Lecture 6
Algorithmic Techniques Part 2

COMP26120

Giles Reger

March 2019

Implement a function to compute the *n*th number of the Fibonacci sequence

Points from Exercise

Standard divide-and-conquer recursive approach is 'top down'

If we meet the same sub-problem during top-down approach we can memoize e.g. remember the result

If we go 'bottom-up' iteratively we can build the final solution from smaller ones

The General Coin Problem

Question How do we solve the general case of the coin problem?

Greedy solutions fail in general.

The General Coin Problem

Question How do we solve the general case of the coin problem?

Greedy solutions fail in general.

Dynamic programming always produces an optimal solution to this problem.

Suppose we have coin types $1, \ldots, N$ and the value of coin type i is v_i.

Suppose that we have enough coins of each type (if we have a limited number of coins of each type, we can modify the idea below).

How do we make a sum with a minimum number of coins?

The General Coin Problem

Question How do we solve the general case of the coin problem?

Greedy solutions fail in general.

Dynamic programming always produces an optimal solution to this problem.

Suppose we have coin types $1, \ldots, N$ and the value of coin type i is v_i.

Suppose that we have enough coins of each type (if we have a limited number of coins of each type, we can modify the idea below).

How do we make a sum with a minimum number of coins?

Let us quickly think about how we might enumerate all solutions using a configuration $(a_1, a_2, \ldots a_N)$ storing how many we have of each coin.

Dynamic Programming: Coin Problem

Key property:

Let $c(i, s)$ be the minimum number of coins from types 1 to i required to make sum s. Consider what happens if we add another coin type $i + 1$:

$$c(i+1, s) = min \begin{pmatrix} c(i, s) \\ c(i, s - v_{i+1}) + 1 \\ \vdots \\ c(i, s - k \times v_{i+1}) + k \end{pmatrix} \text{ where } (k+1)v_{i+1} > s$$

$$c(i, s) = \infty \text{ if value } s \text{ cannot be made with coins } 1 \ldots i.$$

Dynamic Programming: Coin Problem

Key property:

Let $c(i, s)$ be the minimum number of coins from types 1 to i required to make sum s. Consider what happens if we add another coin type $i + 1$:

$$c(i+1, s) = min \left(\begin{array}{l} c(i, s) \\ c(i, s - v_{i+1}) + 1 \\ \vdots \\ c(i, s - k \times v_{i+1}) + k \end{array} \right) \text{ where } (k+1)v_{i+1} > s$$

$$c(i, s) = \infty \text{ if value } s \text{ cannot be made with coins } 1 \ldots i.$$

This says: if we have solved the problem for coins of types $1 \ldots i$ and now we consider coins of type $i + 1$, then an optimal solution may be to use no coins of type $i + 1$ or one such coin combined with an optimal solution of a smaller problem using coin types $1 \ldots i$, or two coins of type $i + 1$...

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3

$$\begin{aligned} c(2,10) = \quad & min(c(1,10), c(1,9)+1, c(1,8)+2, \ldots) \\ & min(\infty, 1+1, \infty+2, \ldots) \\ & 2 \end{aligned}$$

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3											

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1										

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2									

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3								

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3	4							

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3	4	1						

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3	4	1	2					

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3	4	1	2	3				

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3	4	1	2	3	4			

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3	4	1	2	3	4	1		

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3	4	1	2	3	4	1	2	

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3	4	1	2	3	4	1	2	3

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3	4	1	2	3	4	1	2	3
coin types 1,2,3,4	1	2	3	4	1	1	2	3	1	2	2

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3	4	1	2	3	4	1	2	3
coin types 1,2,3,4	1	2	3	4	1	1	2	3	1	2	2

Answer: Using all 4 coin types, 2 coins is the minimum number needed to make a sum of 11.

What is Happening?

Dynamic programming is a bottom-up method – we solve all smaller problems first then combine them to solve the given problem.

What is Happening?

