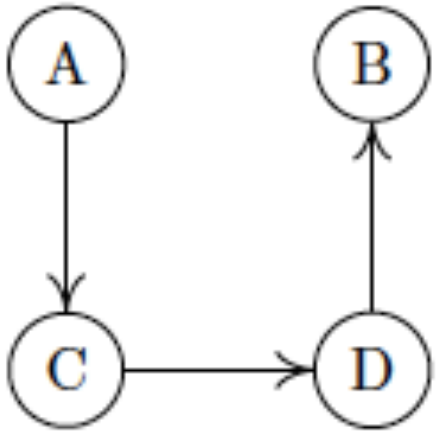
Related Tasks:

- Compute the transitive closure for a digraph
- Find APSP for a weighted graph

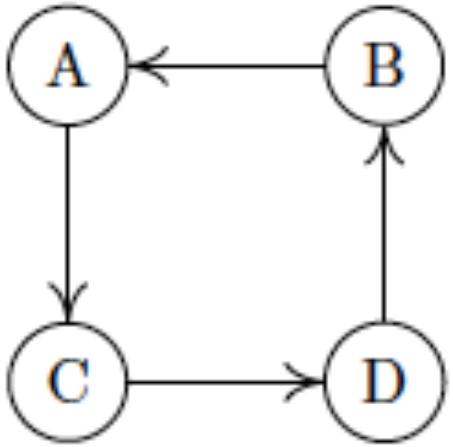
Q8.1: Transitive Closure of digraphs

Draw the transitive closure of the following two graphs:

(a)



(b)



Warshall's Algorithm: a DP algo for Transitive Closure

Input:

- adjacent matrix A[1..n][1..n] of an input graph G

Output:

- adjacent matrix of the transitive closure of G

Main argument:

- transitiveness: if there are paths $i \rightarrow k$ and $k \rightarrow j$, then there is path $i \rightarrow j$ which uses k as an intermediate node ($i, j, k \in 1..n$)

Solve the problem using DP. First decide:

- parameters & sub-problems
- the relationship between (the solutions of) the sub-problems
- the base cases and their soln

Warshall's Algorithm: DP for Transitive Closure

parameters & sub-problems:

- DP(k): k for using node k as the transit point?
- but then:
  - how do we remember which nodes have been used as transit?
  - what's the relationship between DP(n) and some DP(k), k<n?
- how about: DP(k) for using k nodes 1..k as the transit points?

the relationship between (the solutions of) the sub-problems:

the base cases and their soln: DP(0) is the original adj matrix A of G

so: *DP(k)= all paths using nodes 1..k as intermediate points*

DP(0) = A

DP(k): DP(k) [i,j] *= ???? for k= 1..n*

DP(n) is the soln!

Write the algorithm, by first set

DP(0)= A,

then progressing k from 1 to n.

How to store *DP(k)* for lookup? Do we really need a table of 2D arrays?

Q 12.2b: Manually Tracing the Warshall's Algorithm

$$R^0_{ij} := A_{ij}, \qquad R^k_{ij} := R^{k-1}_{ij} \text{ or } \left(R^{k-1}_{ik} \text{ and } R^{k-1}_{kj}\right)$$

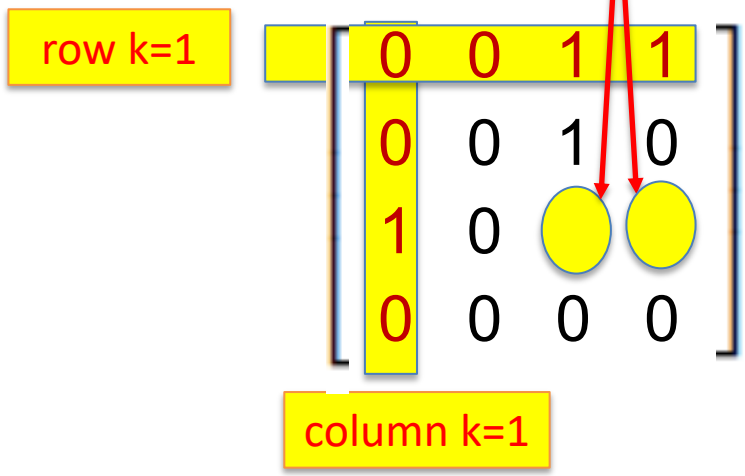
only cell with both non-zero references can change

