Solution Slides FSR Coding Cup 2023

Marian Zuska, Brutenis Gliwa

Universität Rostock

2023

Intuitive Citations

Problem

- Given a list of names. Print the lexicographically smallest surname and add `" et al."`.

Intuitive Citations

Problem

- Given a list of names. Print the lexicographically smallest surname and add `" et al."`.

Solution

- Remove part before space (forename).

Intuitive Citations

Problem

- Given a list of names. Print the lexicographically smallest surname and add `" et al."`.

Solution

- Remove part before space (forename).
- Find the lexicographically smallest string by sorting and taking the first element.

Intuitive Citations

Problem

- Given a list of names. Print the lexicographically smallest surname and add `" et al."`.

Solution

- Remove part before space (forename).
- Find the lexicographically smallest string by sorting and taking the first element.
- Print string + `" et al."`.

Fascinating Books

Problem

- Check if given list of strings contains every letter of the alphabet.

Fascinating Books

Problem

- Check if given list of strings contains every letter of the alphabet.

Solution

- Can concatenate strings and solve for a single string.

Fascinating Books

Problem

- Check if given list of strings contains every letter of the alphabet.

Solution

- Can concatenate strings and solve for a single string.
- For each character:
  - If character is letter: Add as lowercase to set.

Fascinating Books

Problem

- Check if given list of strings contains every letter of the alphabet.

Solution

- Can concatenate strings and solve for a single string.
- For each character:
  - If character is letter: Add as lowercase to set.
- Check if length of set is 26.

Fascinating Books

Problem

- Check if given list of strings contains every letter of the alphabet.

Solution

- Can concatenate strings and solve for a single string.
- For each character:
  - If character is letter: Add as lowercase to set.
- Check if length of set is 26.

Gotchas

- Capitalization does not matter.

Fascinating Books

Problem

- Check if given list of strings contains every letter of the alphabet.

Solution

- Can concatenate strings and solve for a single string.
- For each character:
  - If character is letter: Add as lowercase to set.
- Check if length of set is 26.

Gotchas

- Capitalization does not matter.
- Strings do not only contain letters.

Leaderboard Prediction

Problem

- Given the times you need to solve each of the n problems.
- Determine the minimal penalty you can get on the contest.

Leaderboard Prediction

Problem

- Given the times you need to solve each of the n problems.
- Determine the minimal penalty you can get on the contest.

Solution

- The time you needed for the first problem will be added to the penalty of all problems you solve.

Leaderboard Prediction

Problem

- Given the times you need to solve each of the n problems.
- Determine the minimal penalty you can get on the contest.

Solution

- The time you needed for the first problem will be added to the penalty of all problems you solve.
- It is always best to solve shortest problems first.

Leaderboard Prediction

Problem

- Given the times you need to solve each of the n problems.
- Determine the minimal penalty you can get on the contest.

Solution

- The time you needed for the first problem will be added to the penalty of all problems you solve.
- It is always best to solve shortest problems first.
- Greedy solution: Sort problems by length, then simulate.

Compilers Brackets

Problem

- Check if given bracket pattern is valid.
- `"{{}{}"` is not valid.
- `"{{}{{}}}"` is valid.

Compilers Brackets

Problem

- Check if given bracket pattern is valid.
- `"{{}{}"` is not valid.
- `"{{}{{}}}"` is valid.

Solution

- Count number of currently open brackets.

Compilers Brackets

Problem

- Check if given bracket pattern is valid.
- `"{{}{}"` is not valid.
- `"{{}{{}}}"` is valid.

Solution

- Count number of currently open brackets.
- Begin with `open = 0`

Compilers Brackets

Problem

- Check if given bracket pattern is valid.
- `"{{}{}"` is not valid.
- `"{{}{{}}}"` is valid.

Solution

- Count number of currently open brackets.
- Begin with `open = 0`
- `"{"` $\rightarrow$ `open++`

Compilers Brackets

Problem

- Check if given bracket pattern is valid.
- `"{{}{}"` is not valid.
- `"{{}{{}}}"` is valid.

Solution

- Count number of currently open brackets.
- Begin with `open = 0`
- `"{"` $\rightarrow$ `open++`
- `"}"` $\rightarrow$ `open--`

Compilers Brackets

Problem

- Check if given bracket pattern is valid.
- `"{{}{}"` is not valid.
- `"{{}{{}}}"` is valid.

