COMP20007 Workshop Week 12

1	Discussion: Dynamic Programming, problem 5
2	Warshall's Algorithm: problems 1 & 2
3	Floyd's Algorithm: problem 3
4	Problem 5: Revision: quicksort & top-down mergesort
LAB	baked bean bundles (implementation)

Dynamic Programming

Dynamic Programming is just a fancy way to say **'remembering stuffs I've done in a table to save time later'**

Programming= planning/table filling

Dynamic= multi-stage, walking-around

Example: DP solution for fib(n):

- build table/array F[1..n], *in the bottom-up order*, starting from F[1]=1, F[2]=1, then F[3]= F[2]+F[1], F[4]= F[3]+F[2], …
- Compared with the trivial recursive solution: more efficient in time (but with more memory)
- Note that for this task, we don't actually need to keep the whole array F. It's sufficed just to keep the last 2 elements.

Dynamic Programming

Lessons from Fibonacci and Knapsack Problems:

Normally, in DP we *solve the small-size sub-problems first*, then progressively solve the bigger size. Each time, when a sub-problem is solved, its solution is stored for being used later.

The most important steps in DP are to work out:

- the **problem parameters** (such as n in fib(n)), and
- the **relationship** between the solution for larger and for smaller parameters (such as fib(n)= fib(n-1)+fib(n-2))
- solve small size sub-problems first!

Often, solving small-size subproblems helps in building the relationship between small and large sizes.

DP: Examples

- Fibonacci: find fib(n)
- Warshall's algorithm: Find Transitive Closure of a digraph
- Floyd's Algorithm: all pair shortest paths for a weighted graph
- Baked Bean Bundles (problem 4)

Problem 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 bought n cans of baked beans, n=8.

We have the following things as the start:

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

What is the parameter of the task? What's base case? What's the DP relationship?

Notes that in general:

- the total number of cans n may differ from the maximal bundle size
- bundle size might be any positive, and not consecutive integers like in this task

What are the sub-problems?

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

- We want to have V[w] = best value (in $) we can get if we have w cans
- Let P[k] be the price of a bundle with k cans
- We have two parameters that we could use: *n*, the number of bundles, and *W=8*, the number of cans we have
- Let's try *W:* Let *V*(*w*) be the maximum value we can obtain given the number of cans *w*
  - We want *V*(*W*)
  - Assume we already know *V*(*w*) for all *w* < *W*
  - How can we use the *W* - *w* spare cans?

$V(w) =$ $P[i]$ + $V(w - i)$

Q: If we know V(i) for all i<w, then: V(w)= ?

What is the order?

$V(w) = \max_i\{ V(w - i) + P[i] \}$

$V(0) = 0$

- Clearly order is 0, 1, 2, 3, …, W
- Make a table and fill it in

V[0] = 0
for w in 1 to *W*
  V[w] = max{ P[i] + V[w – i] } for i∈1..*n (*or just *1..w)*

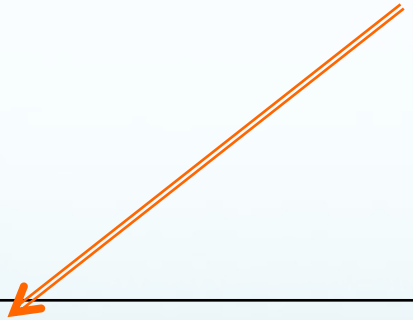
Example: W=8

i	0	1	2	3	4	5	6	7	8
size	0	1	2	3	4	5	6	7	8
price P[i]	$0	$1	$5	$8	$9	$10	$17	$17	$20

$$V(w) = \max_i \{ P[i] + V(w - i) \}$$

$$V(0) = 0$$

$$V(1) = \max\{1 + V(0)\}$$



w=0	1	2	3	4	5	6	7	8
0	1							

Example: W=8

i	0	1	2	3	4	5	6	7	8
size	0	1	2	3	4	5	6	7	8
price P[i]	$0	$1	$5	$8	$9	$10	$17	$17	$20

$V(w) = \max_i\{ P[i] + V(w - i) \}$

$V(0) = 0$

$V(2) = \max\{5+V(0), 1+V(1)\}$

-justify the correctness of V(2)!

w=0	1	2	3	4	5	6	7	8
0	1	5						

Example: W=8

i	0	1	2	3	4	5	6	7	8
size	0	1	2	3	4	5	6	7	8
price P[i]	$0	$1	$5	$8	$9	$10	$17	$17	$20

