COMP20007 Workshop Week 12

1	Discussion: Dynamic Programming Question 12.4
2	Warshall's Algorithm: Questions 12.1 & 12.2 Floyd's Algorithm: Question 12.3
3	Revision with Q11.7 and Q12.5
LAB	A2: questions on pseudocode format baked bean bundles (implementation)

Dynamic Programming: great technique, fancy name

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

To compute f(W)

- first, compute and keep solutions for base cases: **F**[0]= f(0), **F**[1]= f(1), …
- then compute f(i) using **F**[j] (j<i) and store f[i] in F[i] for later use

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], … until getting to F[n] (which holds f(n))

- **Note:** we don't actually need to keep the whole array F. It's sufficed just to keep the last 2 elements.

DP: how to start

The most important steps in DP are to work out:

- the **problem and its 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))

How to work out the relationship?

- use logic and reasoning top-down f(n)= … OR/AND
- manually solve problems for small n=1, 2, 3 … and work out the pattern for f(n)

DP for optimization problems

- Optimization problem:
  - Having a function f, with possible different ways of getting value f(n) for a value n
  - Find a way that maximize (or minimize) the value f(n)
- DP possible if f(n) 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 12.4: 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

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 i	**0**	**1**	**2**	**3**	**…**	**n**
price p_i	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: f(n) → max, f(n) is revenue from selling n cans

  We will store f(i) in F[i] of array F[0..n]

- Recurrence: F[n] = max(p_i + revenue from selling n-i can)= max(p_i + F(n-i))
- Base case: F[0]=0, F[1]= p_1

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

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

```
set additional arrays: F[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
    F[w] ← max_{i∈1..w} (prices[i] + F[w-i])
    B[w] ← index i of the above max
output F[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]) #here W=n

