COMP20007 Workshop Week 9: Dynamic Programming

1	**Dynamic Programming** 1. Review: DP technique, Backtrace 2. Group Exercise: Q9.2 , then revisit the knapsack problem 3. Class exercise: Matrix Chain Multiplication 4. Group Exercise: 9.3 Longest Common Substring
2	**Graph Algorithms** (time permits): 9.1 Dijkstra Algorithm and Negative weight 9.4 Milk Pouring Problem and Implicit Graphs
LAB	W9.3: Implement 9.2

Have your "paper & pen" ready please!

Dynamic Programming: fascinating technique, fancy name

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

Programming= planning/table filling

Dynamic= multi-stage, walking-around

The Technique:

- form a table DP[state] where state identifies a subproblem
- fill in the table, normally bottom-up, until getting the final state when state= original problem

What kind of problems can be solved with DP?

problems that

- can be divided into smaller sub-problems?
- have overlapping sub-problems?
- are recursive in nature?

Notes:

- Unlike the Fibonacci example, DP is typically applied for optimization problems.

- 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: example of the basic technique

The most important & difficult steps in DP are to work out:

- 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
  - 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

1
1
2
3
5

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]
```

DP for optimization (e.g. finding the best solution) tasks

Basic Steps

I. Understand the Problem then Determine:
- Parameters of subproblem (state)
- DP[state] = the value of the optimal solution

II. Formulate the Recurrence Relation:
- Define the rule that expresses the solution to a larger subproblem in terms of solutions to smaller subproblems.
- Base Cases: Determine the simplest subproblems whose solutions can be directly computed.

III. Determine the Order of Computation
- Decide whether to use a bottom-up (tabulation) or top-down (memoization) approach.

IV. Find actual solution (not just the optimal value)
- Devise a method to backtrack through the DP table
- If needed, store decisions made to reconstruct the optimal sequence of choices.

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: choose non-adjacent coins for maximal sum
State Parameters & DP:
Recurrence:
Base case:

Order of computation:
Backtrace to find the optimal solution:

Example:
C= {5, 1, 2, 10, 6, 2}

DP for optimization (e.g. finding the best solution) tasks

Basic Steps

I. Understand the Problem then Determine:
- Parameters of subproblem (state)
- DP[state] = the value of the optimal solution

II. Formulate the Recurrence Relation:
- Define the rule that expresses the solution to a larger subproblem in terms of solutions to smaller subproblems.
- Base Cases: Determine the simplest subproblems whose solutions can be directly computed.

III. Determine the Order of Computation
- Decide whether to use a bottom-up (tabulation) or top-down (memoization) approach.

IV. Find actual solution (not just the optimal value)
- Devise a method to backtrack through the DP table
- If needed, store decisions made to reconstruct the optimal sequence of choices.

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: choose non-adjacent coins for maximal sum

State Parameters & DP:

k: number of coins in the row $c_1, c_2, \ldots, c_k$

F[k]= optimal outcome

= maximum amount that can be picked up from the row of k coins

Recurrence:

Base cases:

Order of computation:

Backtrace to find the optimal solution:

Example:

C= {5, 1, 2, 10, 6, 2}

DP for optimization (e.g. finding the best solution) tasks

Basic Steps

I. Understand the Problem then Determine:
- Parameters of subproblem (state)
- DP[state] = the value of the optimal solution

II. Formulate the Recurrence Relation:
- Define the rule that expresses the solution to a larger subproblem in terms of solutions to smaller subproblems.
- Base Cases: Determine the simplest subproblems whose solutions can be directly computed.

III. Determine the Order of Computation
- Decide whether to use a bottom-up (tabulation) or top-down (memoization) approach.

IV. Find actual solution (not just the optimal value)
- Devise a method to backtrack through the DP table
- If needed, store decisions made to reconstruct the optimal sequence of choices.

Change-making problem

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

Problem:
State Parameters & DP:
Recurrence:
Base case:

Order of computation:
Backtrace to find the optimal solution:

DP for optimization (e.g. finding the best solution) tasks

Basic Steps

I. Understand the Problem then Determine:
- Parameters of subproblem (state)
- DP[state] = the value of the optimal solution

II. Formulate the Recurrence Relation:
- Define the rule that expresses the solution to a larger subproblem in terms of solutions to smaller subproblems.
- Base Cases: Determine the simplest subproblems whose solutions can be directly computed.

III. Determine the Order of Computation
- Decide whether to use a bottom-up (tabulation) or top-down (memoization) approach.

IV. Find actual solution (not just the optimal value)
- Devise a method to backtrack through the DP table
- If needed, store decisions made to reconstruct the optimal sequence of choices.

Change-making problem

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