$V(w) = \max_i\{ P[i] + V(w - i) \}$

$V(0) = 0$

$V(3) = \max\{8+V(0), 5+V(1), 1+V(2)\}$

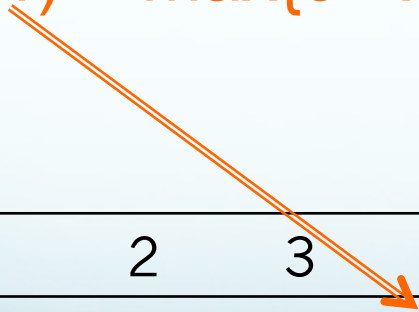
Example: W=8

i	0	1	2	3	4	5	6	7	8
size	0	1	2	3	4	5	6	7	8
price P[i]	$0	$1	$5	$8	$9	$10	$17	$17	$20

$V(w) = \max_i\{ P[i] + V(w - i) \}$

$V(0) = 0$

$V(4) = \max\{9+V(0), 8+V(1), 5+V(2), 1+V(3)\}$



w=0	1	2	3	4	5	6	7	8
0	1	5	8	10	?			

But late, how do we know which bundles used in V(4)?

Example: W=8

bundle i	0	1	2	3	4	5	6	7	8
size	0	1	2	3	4	5	6	7	8
price P[i]	$0	$1	$5	$8	$9	$10	$17	$17	$20

w	0	1	2	3	4	5	6	7	8
V(w)	0	1	5	8	10				
max i	0	1	2	3	2				

$V(w) = \max_i\{ P[i] + V(w - i) \}$

$V(0) = 0$

$V(4) = 10$

But which bundles used in V(4) → v(2) + V(4-2) →
v(2)+V(2) → v(2) + v(2)

Baked Beans Bundles

(b) **Run the algorithm manually**

(c) **Complexity = ?**

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

Run the algorithm:

cans w	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
revenue V(w)	0								
bundle B	0								

Running time & space:

Baked Beans Bundles

(b) **Run the algorithm manually**

(c) **Complexity = ?**

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

Run the algorithm:

cans w	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
revenue V(w)	0								
bundle B	0								

Running time & space:

- O(nw) time → pseudo-polynomial wrt. w
- **Note:** in the task, w=n, and hence $O(n^2)$
- O(w) space

What's pseudo-linear, pseudo-polynomial?

Bean Bundles: Pseudocode?

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

cans w	0	1	2	3	4	5	6	7	8
revenue V(w)	0								
bundle B	0								

function BakedBean(prices[1..n])

```
set additional arrays: V[0..n], B[0..n] with initial values
for w ← 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
    V[w] ← max_{i∈1..w} (prices[i] + V[w-i])
    B[w] ← index i of the above max
output V[n]     # print max value with n cans
# the print the used bundles, need to make a loop for that
...
```

Bean Bundles: Pseudocode?

function BakedBean(prices[1..n])

```
#set additional arrays: V[0..n], B[0..n] with initial values
V[0..n] ← {0,...,0}
B[0..n] ← {0,...,0}
for w ← 1 to n  do # computing max
    maxval←0, maxi ← 0
    for i← 1 to w do
        if V[w-i]+prices[i] > maxval then
            maxval← V[w-i]+prices[i]
            maxi= i
    V[w] ← maxval
    B[w] ← maxi
output V[n]     # print max value with n cans
# the print the used bundles, need to make a loop for that
i ← n
while B[i] >0 do
    output B[i]
    i ← i – B[i]
```

Homework

Can we use the same technique for the case W=10 and:

bundle i		**1**	**2**	**3**	**4**
bundle size w[i]		6	3	4	2
price P[i]		$30	$14	$16	$9

Homework

Can we use the same technique for the case n=4, W=10 and:

bundle i	**0**	**1**	**2**	**3**	**4**
bundle size w[i]		6	3	4	2
price P[i]/value v[i]		$30	$14	$16	$9

V(0)=0

V(w)= ??

w	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
V(w)	0	0	9	14	18						
max i (which bundle)			4	2	4						

Bean Bundles vs. Knapsack?

Are the tasks similar?

Baked Beans
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?

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. A same bundle can be re-used! ➔ with repetition	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). an item can be used only 0 or 1 time ➔ no repetition

Knapsack more complicated = No-repetition seems more difficult because:

Now $V(w) = \max_i\{v_i + V(w - w_i)\}$ is useless: how do we know what items are already in $V(w - w_i)$?

Bean Bundles vs. Knapsack?

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. A same bundle can be re-used! ➔ with repetition	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). an item can be used only 0 or 1 time ➔ without repetition

Knapsack more complicated = No-repetition seems more difficult because:

Now $V(w) = \max_i\{v_i + V(w - w_i)\}$ is useless: how do we know what items are already in $V(w - w_i)$?

Knapsack: without repetition

- Now

$V(w) = \max_i\{ V(w - w_i) + v_i \}$

is useless: how do we know what items are already in $V(w - w_i)$?