$$R^0 = A = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

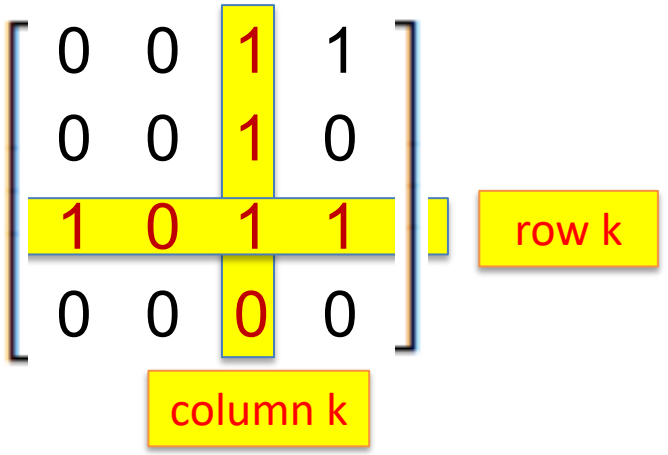
Set R= A, R is R^0

transition from 0 to 1 by:

- looking at for all possible ij
- it can be done by using column 1 and row 1 as references



similarly for any transition from R^{k-1} to R^k: use row k and column k as references.



Q12.2: Check your answer

$$R^0_{ij} := A_{ij}, \qquad R^k_{ij} := R^{k-1}_{ij} \text{ or } \left(R^{k-1}_{ik} \text{ and } R^{k-1}_{kj}\right)$$

$\Leftrightarrow$ if $R_{ij}=0$, change it to 1 iif R_{ik} and R_{kj} are both 1

$$R^0 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad R^1 = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & 0 & 1 & 0 \\ \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & 0 & 0 & 0 \end{bmatrix}$$

$$R^2 = \begin{bmatrix} 0 & \mathbf{0} & 1 & 1 \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ 1 & \mathbf{0} & 0 & 1 \\ 0 & \mathbf{0} & 0 & 0 \end{bmatrix} \qquad R^3 = \begin{bmatrix} \mathbf{1} & 0 & \mathbf{1} & 1 \\ \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ 0 & 0 & \mathbf{0} & 0 \end{bmatrix} \qquad B := R^4 = \begin{bmatrix} 1 & 0 & 1 & \mathbf{1} \\ 1 & 0 & 1 & \mathbf{1} \\ 1 & 0 & 1 & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

DP for APSP: Floyd's Algorithm

Task: APSP – For a weighted graph G, find shortest path between all pair of vertices.

Main idea:

- if we have shortest paths for i→k and for k→j then we might have a shorter path for i→j just by going from i to j using k as the intermediate node.
- the idea is just similar to the Warshall's, but

Can we build the DP relationship?

DP(0) =

DP(k) =

Q12.3: manual exec of Floyd's

Perform Floyd's algorithm on the graph given by the following weights matrix:

$$W = \begin{bmatrix} 0 & 3 & \infty & 4 \\ \infty & 0 & 5 & \infty \\ 2 & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}.$$

$$D_{ij}^0 := W_{ij}, \qquad D_{ij}^k := \min\left\{D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1}\right\}$$

Example: $D^{k-1} \rightarrow D^k$ when k=1

$$D^0_{ij} := W_{ij}, \qquad D^k_{ij} := \min\left\{D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\right\}$$

that is, update D_{ij} if $D_{ik}+D_{kj} < D_{ij}$

$$W = \begin{bmatrix} 0 & 3 & \infty & 4 \\ \infty & 0 & 5 & \infty \\ 2 & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

row **k**: $W_{kj} < \infty$ is the current shortest path **k**→j

column **k**:

