

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 | Q12.5: Revision: quicksort & top-down mergesort |
| L
A
B | 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 $f(W)$:

- start with computing and keeping $F[0]= f(0), F[1]= f(1), \dots$
- then fill in $F[i]= f(i)$ using $\{F[j] \mid j < i\}$

Example: DP solution for $\text{fib}(n)$:

- build table/array $F[1..n]$, *in the bottom-up order*, starting from $F[1]=1, F[2]=1$, then $F[3]= F[2]+F[1], F[4]= F[3]+F[2], \dots$ until getting to $F[n]$ (which is $f(n)$)
- DP versus the recursive solution: better in time (but with more memory?)
- **Note:** we don't actually need to keep the whole array F . It's sufficed just to keep the last 2 elements.

Dynamic Programming

DP is especially good for some optimization problems (see textbook for the conditions)

When DP?

- having recurrence: a task $T(n)$ is expressed as a few same tasks of $T(n_i)$ of smaller size ($n_i < n$)
- having overlap: $T(j)$ might be used for >1 times

How? normally using bottom-up (but read Levitin for minor top-down cases)

- *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.

DP: how to start

The most important steps in DP are to work out:

- the **problem parameters** (such as n in $\text{fib}(n)$), and
- the **relationship** between the solution for larger and for smaller parameters (such as $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$)

Sometimes, solving small-size subproblems manually helps in building the relationship between small and large sizes: *use both top-down and bottom-up reasoning when building the relationship.*

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 $k \in \{1, \dots, n\}$.

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

- Write the pseudocode for such an algorithm.
- 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

- 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 $W=8$ 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

What is the aim and parameter of the task? What's the DP recurrence and base cases?

Aim and Parameter:

Recurrence:

Base case:

Notes that in general:

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

Baked Beans Bundles

(b) Run the algorithm manually

(c) Complexity = ?

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

Run the algorithm:

cans w	0	1	2	3	4	5	6	7	8
revenue $V(w)$	0								
bundle $B_i, i=$	0								

Running time & space:

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

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

sketch for 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] ← maxi∈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: Your Pseudocode

```
function BakedBean(prices[1..n]) #here W=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] ← maxi∈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 vs. Knapsack?

Are the tasks similar?

Bean Bundles

Given n bundles with

- cans/weights: $w_1; w_2; \dots; w_n$
- price/values : $v_1; v_2; \dots; v_n$

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: $w_1; w_2; \dots; w_n$
- values: $v_1; v_2; \dots; v_n$

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

Given n bundles with

- cans/weights: $w_1; w_2; \dots; w_n$
- price/values : $v_1; v_2; \dots; v_n$

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

Knapsack

Given n items with

- weights: $w_1; w_2; \dots; w_n$
- values: $v_1; v_2; \dots; v_n$

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: $w_1; w_2; \dots; w_n$
- price/values : $v_1; v_2; \dots; v_n$

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

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

$$V[0] = 0$$

???

Knapsack

Given n items with

- weights: $w_1; w_2; \dots; w_n$
- values: $v_1; v_2; \dots; v_n$

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

Knapsack: without repetition

- Now

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

is useless here.

- Need another parameter...

Knapsack

Given n items with

- weights: $w_1; w_2; \dots; w_n$
- values: $v_1; v_2; \dots; v_n$

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.
- 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

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

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

$\setminus w$	1	2	3	4	5	6	7	8	9	10
	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, \dots, 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 $d_1 < d_2 < \dots < d_n$. 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

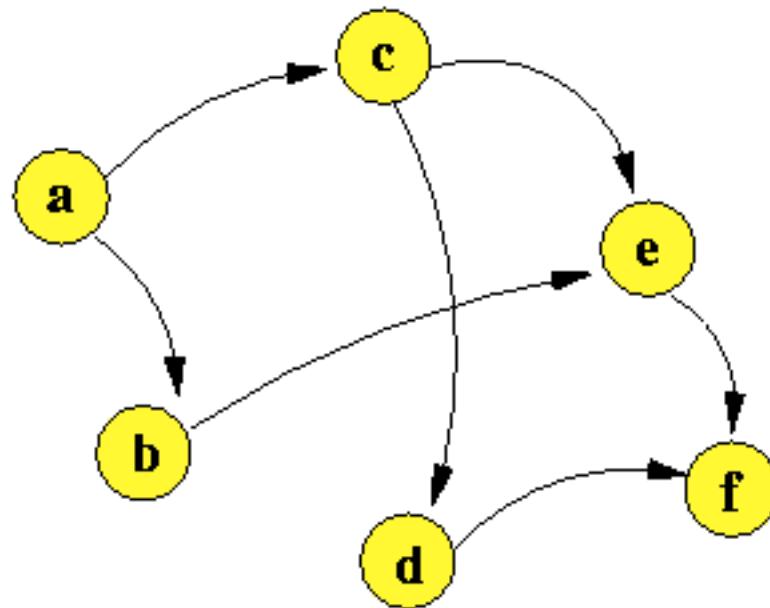
- Coin-collecting problem
Levitin.8.1.example3
- Order of matrix multiplication:
exercise L8.3.11
- Optimal binary search tree L8.3
- ...

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



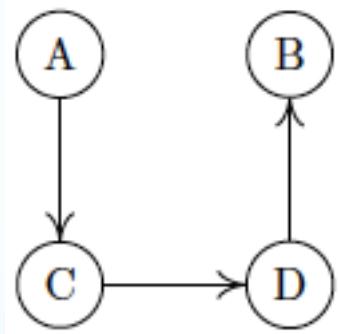
Related Tasks:

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

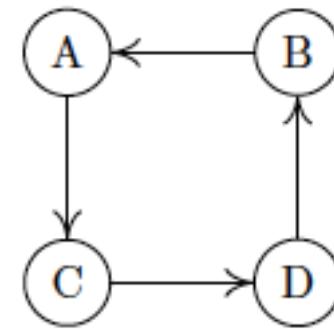
Q12.1: Transitive Closure of digraphs

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 $i \rightarrow k$ and $k \rightarrow j$, then there is path $i \rightarrow j$ 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

R_i : 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 $R^1_{ij} = R^0_{ij} \parallel (R^0_{i1} \ \&\& \ R^0_{1j})$
- we can go from R^{k-1} to R^k by $R^k_{ij} = R^{k-1}_{ij} \parallel (R^{k-1}_{ik} \ \&\& \ R^{k-1}_{kj})$

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 algorithm is characterised by

$$R^0 = A$$

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

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 $i \rightarrow k$ and for $k \rightarrow j$ then we can have shortest path for $i \rightarrow j$ 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 W_{ij} indicates the weight of the edge from i to j (if there is no edge from i to j then $W_{ij} = \infty$). We will ultimately find a distance matrix D in which D_{ij} 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 $\{1, \dots, k\}$ as intermediate nodes?

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

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

Q 12.2: Running Warshall's Algorithm

$$R_{ij}^0 := A_{ij}, \quad R_{ij}^k := R_{ij}^{k-1} \text{ or } (R_{ik}^{k-1} \text{ and } R_{kj}^{k-1})$$

Set $R = A$, R is R^0

transition from 0 to 1 by:

- looking at 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

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

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

Q12.2: your answer

$$R_{ij}^0 := A_{ij}, \quad R_{ij}^k := R_{ij}^{k-1} \text{ or } \left(R_{ik}^{k-1} \text{ and } R_{kj}^{k-1} \right)$$

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

Q12.3: manual exec of Floyd's

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

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

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

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

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

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

$W =$

0	3	∞	4
∞	0	5	∞
2	∞	0	∞
∞	∞	1	0

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

.

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

Q 12.3: your answer

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

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

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.