Solution

- Count number of currently open brackets.
- Begin with `open = 0`
- `"{"` $\rightarrow$ `open++`
- `"}"` $\rightarrow$ `open--`
- Pattern invalid if `open < 0` at any time.

Compilers Brackets

Problem

- Check if given bracket pattern is valid.
- `"{{}{}"` is not valid.
- `"{{}{{}}}"` is valid.

Solution

- Count number of currently open brackets.
- Begin with `open = 0`
- `"{"` $\rightarrow$ `open++`
- `"}"` $\rightarrow$ `open--`
- Pattern invalid if `open < 0` at any time.
- Pattern invalid if `open != 0` in the end.

Problem

- Given (n_1, n_2, n_4) lego bricks of size 1, 2 and 4.
- Build the highest wall of width w.

Dam Construction

Problem

- Given (n_1, n_2, n_4) lego bricks of size 1, 2 and 4.
- Build the highest wall of width w.

Solution

- Should always use bricks of higher size first to maintain flexibility.

Dam Construction

Problem

- Given (n_1, n_2, n_4) lego bricks of size 1, 2 and 4.
- Build the highest wall of width w.

Solution

- Should always use bricks of higher size first to maintain flexibility.
- Greedy solution by using n_4 bricks, then filling up with n_2, then with n_1.

Dam Construction

Problem

- Given (n_1, n_2, n_4) lego bricks of size 1, 2 and 4.
- Build the highest wall of width w.

Solution

- Should always use bricks of higher size first to maintain flexibility.
- Greedy solution by using n_4 bricks, then filling up with n_2, then with n_1.

Gotchas

- In slower languages (like Python) you need to calculate in 1 step how many bricks of each type you need.

Bicycle Lock

Problem

- Input: Initial lock position I and final lock position F of length n.
- Move to final position by always turning two consecutive dials at once.

Bicycle Lock

Problem

- Input: Initial lock position I and final lock position F of length n.
- Move to final position by always turning two consecutive dials at once.

Solution

- Dial 1 can only be turned by turning dials 1 and 2.

Bicycle Lock

Problem

- Input: Initial lock position I and final lock position F of length n.
- Move to final position by always turning two consecutive dials at once.

Solution

- Dial 1 can only be turned by turning dials 1 and 2.
- It needs to be turned from I_1 to I_2.

Bicycle Lock

Problem

- Input: Initial lock position I and final lock position F of length n.
- Move to final position by always turning two consecutive dials at once.

Solution

- Dial 1 can only be turned by turning dials 1 and 2.
- It needs to be turned from I_1 to I_2.
- After turning that, we have a new subproblem of length $n - 1$.

Bicycle Lock

Problem

- Input: Initial lock position I and final lock position F of length n.
- Move to final position by always turning two consecutive dials at once.

Solution

- Dial 1 can only be turned by turning dials 1 and 2.
- It needs to be turned from I_1 to I_2.
- After turning that, we have a new subproblem of length $n - 1$.
- We can solve this recursively.

Bicycle Lock

Problem

- Input: Initial lock position I and final lock position F of length n.
- Move to final position by always turning two consecutive dials at once.

Solution

- Dial 1 can only be turned by turning dials 1 and 2.
- It needs to be turned from I_1 to I_2.
- After turning that, we have a new subproblem of length $n-1$.
- We can solve this recursively.

Gotchas

- Always two ways to turn dials: Clockwise or Anti-clockwise.

Bicycle Lock

Problem

- Input: Initial lock position I and final lock position F of length n.
- Move to final position by always turning two consecutive dials at once.

Solution

- Dial 1 can only be turned by turning dials 1 and 2.
- It needs to be turned from I_1 to I_2.
- After turning that, we have a new subproblem of length $n - 1$.
- We can solve this recursively.

Gotchas

- Always two ways to turn dials: Clockwise or Anti-clockwise.
- Need to check if last dial is at the right positition in the end.

Aquarium Maze

Problem

- Input: grid of "." and "#" squares.
- Grid is filled with water from the top.
- Water can move down, left and right.

Aquarium Maze

Problem

- Input: grid of "." and "#" squares.
- Grid is filled with water from the top.
- Water can move down, left and right.

Solution

- Can simulate water by starting at a top square and then traversing the graph e.g. using BFS or DFS.

Aquarium Maze

Problem

- Input: grid of "." and "#" squares.
- Grid is filled with water from the top.
- Water can move down, left and right.