Dynamic programming is a bottom-up method – we solve all smaller problems first then combine them to solve the given problem.

Question: How efficient is this dynamic programming solution? What is its time complexity?

What is Happening?

Dynamic programming is a bottom-up method – we solve all smaller problems first then combine them to solve the given problem.

Question: How efficient is this dynamic programming solution? What is its time complexity?

The main factor is the size of the table of subproblem results. Thus for N coin types, and a value required of V, the table size is $N \times V$.

For each item we need to scan through the previous row so we get $N \times V^2$.

Notice that some subproblem results are not required for the final solution - but this is not easy to use as a reduction strategy: it is difficult to predict what might be needed.

Properties Required for Dynamic Programming

First two: optimal substructure

Simple Subproblems

The global optimization problem can be broken into subproblems with similar structure to the original problem. Subproblems can be defined with just a few indices, like i, j, k , and so on.

Subproblem Optimality

An optimal solution must be a composition of optimal subproblem solutions, using a relatively simple combining operation. A globally optimal solution should not contain suboptimal subproblems.

Subproblem Overlap

Unrelated subproblems contain subproblems in common.

Dynamic Programming: Edit Distance

Edit (Levenshtein) Distance Problem

Given two strings s_1 and s_2 what are the minimum number or edit operations (insert a new symbol, delete an existing symbol, replace one symbol by another) that can be applied to s_1 to turn it into s_2.

Dynamic Programming: Edit Distance

Edit (Levenshtein) Distance Problem

Given two strings s_1 and s_2 what are the minimum number or edit operations (insert a new symbol, delete an existing symbol, replace one symbol by another) that can be applied to s_1 to turn it into s_2.

Example: The Edit distance between "kitten" and "sitting" is 3.

- kitten → sitten (substitution of "s" for "k")
- sitten → sittin (substitution of "i" for "e")
- sittin → sitting (insertion of "g" at the end).

Dynamic Programming: Edit Distance

Edit (Levenshtein) Distance Problem

Given two strings s_1 and s_2 what are the minimum number or edit operations (insert a new symbol, delete an existing symbol, replace one symbol by another) that can be applied to s_1 to turn it into s_2.

- Simple subproblems - define edit distance between as_1 and bs_2 in terms of distance between of distance between s_1 and s_2
- Subproblem optimality - can show by contradiction
- Subproblem overlap - what if we do insertion+deletion or deletion+insertion? Problem contains lots of symmetry

Dynamic Programming: Edit Distance

The edit distance between strings s_1 and s_2 is distance(s_1, s_2), defined as

$$
\begin{array}{lcl}
\text{distance}(s_1, \epsilon) & = & |s_1| \\
\text{distance}(\epsilon, s_2) & = & |s_2| \\
\text{distance}(as_1, bs_2) & = & \min \begin{cases} \text{distance}(s_1, bs_2) + 1 & \\ \text{distance}(as_1, s_2) + 1 & \\ \text{distance}(s_1, s_2) + 1 & \text{if } a \neq b \\ \text{distance}(s_1, s_2) & \text{if } a = b \end{cases}
\end{array}
$$

Dynamic Programming: Edit Distance

The edit distance between strings s_1 and s_2 is distance(s_1, s_2), defined as

$$
\begin{array}{lcl}
\text{distance}(s_1, \epsilon) & = & |s_1| \\
\text{distance}(\epsilon, s_2) & = & |s_2| \\
\text{distance}(as_1, bs_2) & = & \min \begin{cases} \text{distance}(s_1, bs_2) + 1 & \\ \text{distance}(as_1, s_2) + 1 & \\ \text{distance}(s_1, s_2) + 1 & \text{if } a \neq b \\ \text{distance}(s_1, s_2) & \text{if } a = b \end{cases}
\end{array}
$$

