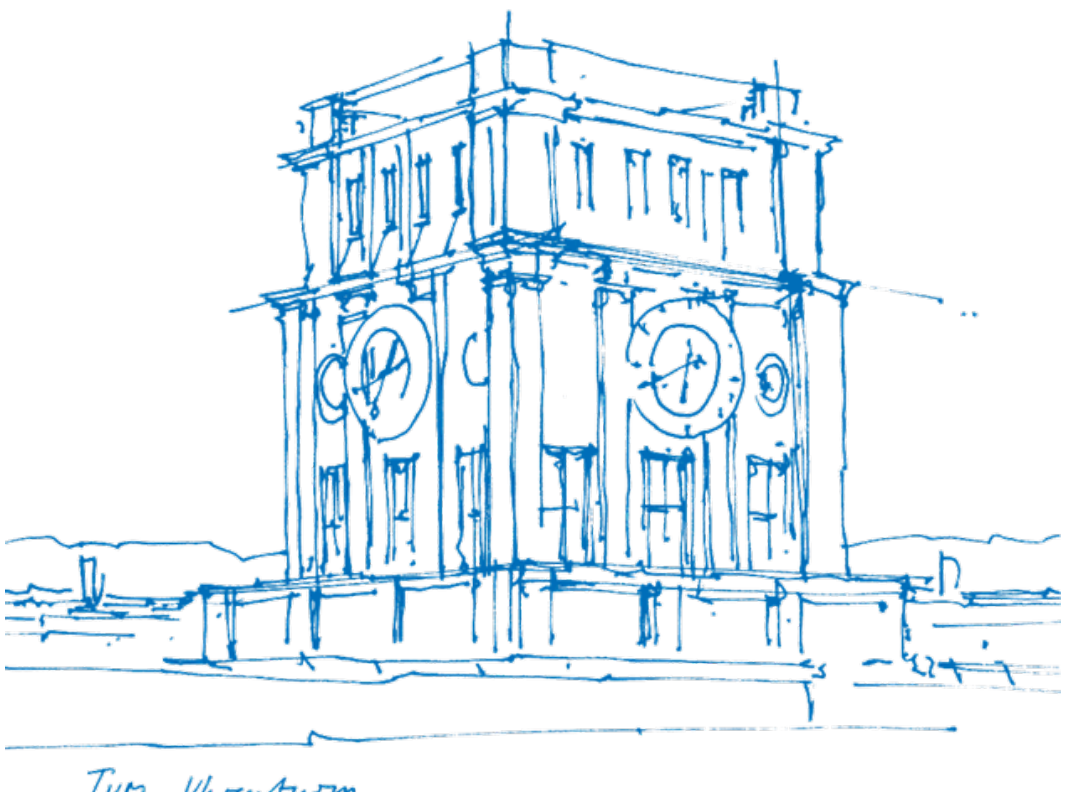
School of Computation, Information and Technology
Technical University of Munich

AACPP 2025

Week 3: Greedy or Dynamic?



Mateusz Gienieczko, **Mykola Morozov**

School of Computation, Information and Technology
Technical University of Munich

2025.05.13

First round – survey

Second round

Second deadline – **27**.05.2025, 10:00 AM.

No class on 20th due to SVV.

TOY – Toy Ordering Plan

TOY – Toy Ordering Plan

Very simple.

TOY – Toy Ordering Plan

Very simple.

Take the stores that have any day where the number of toys is at least n.

TOY – Toy Ordering Plan

Very simple.

Take the stores that have any day where the number of toys is at least n.

Select the one with lowest price and return that times n.

TOY – Toy Ordering Plan

Very simple.

Take the stores that have any day where the number of toys is at least n.

Select the one with lowest price and return that times n.

Time: $\mathcal{O}(sd)$. Can be done in $\mathcal{O}(1)$ memory but no need.

TOY – Toy Ordering Plan

Very simple.

Take the stores that have any day where the number of toys is at least n.