- Need another parameter…

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).

an item can be used only 0 or 1 time
➔ without repetition

Knapsack: without repetition

- Now $V(w) = \max_i\{ V(w - w_i) + v_i \}$ is useless: how do we know what items are already in $V(w - w_i)$?
- Need another parameter…
- Let $K(w, j)$ be the highest value with a knapsack of capacity w only using items 1..j

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).

an item can be used only 0 or 1 time
➔ without repetition

$$K(w, j) = \max \{ K(w - w_j, j\text{-}1) + v_j, K(w, j\text{-}1) \}$$

$$K(w, j) = 0, j < 1 \text{ or } w < 1$$

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

$K(w, j) = \max \{ K(w - w_j, j\text{-}1) + v_j, K(w, j\text{-}1) \}$

$K(w, j) = 0, j < 1$ or $w < 1$

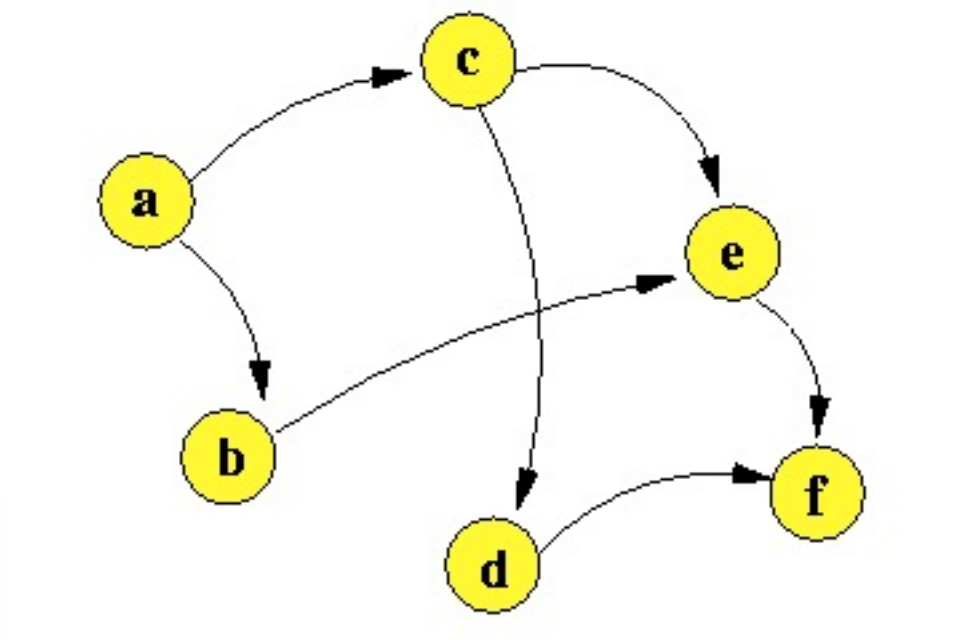
$j \backslash w$		1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0					
2	0										
3	0										
4	0										

Watch lecture Week11.Lecture2.Part3 for example on running the Knapsack

Transitive Closure of digraphs

Understanding

Transitive Closure



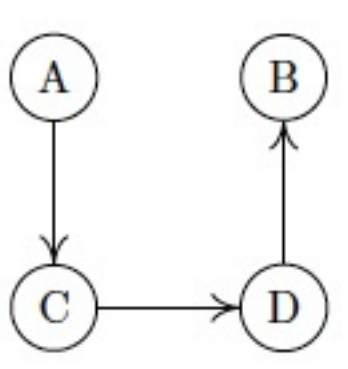
Related Tasks:

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

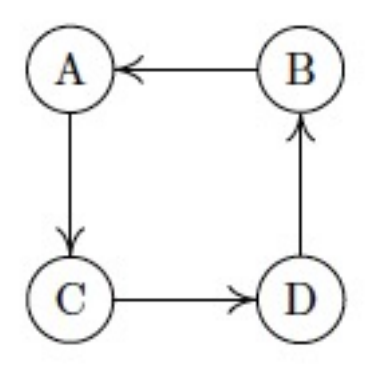
Problem 1: Transitive Closure of digraphs

Draw the transitive closure of the following two graphs:

(a)



(b)



Warshall's Algorithm: DP for Transitive Closure

- Input: adjacent matrix A
- Main argument: transitiveness: if there are paths i→k and k→j, then there is path i→j which uses k as an interim stepstone.