Problem: minimizing number of coins for the amount n

State Parameters & DP: amount k, DP[k]= min number of coins for k

Recurrence: $F[k]= \min_{k \geq d_j} \{ F[k-d_j] + 1 \}$

Base case: F[0]= 0

Order of computation: bottom-up from k=1 to n

Backtrace to find the optimal solution:

Example: n=6, D= {1, 3, 4}

Backtrace Notes

DP works out the optimal value only. To find the optimal solution, we need to backtrack in the DP table, start from the optimal solution. Two cases:

- backtrace is possible just by using the DP table, like in the coin-row problem
- backtrace needs to know the decision made in each step → while building DP table, we need to store the decisions in a parallel table, like in the change-making problem

[Group/Individual] Question 9.2: 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 $k \in \{1,\ldots,n\}$.

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?

Basic Steps

I. Understand the Problem then Determine:
- Parameters of subproblem (state)
- DP[state] = the value of the optimal solution

II. Formulate the Recurrence Relation:
- Define the rule that expresses the solution to a larger subproblem in terms of solutions to smaller subproblems.
- Base Cases: Determine the simplest subproblems whose solutions can be directly computed.

III. Determine the Order of Computation
- Decide whether to use a bottom-up (tabulation) or top-down (memoization) approach.

IV. Find actual solution (not just the optimal value)
- Devise a method to backtrack through the DP table
- If needed, store decisions made to reconstruct the optimal sequence of choices.

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..i

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

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

(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]									
bundle B[i]									

revenue= ?

bundles used= ?

c) Complexity = ?

Running time:

Space:

Baked Beans Bundles: check your solution

(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

revenue= 22, bundles: 2 (B[8]=2), then 6 (B[8-2]=6)

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:

Bean Bundle Problem	**Generalized Bean Bundle Problem**
Given n bundles with • cans/weights in bundles: 1, 2, ..., n • price/values of bundles : $v_1; v_2;\dots;v_n$ And given amount of cans n find how we should split up out n cans into bundles to maximize the total revenue we can receive.	Given n bundles with • cans/weights in bundles: $w_1;w_2;\dots;w_n$ • price/values of bundles : $v_1; v_2;\dots;v_n$ And given amount of cans W find how we should split up out W cans into bundles to maximize the total revenue we can receive.