Select the one with lowest price and return that times n.

Time: $\mathcal{O}(sd)$. Can be done in $\mathcal{O}(1)$ memory but no need.

Subtasks are a lie.

ZOO – Zoomies

One way of looking at it: efficiently distribute k breaks to minimize energy costs.

ZOO – Zoomies

One way of looking at it: efficiently distribute k breaks to minimize energy costs.

However.

If we fix starting e we can check if it's enough in $\mathcal{O}(n)$.

ZOO – Zoomies

Simulate by keeping the current energy level.

If we can jump, we jump.

If not, we take a break and continue.

Either we reach the end or run out of breaks.

ZOO – Zoomies

Subtask 1: $n \leq 250, e_i \leq 10^3$. So $E = \sum_i e_i \leq 250\,000$.

Obviously the result is in $[1, E]$. So we can go through all possible starting values and check if they work.

Takes $O(nE)$ time.

ZOO – Zoomies

Subtask 2 – we'll talk about it later (it's DP!).

ZOO – Zoomies

Subtask 3 – all energy costs are equal to some e.

You can decide the required energy with some math in $\mathcal{O}(1)$.

If $k+1 \mid n$ then we can have $k+1$ equal jump series of length $l = \frac{n}{k+1}$.

We need $le + k$ starting energy.

If $k+1 \nmid n$ then we need to do rem $= n - l(k+1)$ series of length $l+1$.

This requires $e - k + \text{rem} - 1$ additional energy (if positive).

BTW

You can combine subtasks.

E.g.

```
let input = Input::read();
if input.n <= 250 {
  run_brute_force(&input);
} else {
  run_special_case(&input);
}
```

ZOO – Zoomies

Model solution – binary-search through the result.

ZOO – Zoomies

Model solution – binary-search through the result.

Monotonicity – if e energy is enough then obviously $e + 1$ is also enough.

ZOO – Zoomies

Model solution – binary-search through the result.

Monotonicity – if e energy is enough then obviously $e + 1$ is also enough.

We already have a $\mathcal{O}(n)$ check for a given energy level.

So $\mathcal{O}(n \log E)$ time and

ZOO – Zoomies

Manual bin-search (Rust).

```rust
let mut lower_bound: u64 = 1;
let mut upper_bound: u64 = input.jumps.iter().sum();

while lower_bound < upper_bound {
  let mid = lower_bound.midpoint(upper_bound);
  if is_enough(&input, mid) {
    upper_bound = mid;
  } else {
    lower_bound = mid + 1;
  }
}
```

ZOO – Zoomies

Manual bin-search (C++).

```
uint64_t midpoint(uint64_t x, uint64_t y) {
  return ((x ^ y) >> 1) + (x & y);
}
while (lower_bound < upper_bound) {
  uint64_t mid = midpoint(lower_bound, upper_bound);
  if (is_enough(input, mid)) {
    upper_bound = mid;
  } else {
    lower_bound = mid + 1;
  }
}
```

Recall the plan

- **Greedy and dynamic programming (DP)** ← *we are here*
- Trees
- Graphs
- Ways to turn graphs into trees (DFS, BFS, Dijkstra, MST)
- Ways to run DP on graphs (Toposort)
- Advanced graph algorithms (Matchings, flows)
- Binary Search Trees
- Number theory
- String algorithms (KMP, tries, suffix tables)

Some problems can't even be solved efficiently (NP-completeness)

Optimisation problems

Make a number of decisions creating a strategy that produces an optimal (minimal, maximal, ...) result.

Can often be solved with *dynamic programming*.

Some can be solved *greedily*.

Greedy

During each step make the decision that appears the best (locally optimal).

Classic example: task assignment.