$W_{ik} < \infty$ is the current shortest path i→**k**

Q 12.3: Check your answer

$$D^0_{ij} := W_{ij}, \qquad D^k_{ij} := \min\left\{D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\right\}$$

$$D^0 = \begin{bmatrix} 0 & 3 & \infty & 4 \\ \infty & 0 & 5 & \infty \\ 2 & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix} \qquad D^1 = \begin{bmatrix} 0 & 3 & \infty & 4 \\ \infty & 0 & 5 & \infty \\ 2 & \mathbf{5} & 0 & \mathbf{6} \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

$$D^2 = \begin{bmatrix} 0 & 3 & \mathbf{8} & 4 \\ \infty & 0 & 5 & \infty \\ 2 & 5 & 0 & 6 \\ \infty & \infty & 1 & 0 \end{bmatrix} \qquad D^3 = \begin{bmatrix} 0 & 3 & 8 & 4 \\ \mathbf{7} & 0 & 5 & \mathbf{11} \\ 2 & 5 & 0 & 6 \\ \mathbf{3} & \mathbf{6} & 1 & 0 \end{bmatrix} \qquad D := D^4 = \begin{bmatrix} 0 & 3 & \mathbf{5} & 4 \\ 7 & 0 & 5 & 11 \\ 2 & 5 & 0 & 6 \\ 3 & 6 & 1 & 0 \end{bmatrix}$$

Notes: the Warshall's & Floyd's Algorithm: DP vs Recursion

The time complexity of the recursive version of Warshall's is $\boldsymbol{\theta}(n^3)$ ➔ the recursive version has the same time complexity as the DP solution.

However, the recursive version

- has a larger constant factor than the DP solution, as it involves making n recursive calls
- use memory: stack memory usage overhead, including $\boldsymbol{\theta}(n^3)$ for using a matrix each time. Note that the DP soln uses $\boldsymbol{\theta}(n^2)$ memory.

➔ the DP solution is generally preferred for Warshall's algorithm.

```
function warshall_recursive(T[1..n][1..n], k):
    if k == 0 return T      # base case

    # Recursively compute the transitive closure, using nodes 1..k-1 as intermediate nodes
    T = warshall_recursive(T, k-1)
    # Compute soln for k using the results of the (k-1)-th iteration
    for i := 1 to n
        for j := 1 to n
            T[i][j] = T[i][j] or (T[i][k-1] and T[k-1][j])
    return T
```

is the DP soln for factorial(n) better than the recursive soln?

another example of DP for non-optimization tasks: all palindromes

The Task: Given a string S. Find all non-empty palindromic substrings of S.

5-minute break

DP for optimization problems

Optimization problem:

- Having an objective function f(), with possible different ways to get value for f(n)
- Find a way that maximize (or minimize) the value f(n)

Optimization problems can be solved using DP if:

- problems that can be divided into smaller, simpler sub-problems,
- the problem satisfies the ***principle of optimality:*** an optimal solution for input n is composed of optimal solutions for inputs k<n

Example: Coin-row problem
There is a row of n coins whose values are some positive integers $c_1, c_2, \ldots, c_n$, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up. ***Problem & Parameters***: ***Recurrence***: ***Base case***:

Question 8.4: Baked Beans Bundles

We have bought n cans of baked beans wholesale and are planning to sell bundles of cans at the University of Melbourne's farmers' market. Our business-savvy friends have done some market research and found out how much students are willing to pay for a bundle of k cans of baked beans, for each k ∈ {1,...,n}.

We are tasked with writing a dynamic programming algorithm to determine how we should split up out n cans into bundles to maximise the total price we will receive.

(a) Write the pseudocode for such an algorithm.

(b) Using your algorithm determine how to best split up 8 cans of baked beans, if the prices you can sell each bundle for are as follows:

Bundle Size k	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	17	17	20

(c) What's the runtime of your algorithm? What are the space requirements?

Baked Beans Bundles

(a) Design a DP algorithm, Write the pseudocode for that algorithm.

We have n=8 cans of baked beans.

We also have:

bundle size i	**0**	**1**	**2**	**3**	**…**	**n**
price p_i	0	p_1	p_2	p_3	…	p_n

example data

Bundle Size k	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	17	17	20

For this task: we want to maximize the revenue of selling n cans

Aim and Parameter:

Recurrence:

Base case:

Baked Beans Bundles

(a) Design a DP algorithm, Write the pseudocode for that algorithm.

We have n=8 cans of baked beans.