Solution is similar: $F[0]= 0$ $F[i]= \max_{j \leq i} (v_j + F(i - j))$ Solution = F[n]	Solution is similar: $F[0]= 0$ $F[i]= \max_{w_j \leq i} (v_j + F(i - w_j))$ Solution = F[W]

Bean Bundles vs. Knapsack?

Are the tasks similar?

Bean Bundles	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).

Knapsack solution looks more complicated:

- What's exactly "more complicated"
- Why?

Bean Bundles vs. Knapsack?

Are the tasks similar?

Bean Bundles	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. ➔ single variable W $F[w] = \max_i\{v_i + V[w - w_i]\}$ $F[0] = 0$	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 $F[w] = \max_i\{v_i + F[w - w_{i]}\}$ 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) (number of items remains unchanged) • 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).

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

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

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

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

$i \backslash w$		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 for example on running the Knapsack

Class Exercise: Matrix Chain Multiplication

Problem: Find the most efficient way to do the multiplications

$$A_1 * A_2 * A_3 * \ldots * A_n$$

where A_i is a matrix

Understanding:

- What is a matrix multiplication?
- Can DP help with multiplication of 2 matrices?
- How about the multiplication of ≥ 3 matrices

What's next:

- Make the matrix chain multiplication clearer for efficiency analysis
- Define an DP task for optimisation
- Do steps 1 and 2: determine parameters, sub-problems, recurrences

[Group/Individual] Problem 9.3: Longest Common Substring

The Task:

- Input: 2 strings: a[0..n-1] of length n and b[0..m-1] of length m
- Output: a longest string that appears in both a and b
- Example: a= “UNDERSIGN”, b= “DESIGN” → soln= “SIGN”
- Notes:
  - do not confuse with *finding a longest common subsequence* where soln is “DESIGN”, where “subsequence” has no requirement of being contiguous as in substring

Method 1: exhaustive search, discuss:

- what are all possible substrings?
- how to check if a substring is common, and how to find the longest one?

Problem 9.2.Method1.soln: Longest Common Substring – exhaustive soln

The Task: a[0..n-1], b[0..m-1] → LCS of a, b

Example: a= "UNDERSIGN", b= "DESIGN" → "SIGN"

a) [very bad exhaustive search]

- a has $n(n+1)/2$, b has $m(m+1)/2$ substrings
- we can compare them pair-wise to find the common, and hence the LCS,
- complexity: $O(m^2n^2m)=O(n^2m^3)$!

b) [exhaustive, but better]