Greedy example

Dexter is a very busy cat. During the day he has many (n) activities he can choose to do like napping, running, sleeping, eating, resting, etc. Each activity has a start time s_i and end time f_i. Some of them conflict with each other (times overlap), and there's only one Dexter who can do only one activity at a time. Help him choose the maximum possible number of activities without any conflicts!

```
11
1 3 0 5 3 5 6 8 8 2 12
4 5 6 7 9 9 10 11 12 14 16

4
1 4 8 11
```

Greedy example

Once we choose an activity we can’t choose anything until it ends.

Intuition: we want to choose the activity that limits as least, i.e. ends earliest.

Turns out it’s true!

Greedy solution

Sort the activities by their end time.

Set time to $t = 0$.

Choose the first activity that starts at or after t.

Set t to its ending time and continue.

$\mathcal{O}(n \log n)$ for sorting ($\mathcal{O}(n)$ if input is sorted).

Greedy solution

Another formulation: let S_i be the solution for the problem assuming we can only schedule activities that start after i.

Two crucial properties of greedy solutions:

- Optimal problem substructure. Here if we choose some activity a ending at j then we can use S_j as the optimal subsolution, i.e. pick $S_j \cup \{a\}$.
- The greedy choice always leads to optimal results. Here minimal j gives maximal S_j.

Greedy solution

Another formulation: let S_i be the solution for the problem assuming we can only schedule activities that start after i.

Two crucial properties of greedy solutions:

- Optimal problem substructure. Here if we choose some activity a ending at j then we can use S_j as the optimal subsolution, i.e. pick $S_j \cup \{a\}$.
- The greedy choice always leads to optimal results. Here minimal j gives maximal S_j.

BTW only S_i where $i = f_k$ for some activity a_k are interesting.

Greedy – proving correctness

Greedy algorithms are naturally inductive.

Usual proof strategy – **replacement**.

Show that if we have an optimal solution then it can just as well be the greedy one.

Greedy – proving correctness

Solution for $S_{\max f}$ is trivially empty.

Assume S_i is optimal for $i > n$ and in S_n the first activity to end is a ending at j. Then there exists optimal $S'_n = \{a\} \cup S_j$ (and thus $|S_n| = |S_j| + 1$).

Moreover, if there exists a' that starts at or after n and ends at $j' \leq j$ then $\{a'\} \cup S_{j'}$ is also optimal, in particular S_j is also optimal for j'.

Greedy – other examples

Shortest paths in a graph.

Shortest paths in a weighted graph (with non-negative edges).

Huffman coding.

Special cases of graph colouring.

Greedy – other examples

Shortest paths in a graph.

Shortest paths in a weighted graph (with non-negative edges).

Huffman coding.

Special cases of graph colouring.

Continuous **knapsack problem.**

Greedy – continuous knapsack problem

Dexter got his paws on the stash of creamy snacks. He'd love to eat them all, but his stomach has a fixed capacity c. Each snack i has an amount a_i and its tastiness t_i. Dexter can eat any integer amount between 0 and a_i of the snack, and he wants to maximise the overall tastiness sum.

```
3 5
1 2 3
6 10 12

24
```

Greedy – continuous knapsack problem

Greedy approach works: order all snacks by their value ratio $\frac{t_i}{a_i}$ and fill up to capacity.

Here: take all of item 1, 2, and then 2 units of item 3 for a total of 24.

$\mathcal{O}(n \log n)$ for sorting.

Sidenote – comparing rationals

No need to use floating-point to work with rational numbers.

Example: comparison for non-negative numbers:

$$\frac{a}{b} < \frac{c}{d} \Leftrightarrow ad < cb$$

NOT Greedy – discrete knapsack problem

Dexter got his paws on the stash of crunchy *snacks. He'd love to eat them all, but his stomach has a fixed capacity c. Each snack i has an amount a_i and its tastiness t_i. Dexter can either eat the whole snack or leave it alone, and he wants to maximise the overall tastiness sum.*

```
3 5
1 2 3
6 10 12

22
```

NOT Greedy – discrete knapsack problem

A similar greedy approach **does not** work.