Example:

$$
\text{distance}(cat, hat) = \min \begin{pmatrix} \text{distance}(at, hat) + 1, \\ \text{distance}(cat, at) + 1, \\ \text{distance}(at, at) + 1 \end{pmatrix}
$$

Dynamic Programming: Edit Distance

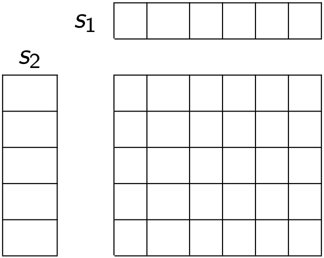
The edit distance between strings s_1 and s_2 is distance(s_1, s_2), defined as

$$
\begin{array}{lcl}
\text{distance}(s_1, \epsilon) & = & |s_1| \\
\text{distance}(\epsilon, s_2) & = & |s_2| \\
\text{distance}(as_1, bs_2) & = & \min \begin{cases} \text{distance}(s_1, bs_2) + 1 & \\ \text{distance}(as_1, s_2) + 1 & \\ \text{distance}(s_1, s_2) + 1 & \text{if } a \neq b \\ \text{distance}(s_1, s_2) & \text{if } a = b \end{cases}
\end{array}
$$

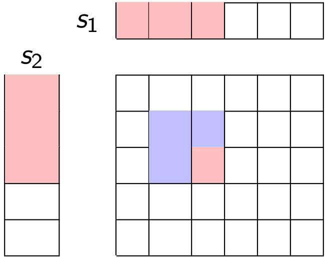
Example:

$$
\text{distance}(cat, hat) = \min \left(\begin{array}{l}
\min \left(\begin{array}{l} \text{distance}(t, hat) + 1, \\ \text{distance}(at, at) + 1, \\ \text{distance}(t, at) + 1 \end{array} \right) + 1 \\
\min \left(\begin{array}{l} \text{distance}(at, at) + 1, \\ \text{distance}(cat, t) + 1, \\ \text{distance}(at, t) + 1 \end{array} \right) + 1 \\
\min \left(\begin{array}{l} \text{distance}(t, at) + 1, \\ \text{distance}(at, t) + 1, \\ \text{distance}(t, t) \end{array} \right) + 1
\end{array} \right)
$$

Dynamic Programming: Edit Distance

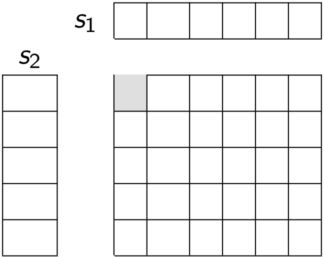


Dynamic Programming: Edit Distance

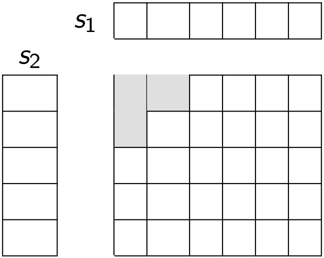


$$D(i,j) = \min \begin{pmatrix} D(i-1,j)+1 & \\ D(i,j-1)+1 & \\ D(i-1,j-1)+1 & \text{if } s_1[i] \neq s_2[j] \\ D(i-1,j-1) & \text{if } s_1[i] = s_2[j] \end{pmatrix}$$