- only consider the starting point of a substring in b, we have m starting points,
- for each such point i, do exhaustive pattern matching of b[i..m-1] in a, keep track of the longest match

  → total time complexity= $O(m^2n)$
- Worst case: when all character comparisons are matched (well, when all characters in both strings are just the same like AAAAAA and AAA) → $\theta(m^2n)$

DP solution for the LCS problem

Basic Steps
I. Understand the Problem then Determine: • Parameters of subproblem (state) • DP[state] = the value of the optimal solution **II. Formulate the Recurrence Relation:** • Define the rule that expresses the solution to a larger subproblem in terms of solutions to smaller subproblems. • Base Cases: Determine the simplest subproblems whose solutions can be directly computed. **III. Determine the Order of Computation** • Decide whether to use a bottom-up (tabulation) or top-down (memoization) approach. **IV. Find actual solution** (not just the optimal value) • Devise a method to backtrack through the DP table • If needed, store decisions made to reconstruct the optimal sequence of choices.

Basic Steps
I. value to optimize= ? paramaters for a state/subproblem state= DP[state]= **II. Recurrence Relation:**

DP for LCS: initial thoughts

Store DP(i,j) in a table T →

- T[i][j] = DP(i,j)
- and T[n-1][m-1] should give solution? or what?

T[i][j]= something from comparing a_i with b_j

= 0 if $a_i \neq b_j$

= ? if = $a_i = b_j$

perhaps use a simple example "ABC" "XBCD"

we knew the soln is "BC"

	X	B	C	D
A	0	0	0	0
B	0	?	0	0
C	0	0	?	0

Base cases: perhaps when i=0 or j=0 ?

Table: 2D, row i for a_i, with i= 0..n-1

col j for b_j, with j= 0..m-1

DP-soln: longest common substrings (LCS)

Homework:
- Complexity= ?
- Pseudocode

T[i][j]= something from comparing a_i with b_j

= 0 if $a_i \neq b_j$

= if $a_i = b_j$

len of LCS ended at (i in a,j in b)

T[i-1][j-1] + 1

Backtrace (for soln)

find the largest value (4) in the table and take the diagonal

		D	E	S	I	G	N
	i j	0	1	2	3	4	5
U	0	0	0	0	0	0	0
N	1	0					
D	2	1					
E	3	0					
R	4	0					
S	5	0					
I	6	0					
G	7	0					
N	8	0					

Base cases: i=0 OR j=0

General case: building up the table for
(i= 1..n-1,
j= 1..m-1):
= 0 or
= T[i-1][j-1]+1
depending on
$a_i \neq b_j$ or not

		D	E	S	I	G	N
	i j	0	1	2	3	4	5
U	0	0	0	0	0	0	0
N	1	0	0	0	0	0	1
D	2	1	0	0	0	0	0
E	3	0	2	0	0	0	0
R	4	0	0	0	0	0	0
S	5	0	0	1	0	0	0
I	6	0	0	0	2	0	0
G	7	0	0	0	0	3	0
N	8	0	0	0	0	0	4

Class Exercise: Quick Discussion on Question 9.5

Part 2: Quick Discussions on some Graph Algorithms

9.4: The Milk Pouring Problem and Implicit Graphs

9.1: Dijkstra’s Algorithm and Negative Weight

The Milk Pouring Problem & Implicit Graphs