```
#set arrays: F[0..n], B[0..n] with initial values
F[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 F[w-i]+prices[i] > maxval then
            maxval← F[w-i]+prices[i]
            maxi ← i
    F[w] ← maxval
    B[w] ← maxi
output F[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]
```

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 w	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
revenue F[w]	0								
bundle Bi, i=	0								

(c) Complexity = ?

Running time:

Space:

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 $F[i]= \max_{w_i \le i} (v_i + F(i - w_i))$

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?

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 at most 1 time ➔ no repetition

Knapsack more complicated = No-repetition seems more difficult!

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.

A same bundle can be re-used!
➔ with repetition

$F[w] = \max_i\{v_i + V[w - w_i]\}$
$F[0] = 0$

???

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).

an item can be used only 0 or 1 time
➔ no repetition

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

Knapsack: without repetition

- Now

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

is useless here.

- 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 $F[w] = \max_i\{ F[w - w_i] + v_i \}$ is useless.
- Need another parameter…for the used items
- 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).

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

$$K[i,w] = 0,\ i < 1 \text{ or } w < 1$$

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

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

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

$i \backslash w$		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

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.

What's the output looks like?

Parameters:
Recurrence:
Base case:

Some other problems

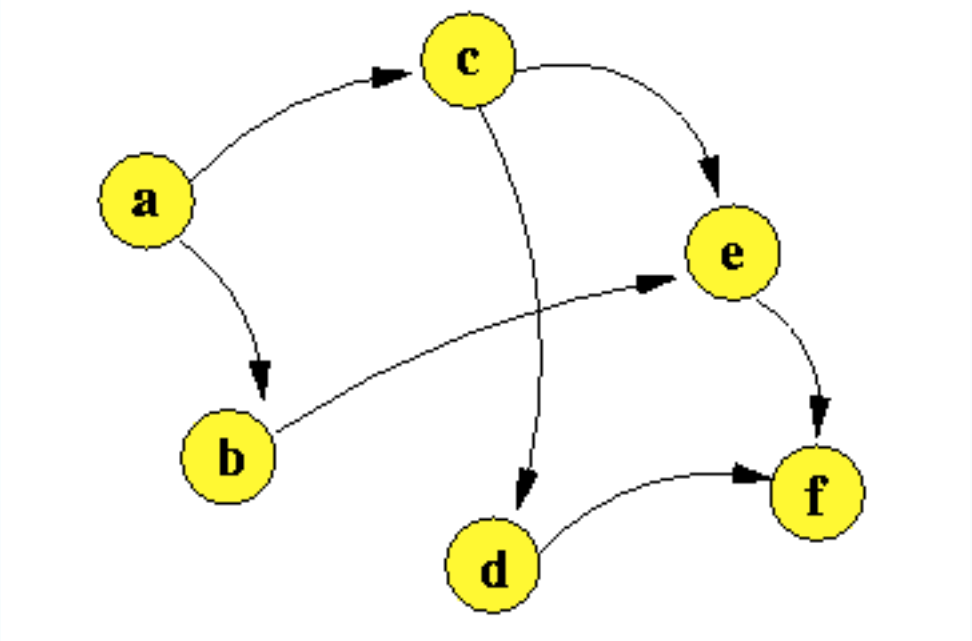
- Coin-collecting problem Levitin.8.1.example3
- Order of matrix multiplication: exercise L8.3.11
- Optimal binary search tree L8.3
- ...

DP is typically applied to optimization problems such as in 3 of the above 4 boxes, knapsack, bean bundles. Exception: fibonacci, Warshall, Floyd, all palindromes...

Transitive Closure of digraphs

Understanding

Transitive Closure



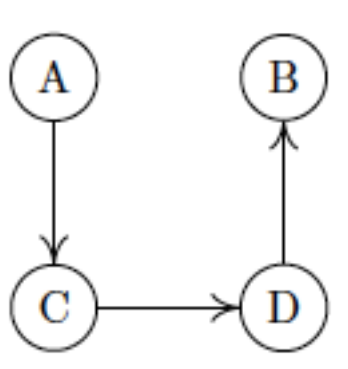
Related Tasks:

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

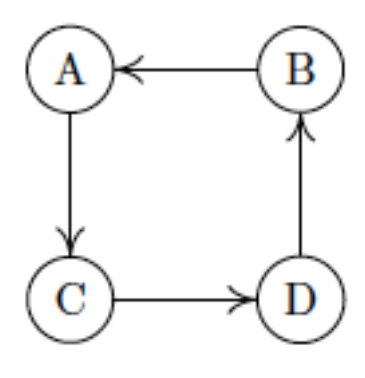
Q12.1: Transitive Closure of digraphs

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:

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

Q 12.2: Running Warshall's Algorithm

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

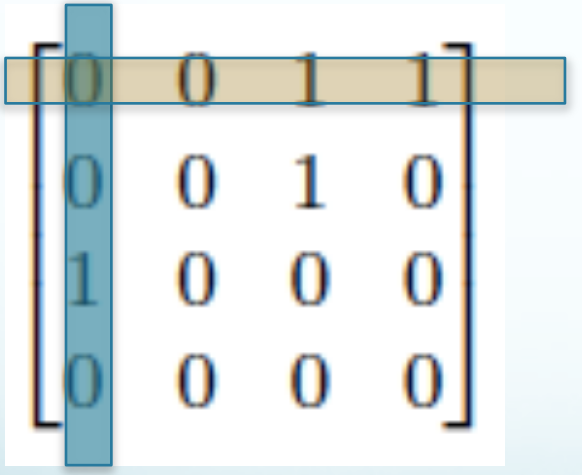
Set R= A, R is R^0

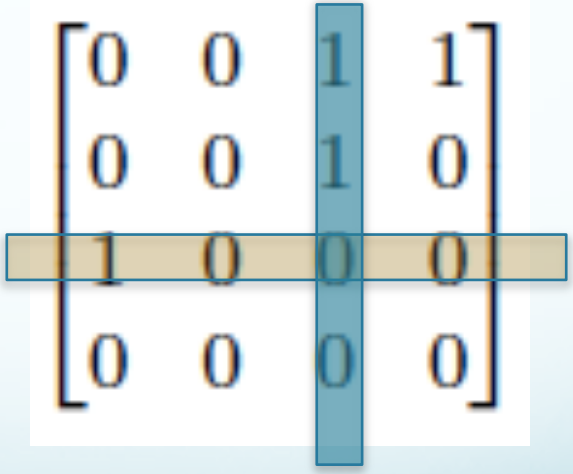
transition from 0 to 1 by:

- 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

Q12.2: Check your answer

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

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

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

Q12.3: manual exec of Floyd's

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

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

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

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**

Q 12.3: Check your answer

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

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

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

Revision

Q 11.7: Solve the following recurrence relations. Give both a closed form expression in terms of n and a Big-Theta bound.

(a) T(n)= T(n/2)+1, T(1)= 1

(b) T(n)= T(n-1) + n/5, T(0)= 0

Q12.5: 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.
- 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.