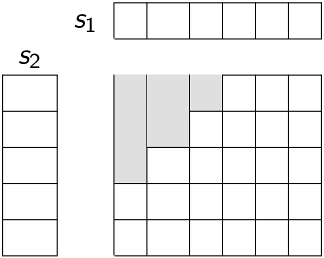
Dynamic Programming: Edit Distance



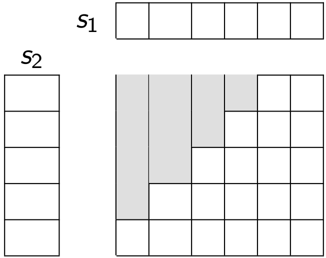
Dynamic Programming: Edit Distance



Dynamic Programming: Edit Distance



Dynamic Programming: Edit Distance



Dynamic Programming: Edit Distance

	c	a	t
h			
a			
t			

Dynamic Programming: Edit Distance

	c	a	t
h	1		
a			
t			

Dynamic Programming: Edit Distance

	c	a	t
h	1	2	
a	2		
t			

Dynamic Programming: Edit Distance

	c	a	t
h	1	2	3
a	2	1	
t	3		

Dynamic Programming: Edit Distance

	c	a	t
h	1	2	3
a	2	1	2
t	3	2	

Dynamic Programming: Edit Distance

	c	a	t
h	1	2	3
a	2	1	2
t	3	2	1

Dynamic Programming: Edit Distance

	c	a	t
h	1	2	3
a	2	1	2
t	3	2	1

Complexity is clearly $|s_1| \times |s_2|$. This is also the space complexity.

Dynamic Programming: Travelling Salesman

Travelling Salesman Problem

Given a graph $\langle V, E, W \rangle$ find a path of minimum weight that visits all vertices

This is an NP-complete problem.

There is a dynamic programming solution (but still exponential of course)

Dynamic Programming: Travelling Salesman

Assume nodes numbered $1, \ldots, n$. Define $D(S, i)$ to be a minimum path starting at 1 and ending at i visiting all nodes in S defined recursively as

$$\begin{aligned} D(\{i\}, i) &= W(1, i) \\ D(S, i) &= \min_{x \in S - i}(D(S - i, x) + W(x, i)) \end{aligned}$$

e.g. to find distance from 1 to i in S first find $x \in S$ with $x \neq i$ that whose path from 1 to x combined with the edge from x to i is minimum.

- Simple subproblems - simpler
- Subproblem optimality - have the recursive definition
- Subproblem overlap - all larger sets depend on all smaller subsets

Dynamic Programming: Travelling Salesman

Assume nodes numbered $1, \ldots, n$. Define $D(S, i)$ to be a minimum path starting at 1 and ending at i visiting all nodes in S defined recursively as

$$\begin{array}{lcl} D(\{i\}, i) & = & W(1, i) \\ D(S, i) & = & \min_{x \in S-i}(D(S-i, x) + W(x, i)) \end{array}$$

e.g. to find distance from 1 to i in S first find $x \in S$ with $x \neq i$ that whose path from 1 to x combined with the edge from x to i is minimum. Iterate through larger subsets of V as $D(S, i)$ always defined in terms of smaller subsets. Solution is $D(V-1, 1)$. Held-Karp algorithm.

Complexity is still the size of the table but how big is it?

Index columns by states and rows by ?

Dynamic Programming: Travelling Salesman

Assume nodes numbered $1, \ldots, n$. Define $D(S, i)$ to be a minimum path starting at 1 and ending at i visiting all nodes in S defined recursively as

$$\begin{array}{lcl} D(\{i\}, i) & = & W(1, i) \\ D(S, i) & = & \min_{x \in S-i}(D(S - i, x) + W(x, i)) \end{array}$$

e.g. to find distance from 1 to i in S first find $x \in S$ with $x \neq i$ that whose path from 1 to x combined with the edge from x to i is minimum. Iterate through larger subsets of V as $D(S, i)$ always defined in terms of smaller subsets. Solution is $D(V - 1, 1)$. Held-Karp algorithm.

Complexity is still the size of the table but how big is it?

Index columns by states and rows by subets of V

Dynamic Programming: Travelling Salesman

Assume nodes numbered $1, \ldots, n$. Define $D(S, i)$ to be a minimum path starting at 1 and ending at i visiting all nodes in S defined recursively as

$$
\begin{aligned}
D(\{i\}, i) &= W(1, i) \\
D(S, i) &= \min_{x \in S - i}(D(S - i, x) + W(x, i))
\end{aligned}
$$

e.g. to find distance from 1 to i in S first find $x \in S$ with $x \neq i$ that whose path from 1 to x combined with the edge from x to i is minimum. Iterate through larger subsets of V as $D(S, i)$ always defined in terms of smaller subsets. Solution is $D(V - 1, 1)$. Held-Karp algorithm.