We also have:

Bundle Size k	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	17	17	20

bundle size k	**0**	**1**	**2**	**3**	**…**	**n**
price p_k	0	p_1	p_2	p_3	…	p_n

For this task: we want to maximize the revenue from selling n cans

Aim and Parameter: r(i) → max, r(i) is revenue from selling i cans

We will store r(i) in R[i] of array R[0..n]

num of cans i	**0**	**1**	**2**	**3**	**…**	**n**
F[i]	0	p_1	?		…	

Recurrence: R[i] = max(p_j + revenue from selling n-i can)= max(p_j + F(i-j)) for j= 1..n-1

Base case: R[0]=0, R[1]= p_1

BUT: **we also need a way to backtrack the used bundles! HOW?**

12.1 a) Bean Bundles: Pseudocode?

bundle size i	**0**	**1**	**2**	**3**	**…**	**n**
price p_i	0	p_1	p_2	p_3	…	p_n

cans i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
revenue R[i]	0	1	5	8	10	13	17	18	22
bundle used B[i]	0	1	2	3	2	2	6	1	2

function BakedBean(prices[1..n]) - sketch

set additional arrays: R[0..n], B[0..n] with initial values
for i ← 1 to n do # here n is number of cans we have, not number of bundles
although they have the same value in this task
need to make a loop to compute the next 2 values
R[i] ← $\max_{j \in 1..w}$ (prices[j] + R[w-i])
B[i] ← index j found in the above max
output R[n] # print max value with n cans
the print the used bundles, need to make a loop for backtracking
…

Bundle Size k	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	17	17	20

Bean Bundles: Pseudocode?

function BakedBean(prices[1..n]) #here W=n

```
# set arrays: F[0..n], B[0..n] with initial values
# and base case: F[0]= 0, B[0]= 0
R[0..n] ← {0,…,0}
B[0..n] ← {0,…,0}

for i ← 1 to n  do          # solving sub-problem for i, ie. find R[i] & B[i]
    maxval ← 0, jmax ← 0
    for j ← 1 to i do
        if R[w-i]+prices[i] > maxval then
            maxval← R[w-j]+prices[j]
            jmax ← j
    R[i] ← maxval
    B[i] ← jmax

output F[n]     # print max value with n cans
# the print the used bundles, need to make a loop for that
i ← n
while i >0 do
    output B[i]
    i ← i – B[i]
```

Baked Beans Bundles

(b) Run the algorithm manually

bundle size	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
price $	0	1	5	8	9	10	17	17	20

cans i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
revenue R[i]	0	1	5	8	10	13	17	18	22
bundle B[i]	0	1	2	3	2	2	6	1	2

bundles:

c) Complexity = ?

Running time:

Space:

Baked Beans Bundles: check your tracing

(b) Run the algorithm manually

bundle size	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
price $	0	1	5	8	9	10	17	17	20

cans i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
revenue R[i]	0	1	5	8	10	13	17	18	22
bundle B[i]	0	1	2	3	2	2	6	1	2

bundles: 2 (B[6]), then 6 (B[8-2])

c) Complexity = ?

Running time: $\boldsymbol{\theta}(n^2)$

Space: $\boldsymbol{\theta}(n)$ (n+1 sub-problems)

Notes on the bean bundles problem

The algorithm can be applied to a more general case:

General Bean Bundle Problem

Given n bundles with

- cans/weights in bundles: $w_1;w_2;\ldots;w_n$
- price/values of bundles : $v_1; v_2;\ldots;v_n$

And given amount of cans W find how we should split up out W cans into bundles to maximize the total value f(W) we can receive.

Solution is similar:

$F[0]= 0$

$F[i]= \max_{w_j \le i} (v_j + F(i - w_j))$

Bean Bundles vs. Knapsack?

Are the tasks similar?

Bean Bundles
Given n bundles with • cans/weights: $w_1;w_2;\ldots;w_n$ • price/values : $v_1; v_2;\ldots;v_n$ And given amount of cans W find how we should split up out W cans into bundles to maximise the total price we will receive.

Knapsack
Given n items with • weights: $w_1;w_2;\ldots;w_n$ • values: $v_1; v_2;\ldots;v_n$ And given knapsack of capacity W find the most valuable selection of items that will fit in the knapsack (of capacity W).

Knapsack solution looks more complicated, why?