Here item 1 is the most valuable so we would take it, but the optimal solution does not include it.

This is actually a classic DP problem.

Dynamic programming

The name is quite literally a buzzword so don’t worry about it.

We will still be building solutions inductively, but without greedy assumptions.

Dynamic programming – discrete knapsack

Let $V_{k,i}$ be the optimal value for having i capacity remaining after picking from the first k items.

$V_{0,c} = 0$

when $i < a_k$: $V_{k,i} = V_{k-1,i}$; otherwise

$V_{k,i} = \max\left(V_{k-1,i}, V_{k-1,i-a_k} + v_k\right)$

Dynamic programming – top-down

Such formulation naturally leads to a recursive algorithm.

However, naively doing it will lead to an exponential running time due to recomputation.

Memoisation means remembering all results for previously computed tasks to be used on demand.

Dynamic programming – memoisation

```
solve_knapsack(k, i)
  if k == 0 { return 0 }
  if memo.contains(k, i) { return memo.get(k, i) }
  let res = solve_knapsack(k - 1, i)
  if a[k] <= i {
    res = max(res, solve_knapsack(k - 1, i - a[k]) + v[k])
  }
  memo.set(k, i, res)
  return res
}
```

Dynamic programming – bottom-up

Usually there is an iterative method of filling up the *DP array*.

$i \backslash k$	0	1	2	3
0				
1				
2				
3				
4				
5	0			

Dynamic programming – bottom-up

Usually there is an iterative method of filling up the *DP array*.

$i \backslash k$	0	1	2	3
0				
1				
2				
3				
4		6		
5	0	0		

Dynamic programming – bottom-up

Usually there is an iterative method of filling up the *DP array*.

$i \backslash k$	**0**	**1**	**2**	**3**
0				
1				
2			16	
3			10	
4		6	6	
5	0	0	0	

Dynamic programming – bottom-up

Usually there is an iterative method of filling up the *DP array*.

$i \backslash k$	**0**	**1**	**2**	**3**
0				22
1				18
2			16	16
3			10	10
4		6	6	6
5	0	0	0	0

Dynamic programming – which to choose?

Subjective opinion: usually recursive is more straight-forward as it follows directly from the formula.

Iteration tends to be faster, especially if recursion is not tail.

On the other hand, recursion by design visits only reachable states – in the DP table before all empty cells have to be visited, even though they are useless.

In both cases here we have $\mathcal{O}(nc)$ complexity.

Dynamic programming – result recovery

Often tasks ask not only for the optimal value, but the entire solution.

This is usually not hard to recover by recording some additional information in the DP array (“where did we come from”) and backtracking.

Dynamic programming – another example

Dexter loves stick snacks. He's very picky and enjoys different lengths of sticks differently. You have a single stick of length n and can divide it into any number of smaller pieces with integral length. How to divide the stick to maximise Dexter's total enjoyment?

```
10
1 5 8 9 10 17 17 20 24 25

27
```

Dynamic programming – another example

When dividing a stick of length n:

- either don't divide at all; or
- divide once and use the optimal solution for the two resulting fragments

Dynamic programming – another example

When dividing a stick of length n:

- either don't divide at all; or
- divide once and use the optimal solution for the two resulting fragments

$V_k = \max(v_k, \max_i V_i + V_{k-i})$

k	0	1	2	3	4	5	6	7	8	9	10
V_k	0	1	5	8	10	13	17	18	22	25	27

$\mathcal{O}(n^2)$, n states each computed in $\mathcal{O}(n)$.

Dynamic programming – yet another

Zoomies can be solved with DP! (Subtask 2)

$\mathrm{DP}[b][f][t]$ – how much energy is required to complete all jumps between f and t using b breaks.

$\mathrm{DP}[0][f][t] = \sum_{f \le i \le t} e_i$

If we break at $f < j < t$ then we pay $\sum_{f \le i \le j} e_i + \mathrm{DP}[b-1][j+1][t] + 1$.