Complexity is still the size of the table but how big is it?

Index columns by states and rows by subets of V ... so get $(n \times 2^n) \times n$

A subsequence of a string/list is obtained by dropping some elements. A subsequence differs from a substring/sublist as the resulting elements do not need to be next to each other in the original.

Longest Common Subsequence Problem

Given two sequences find the longest common subsequence each the maximum sequence that is a subsequence of both original sequences.

Can define recursively over prefixes.

But does it look familiar?

Another Example

Does anybody recognise this equation?

$$\mathcal{L}^{\leq k}_{j\to i} = \mathcal{L}^{\leq k-1}_{j\to i} \cup \mathcal{L}^{\leq k-1}_{j\to k} \cdot (\mathcal{L}^{\leq k-1}_{k\to k})^* \cdot \mathcal{L}^{\leq k-1}_{k\to i}.$$

Another Example

Does anybody recognise this equation?

$$\mathcal{L}_{j\to i}^{\leq k} = \mathcal{L}_{j\to i}^{\leq k-1} \cup \mathcal{L}_{j\to k}^{\leq k-1} \cdot (\mathcal{L}_{k\to k}^{\leq k-1})^* \cdot \mathcal{L}_{k\to i}^{\leq k-1}.$$

Translation from finite state automata to regular expressions can be computed using dynamic programming

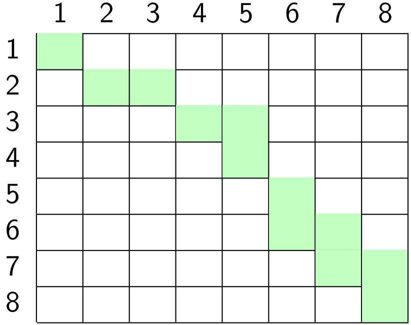
Not A Drinking Game

Your friends suggest a game that is not a drinking game. They place a random drink on each square of a Chess Board. Each drink contains a different amount of liquid. You place a Queen on $(1, 1)$ and move it to $(8, 8)$ whilst drinking every drink you land on. You know the amount of liquid on each square, given by a handy function $d(i, j)$. The winner is the person who drinks the least liquid.

	1	2	3	4	5	6	7	8
1								
2								
3								
4								
5								
6								
7								
8								

Not A Drinking Game

Your friends suggest a game that is not a drinking game. They place a random drink on each square of a Chess Board. Each drink contains a different amount of liquid. You place a Queen on $(1, 1)$ and move it to $(8, 8)$ whilst drinking every drink you land on. You know the amount of liquid on each square, given by a handy function $d(i, j)$. The winner is the person who drinks the least liquid.



Dynamic Programming: Other Applications

A few examples of optimisation problems with dynamic programming solutions

- Some path-finding algorithms use dynamic programming, for example Floyd's algorithm for the all-nodes shortest path problem.
- Some text similarity tests: For example, longest common subsequence.
- Knapsack problems: The 0/1 Knapsack problem can be solved using dynamic programming.
- Constructing optimal search trees.
- Some travelling salesperson problems have dynamic programming solutions.
- Genome matching and protein-chain matching use dynamic programming algorithms - invited lecture.

Dynamic Programming: Other Applications

A few examples of optimisation problems with dynamic programming solutions

- Some path-finding algorithms use dynamic programming, for example Floyd's algorithm for the all-nodes shortest path problem.
- Some text similarity tests: For example, longest common subsequence.
- **Knapsack problems: The 0/1 Knapsack problem can be solved using dynamic programming.**
- Constructing optimal search trees.
- Some travelling salesperson problems have dynamic programming solutions.
- Genome matching and protein-chain matching use dynamic programming algorithms - invited lecture.