Bean Bundles vs. Knapsack?

Are the tasks similar?

Bean Bundles

Given n bundles with
- cans/weights: $w_1;w_2;\ldots;w_n$
- price/values : $v_1; v_2;\ldots;v_n$

And given amount of cans W

find how we should split up out W cans into bundles to maximise the total price we will receive.

➔ single variable W

$F[w] = \max_i\{v_i + V[w - w_i]\}$

$F[0] = 0$

Knapsack

Given n items with
- weights: $w_1;w_2;\ldots;w_n$
- values: $v_1; v_2;\ldots;v_n$

And given knapsack of capacity W

find the most valuable selection of items that will fit in the knapsack (of capacity W).

Now $F[w] = \max_i\{v_i + F[w - w_{ij}]\}$ is useless: how do we exclude the items that are already used in $F(w - w_i)$?

Bean Bundles vs. Knapsack? Knapsack soln seems much more complicated, why?

Are the tasks similar?

Baked Beans	Knapsack
Given n bundles with • cans/weights: $w_1;w_2;\ldots;w_n$ • price/values : $v_1; v_2;\ldots;v_n$ And given amount of cans W find how we should split up out W cans into bundles to maximise the total price we will receive.	Given n items with • weights: $w_1;w_2;\ldots;w_n$ • values: $v_1; v_2;\ldots;v_n$ And given knapsack of capacity W find the most valuable selection of items that will fit in the knapsack (of capacity W).
• a single variable (the size W) • The optimal solution for a given W depends only on the optimal solutions for smaller.	• two variables (the number of items and the remaining capacity of the knapsack). • The optimal solution for a given number of items and remaining capacity depends on the optimal solutions for smaller numbers of items and remaining capacities.

Knapsack: without repetition

Now $F[w] = \max_i\{ F[w - w_i] + v_i \}$ is useless.

Need another parameter…for the used items

➔ 2 variables:

- w : remaining capacity
- i : already used items 1..i

Let $K(i, w)$ be the highest value with a knapsack of capacity w only using items $1..i$

Knapsack

Given knapsack of capacity W and n items:

item	1	2	…	n
weight	w_1	w_2		w_n
value	v_1	v_2		v_n

find the most valuable selection of items that will fit in the knapsack (of capacity W).

$$K[i,w] = \max (K[i-1, w - w_i] + v_i,\ K[i-1, w])$$

base cases: $K[i,w] = 0,\ i < 1$ or $w < 1$

Example: W=10, we start with the table:

i	w_i	v_i
1	6	$30
2	3	$14
3	4	$16
4	2	$9

$K(i,w) = \max (K(i-1, w - w_i) + v_i,\ K(i-1, w))$

$K(i,w) = 0,\ i < 1$ or $w < 1$

$i \backslash w$	0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0					
2	0										
3	0										
4	0										

Watch lecture for example on running the Knapsack

Most crucial step in DP: finding parameters & recurrences

Coin-row problem

There is a row of n coins whose values are some positive integers $c_1, c_2, \ldots, c_n$, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

Parameters:
Recurrence:
Base case:

Change-making problem

Give change for amount W using the minimum number of coins of denominations $d_1 < d_2 < \ldots < d_n$. Assume availability of unlimited quantities of coins for each of the n denominations d_i and $d_1 = 1$.

Parameters:
Recurrence:
Base case:

all palindromes

Given a string S of length n, construct a DP algorithm to find all nonempty substrings that are palindromes.

Parameters:
Recurrence:
Base case:

Some other problems

- Coin-collecting problem Levitin.8.1.example3
- Order of matrix multiplication: exercise L8.3.11

- DP is typically used in optimization problems, where we are trying to find the best possible solution given a set of constraints. Examples: 3 of the above 4 boxes, knapsack, bean bundles...
- DP can also be used to solve problems that have overlapping sub-problems or can be divided into smaller, simpler sub-problems. Examples: fibonacci, Warshall, Floyd, all palindromes…

Solve these problems, consult ChatGPT and Gemini after having enough effort!

Lab

Implement the baked bean bundles problem at home.

Notes:

- the program will be short,
- the solution will be supplied in LMS, but:
- try not to use it, and build your code from scratch without relying on the algorithm → a good way to understand & remember DP.