- Recall: To do DP, we need to decide:
  - what are parameters: ???
  - what is the relationship between solutions for a bigger and a smaller parameter: ???
  - what are the base cases (when solution is ready)

Warshall's Algorithm: DP for Transitive Closure

Ri : all paths get from using nodes 1..i as interim stepstones

- adjacent matrix A is R^0
- we can go from R^0 to R^1 by $R^1_{ij} = R^0_{ij} \;||\; (R^0_{i1} \;\&\&\; R^0_{1j})$

- we can go from R^{k-1} to R^k by $R^k_{ij} = R^{k-1}_{ij} \;||\; (R^{k-1}_{ik} \;\&\&\; R^{k-1}_{kj})$

How to store R^k_{ij} for lookup? Do we really need an array of 2D arrays?

- Write the algorithm, by first set R= A, then progressing k from 1 to n.

Remember: this DP ajgorithnm is characterised by

$$R^0 = A$$

$$R^k_{ij} = R^{k-1}_{ij} \;||\; (R^{k-1}_{ik} \;\&\&\; R^{k-1}_{kj}) \text{ for } k = 1..n$$

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 can have shortest path for i→j just by joining the formers.
- So, the idea is just similar to the Warshall's. Can we build the DP relationship?

DP: Floyd's Algorithm (APSP)

Floyd's algorithm builds on Warshall's algorithm to solve the all pairs shortest path problem: computing the length of the shortest path between each pair of vertices in a graph.

Rather than an adjacency matrix, we will require a weights matrix W , where Wij indicates the weight of the edge from i to j (if there is no edge from i to j then then Wij = ∞). We will ultimately find a distance matrix D in which Dij indicates the cost of the shortest path from i to j.

The sub-problems in this case will be answering the following question: What's the shortest path from i to j using only nodes in {1, . . . , k} as intermediate nodes?

To perform the algorithm we find D^k for each k ∈ {0,...,n} and set D := D^n. The update rule becomes the following:

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

Problem 2: Running Warshall's Algorithm

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

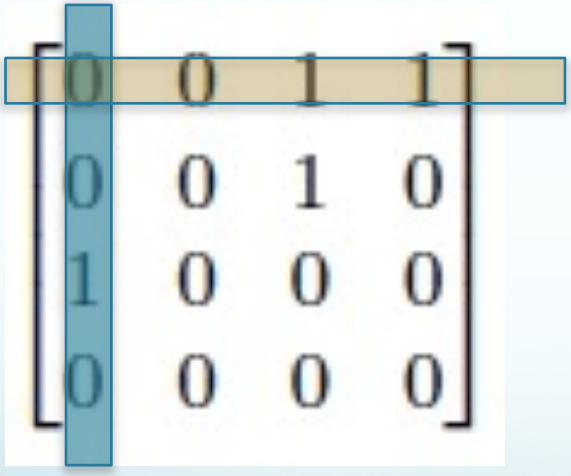
Set R= A, R is R^0

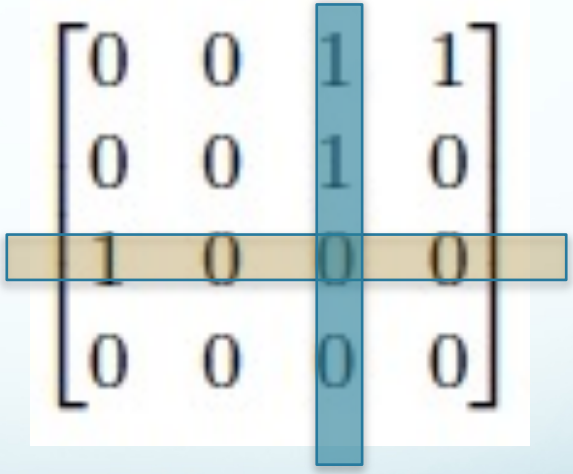
transition from 0 to 1 by:

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

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

similarly for any transition from k-1 to k





Notes: *Daniel demonstrated a detailed manual run of Warshall's in* Week11.Lecture1.Part 2

Problem 2: Check your answer

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

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

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

Problem 3: manual exec of Floyd's

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

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

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

Notes: *Daniel demonstrated a detailed manual run of Floyd's's in* Week11.Lecture1.Part 5

Example: $D_{k-1} \rightarrow D_k$ when k=1

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

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

row **k**:
$A_{kj} < \infty$ if there is a path **k**→j

column **k**:
$A_{ik} < \infty$ if there is a path i→**k**

Problem 3: Check your answer

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

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

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

Problem 5: Revision for quicksort & mergesort

(a) Perform a single Hoare Partition on the following array, taking the first element as the pivot.
[3, 8, 5, 2, 1, 3, 5, 4, 8]

(b) Perform Quicksort on the array from (a). You may use whatever partitioning strategy you like

(c) Perform Mergesort on the array from (a).

Lab

- Implement the baked bean bundles problem. You can use input.txt from LMS for the input data
- Notes:
  - the program will be short,
  - the solution is supplied in LMS, but:
  - try not to use it, and build your code from scratch first → a good way to understand & remember DP.