We can also not break at all for a baseline of $\mathrm{DP}[b-1][f][t]$.

This is $\mathcal{O}(n^2 k)$ (the running sum of e_i is computed on the fly).

Dynamic programming – requirements

For DP to be applicable the problem has to have **optimal substructure**.

To solve a big problem we can use optimal solutions of smaller problems solved **independently**.

Dynamic programming – correctness

In our two examples we can use DP because of optimal substructure.

Choosing other items up to some capacity that still allows for our current item is independent.

Splitting two smaller sticks is independent.

Correctness can be easily proven by induction.

Dynamic programming – other examples

Longest common subsequence of two strings in $\mathcal{O}(nm)$.

		b	d	c	a	b	a
	0	0	0	0	0	0	0
a	0						
b	0						
c	0						
b	0						
d	0						
a	0						
b	0						

Dynamic programming – other examples

Longest common subsequence of two strings in $\mathcal{O}(nm)$.

		b	d	c	a	b	a
	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1
b	0	1	1	1	1	2	2
c	0	1	1	2	2	2	2
b	0	1	1	2	2	3	3
d	0	1	2	2	2	3	3
a	0	1	2	2	3	3	4
b	0	1	2	2	3	4	4

Dynamic programming – other examples

Longest common subsequence of two strings in $\mathcal{O}(nm)$.

		b	d	c	a	b	a
	0	0	0	0	0	0	0
a	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖
b	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖	2 ←
c	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑
b	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ←
d	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑
a	0	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	4 ↖
b	0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑

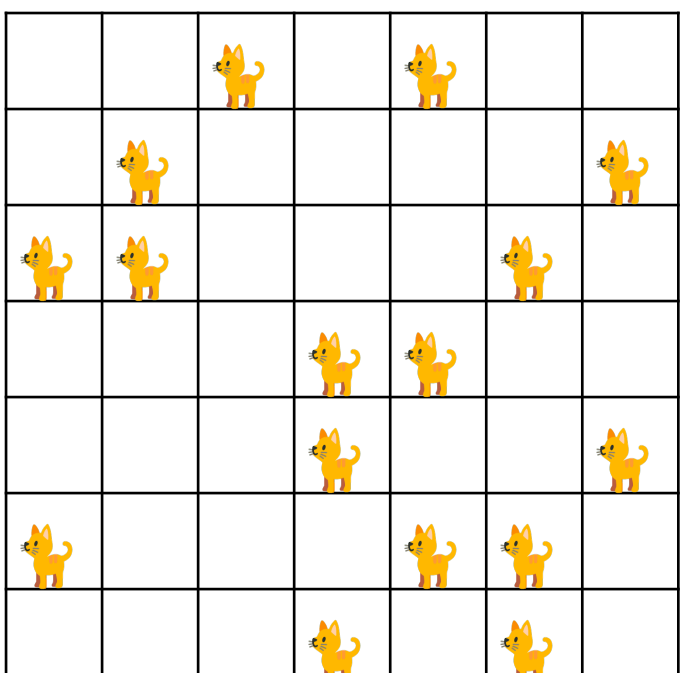
Dynamic programming – other examples

Longest common subsequence of two strings in $\mathcal{O}(nm)$.

		b	d	c	a	b	a
	0	0	0	0	0	0	0
a	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖
b	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖	2 ←
c	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑
b	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ←
d	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑
a	0	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	4 ↖
b	0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑

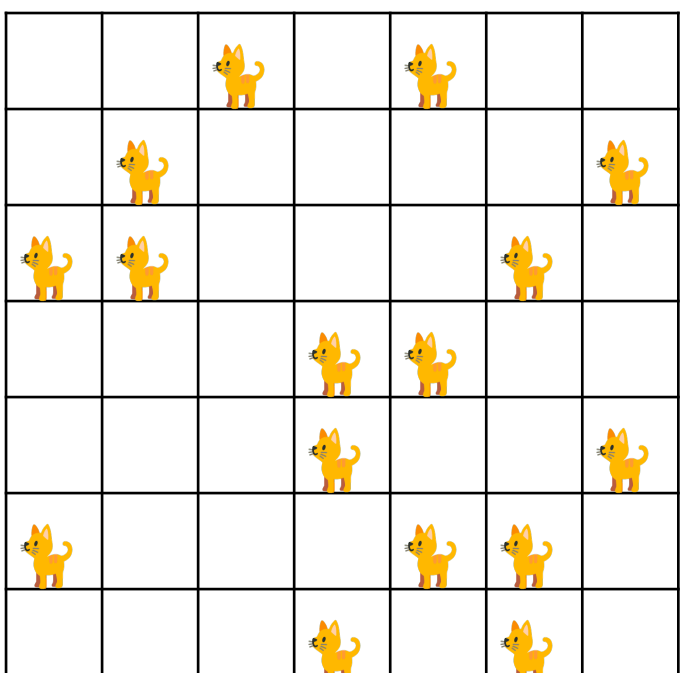
Dynamic programming – other examples

Tasks of the form “find a path from top-left to bottom-right optimising some value”.



Dynamic programming – other examples

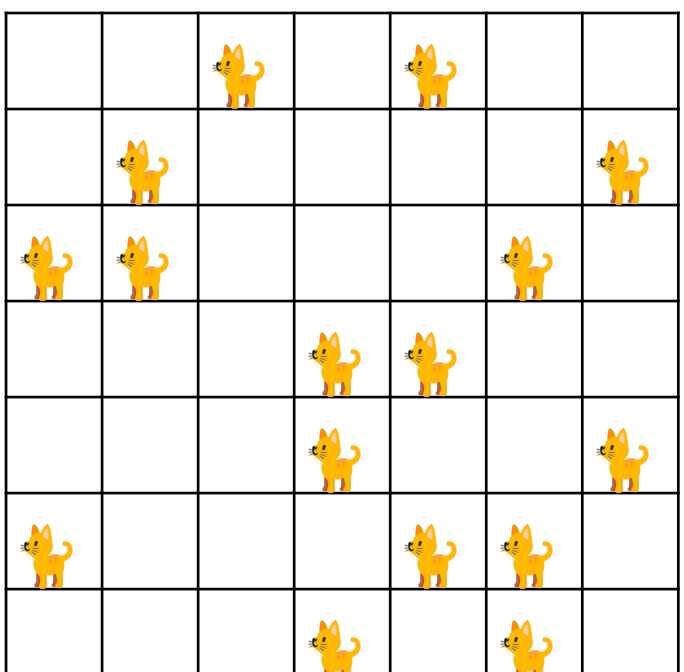
Tasks of the form “find a path from top-left to bottom-right optimising some value”.



0	0	1	1	2	2	2

Dynamic programming – other examples

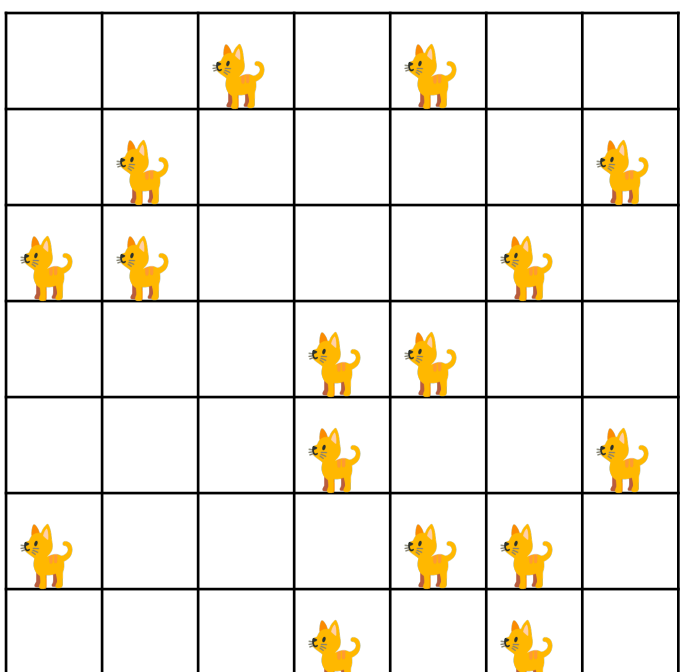
Tasks of the form “find a path from top-left to bottom-right optimising some value”.