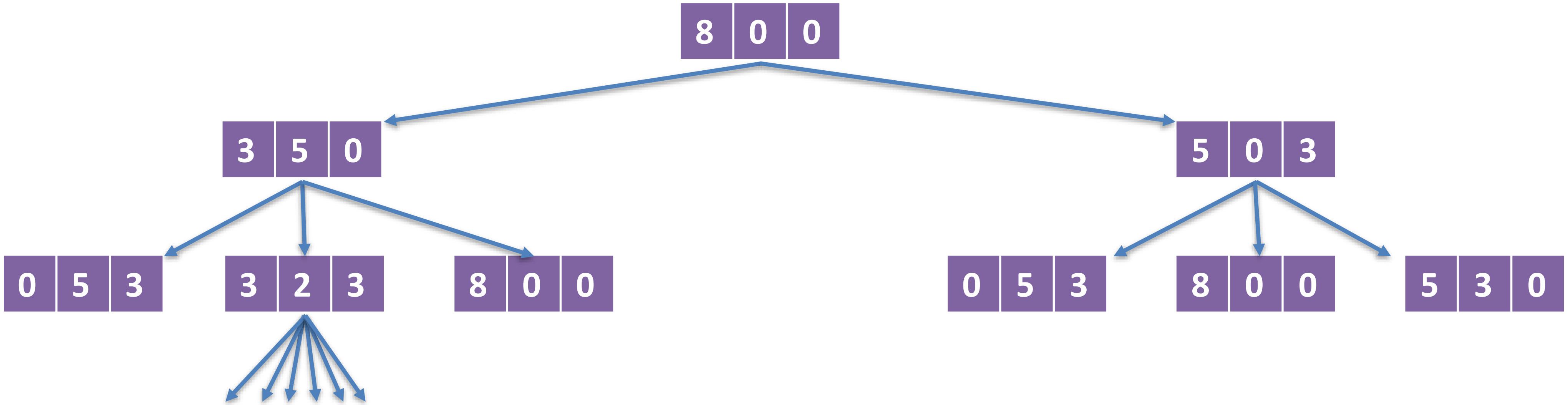
You are provided with three cans of varying capacities, denoted as A, B, and C. The objective is to measure precisely one litre of milk using these three cans, given that:

- Each can has an integer capacity greater than 1.
- Initially, the first can is filled with milk while the others remain empty.
- The only permissible action is to pour milk from one can to another until the source can is empty or the destination can is full.

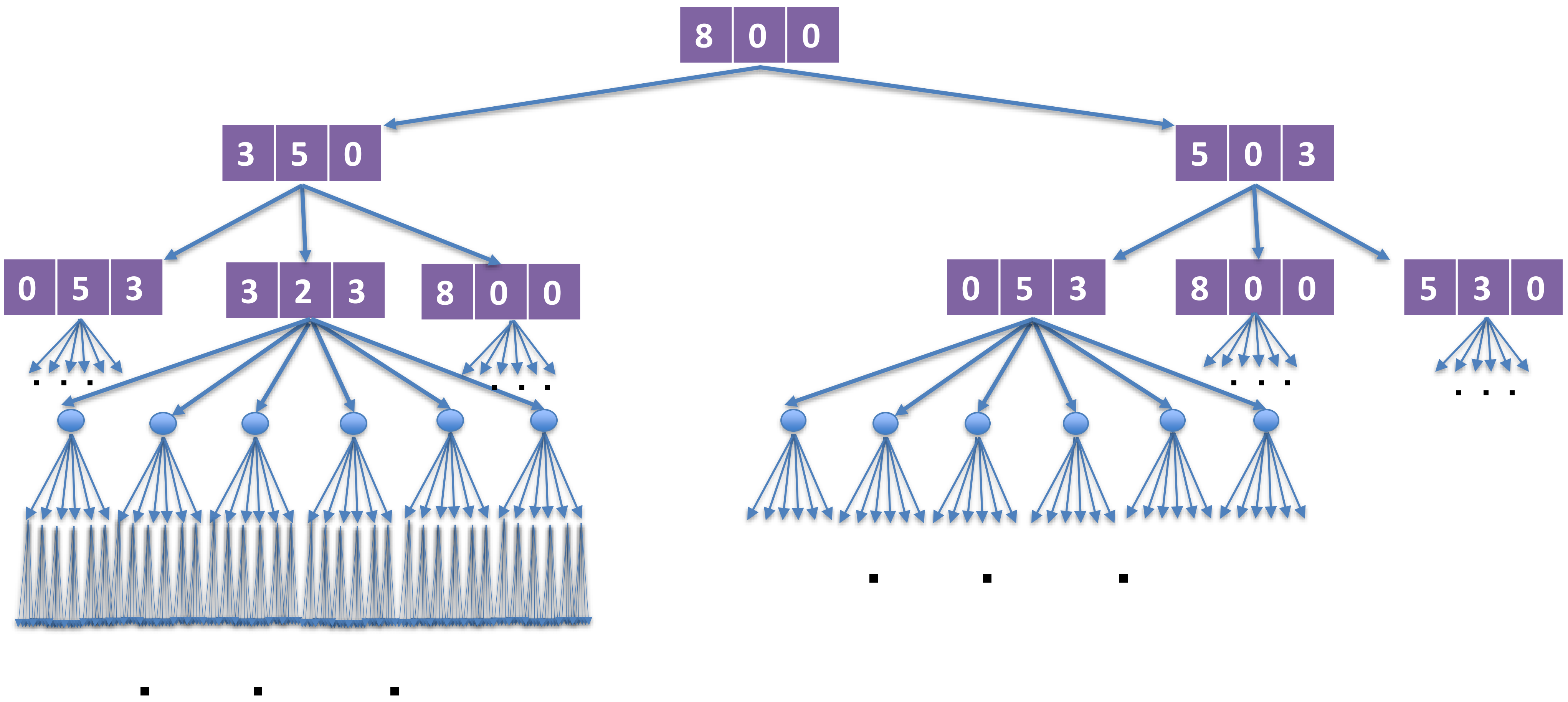
One classical example of this task involves given capacities for cans A, B, and C, with values of 8, 5, and 3, respectively.

How to solve the problem algorithmically?

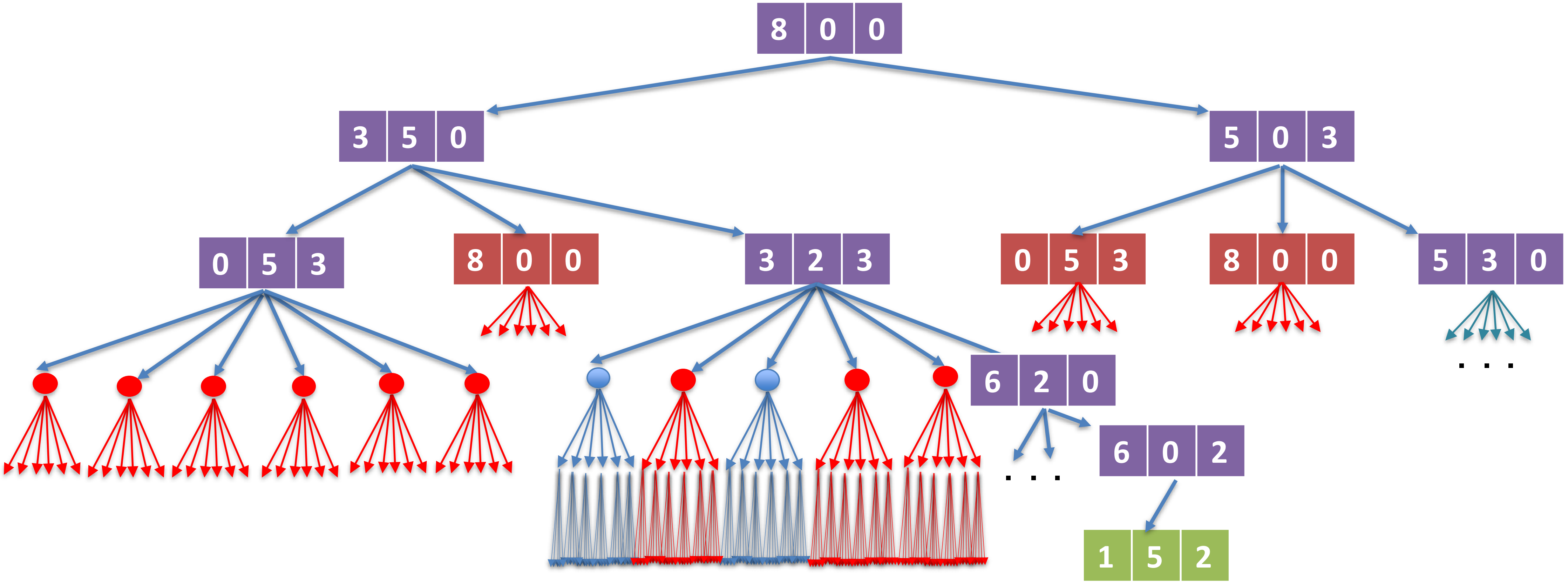
milk pouring A=8, B=5, C=3



milk pouring A=8, B=5, C=3



milk pouring A=8, B=5, C=3: optimization is important!



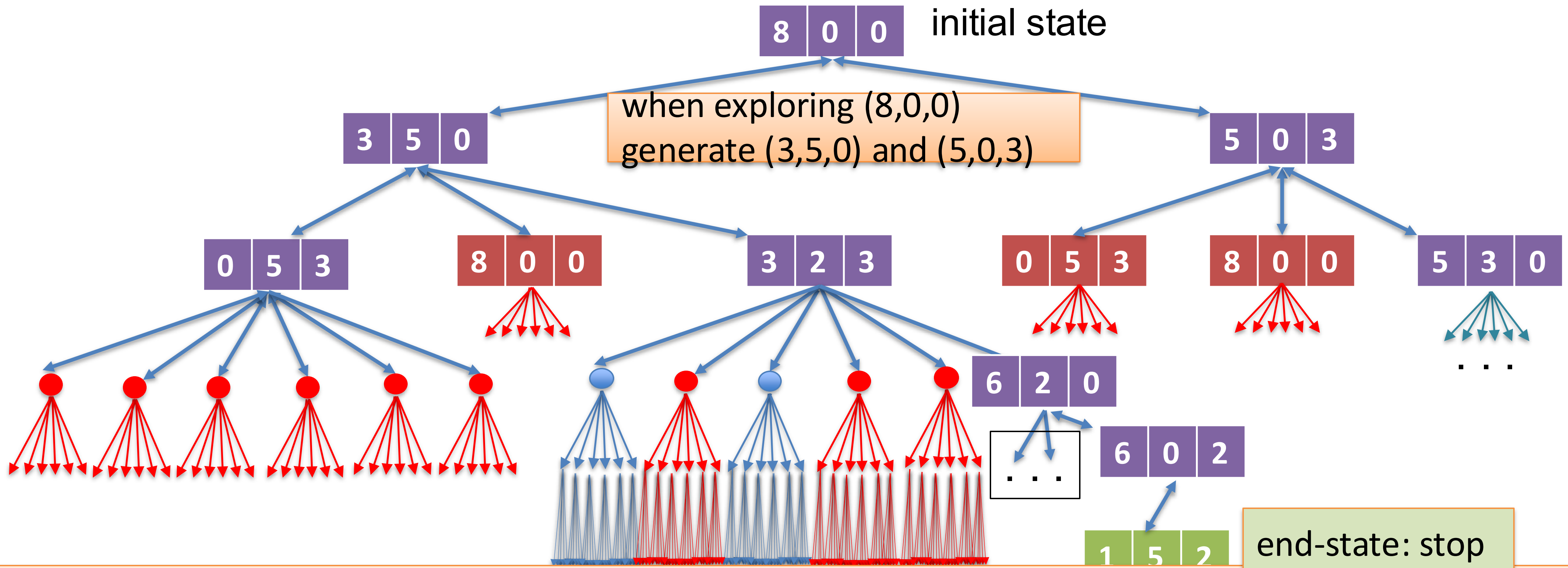
The Milk Pouring Problem & Implicit Graphs

Now, let's consider a scenario where the capacities are parameters A, B, C. Your task is to:

- a) Formulate the task using graphs.
- b) Provide a brief description of an algorithm for solving the reformulated task. For simplicity, assume that a solution does exist.
- c) Suppose that *d* is the number of pourings in the solution, express the algorithm's complexity as a function of *d*.

in this task, a tuple (a, b, c) fully describes the 1 configuration. It's called a state of the task. A state can be tought as a node in a graph.

the edge from node A to node B represent a possible transformation from A to B using a valid pouring action.



The graph is implicit because we .do not strore all nodes and edges at the start. We instead do BFS by:

- generate the inititial node and enqueue it, then do a loop while the queue is not empty:
  - dequeue a state this state to generate all possible next states
  - if a next state is an end state (green): end the algorithm
  - otherwise, if it haven't been seen before: enqueue it
  - How DP helps here?

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.

Additional Slides

[Group/Individual] Question 9.2: 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..i

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

ç	**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∈1..i} (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]									
bundle B[i]									

bundles:

c) Complexity = ?

Running time:

Space:

Baked Beans Bundles: check your solution

(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[8]=2), then 6 (B[8-2]=6)

c) Complexity = ?

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

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