0	0	1	1	2	2	2
0	1	1	1	2	2	3

Dynamic programming – other examples

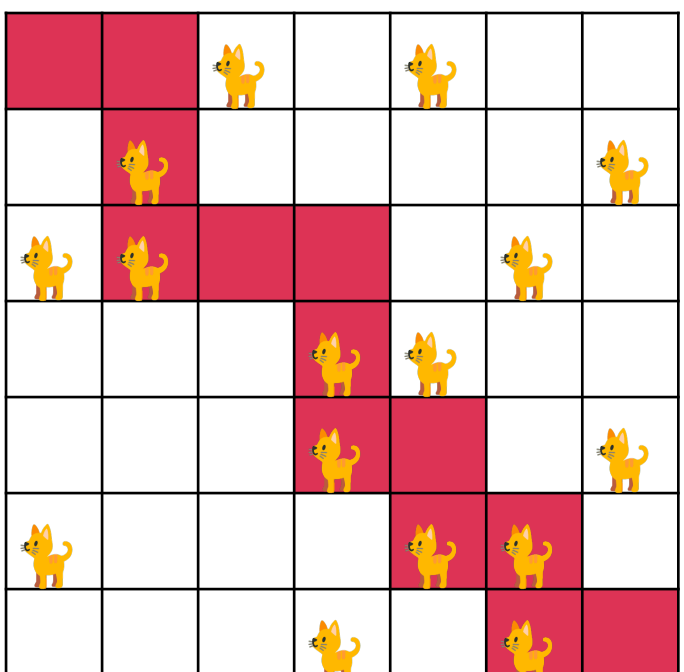
Tasks of the form “find a path from top-left to bottom-right optimising some value”.



0	0	1	1	2	2	2
0	1	1	1	2	2	3
1	2	2	2	2	3	3
1	2	2	3	4	4	4
1	2	2	4	4	4	5
2	2	2	4	5	6	6
2	2	2	5	5	7	7

Dynamic programming – other examples

Tasks of the form “find a path from top-left to bottom-right optimising some value”.



0	0	1	1	2	2	2
0	1	1	1	2	2	3
1	2	2	2	2	3	3
1	2	2	3	4	4	4
1	2	2	4	4	4	5
2	2	2	4	5	6	6
2	2	2	5	5	7	7

Dynamic programming – other examples

LCS-likes (e.g. editing distance)

Optimal ordering of matrices to multiply.

Join ordering.

Counting combinations.

Shortest paths in graphs with negative weights.

Longest path **in a DAG**.

NOT optimal substructure

Shortest paths in a graph have optimal substructure.

For the shortest path $v \rightsquigarrow u$ pick a middle vertex w and consider $v \rightsquigarrow w \rightsquigarrow u$.

Picking the shortest path $v \rightsquigarrow w$ and $w \rightsquigarrow u$ works.

For **longest** simple paths this substructure does not exist.

The longest paths $v \rightsquigarrow w$ and $w \rightsquigarrow u$ might share vertices which cannot be picked again, so they're **not independent**.

This problem is actually NP-complete.

That being said...

Most NP-complete problems have a natural DP characterisation, but the state space is exponential.

E.g. in the longest-path case consider all subsets of vertices to pick to the path.

See you in two weeks

SLI and TES: 27.05.2025, 10:00 AM

Good luck!