Solution

- Can simulate water by starting at a top square and then traversing the graph e.g. using BFS or DFS.
- For each top square:
  - if square == "." and not yet visited:
    - add 1 to answer.

Aquarium Maze

Problem

- Input: grid of "." and "#" squares.
- Grid is filled with water from the top.
- Water can move down, left and right.

Solution

- Can simulate water by starting at a top square and then traversing the graph e.g. using BFS or DFS.
- For each top square:
  - if square == "." and not yet visited:
    - add 1 to answer.
    - visit left, right and bottom neighbour recursively.

Aquarium Maze

Problem

- Input: grid of "." and "#" squares.
- Grid is filled with water from the top.
- Water can move down, left and right.

Solution

- Can simulate water by starting at a top square and then traversing the graph e.g. using BFS or DFS.
- For each top square:
  - if square == "." and not yet visited:
    - add 1 to answer.
    - visit left, right and bottom neighbour recursively.

Gotchas

- Need to start once at every point on the top.

Aquarium Maze

Problem

- Input: grid of "." and "#" squares.
- Grid is filled with water from the top.
- Water can move down, left and right.

Solution

- Can simulate water by starting at a top square and then traversing the graph e.g. using BFS or DFS.
- For each top square:
  - if square == "." and not yet visited:
    - add 1 to answer.
    - visit left, right and bottom neighbour recursively.

Gotchas

- Need to start once at every point on the top.
- Otherwise we might miss some air bubbles.

Hidden Words

Problem

- Given n strings s_1 ... s_n.
- Find a string S that contains all n strings in consecutive order and where no character is part of 3 strings.

Hidden Words

Problem

- Given n strings s_1 ... s_n.
- Find a string S that contains all n strings in consecutive order and where no character is part of 3 strings.

Solution

- Start with two first strings s_1 and s_2.

Hidden Words

Problem

- Given n strings s_1 ... s_n.
- Find a string S that contains all n strings in consecutive order and where no character is part of 3 strings.

Solution

- Start with two first strings s_1 and s_2.
- If s_2 starts with s_1:

Hidden Words

Problem

- Given n strings $s_1 \dots s_n$.
- Find a string S that contains all n strings in consecutive order and where no character is part of 3 strings.

Solution

- Start with two first strings s_1 and s_2.
- If s_2 starts with s_1:
- Add s_1 to S, continue with rest of s_2 and next string.

Hidden Words

Problem

- Given n strings s_1 ... s_n.
- Find a string S that contains all n strings in consecutive order and where no character is part of 3 strings.

Solution

- Start with two first strings s_1 and s_2.
- If s_2 starts with s_1:
- Add s_1 to S, continue with rest of s_2 and next string.
- Else: remove first letter of s_1 and repeat process.

Jolly Fishing

Problem

- Choose if fishers are allowed to fish on each day of the year.
- Fishes will only reproduce when not being fished.
- After the year, there need to be at least as many fishes as in the beginning.

Jolly Fishing

Problem

- Choose if fishers are allowed to fish on each day of the year.
- Fishes will only reproduce when not being fished.
- After the year, there need to be at least as many fishes as in the beginning.

Solution

- It is always better to not allow fishing for the first part of the year and then allow fishing for the rest of the year.

Jolly Fishing

Problem

- Choose if fishers are allowed to fish on each day of the year.
- Fishes will only reproduce when not being fished.
- After the year, there need to be at least as many fishes as in the beginning.

Solution

- It is always better to not allow fishing for the first part of the year and then allow fishing for the rest of the year.
- (If we did not allow fishing after a day where we did, we would get a higher score by swapping the two days).

Jolly Fishing

Problem

- Choose if fishers are allowed to fish on each day of the year.
- Fishes will only reproduce when not being fished.
- After the year, there need to be at least as many fishes as in the beginning.

Solution

- It is always better to not allow fishing for the first part of the year and then allow fishing for the rest of the year.
- (If we did not allow fishing after a day where we did, we would get a higher score by swapping the two days).
- Thus we only need to determine the day we start allowing fishing.

Jolly Fishing

Problem

- Choose if fishers are allowed to fish on each day of the year.
- Fishes will only reproduce when not being fished.
- After the year, there need to be at least as many fishes as in the beginning.

Solution

- It is always better to not allow fishing for the first part of the year and then allow fishing for the rest of the year.
- (If we did not allow fishing after a day where we did, we would get a higher score by swapping the two days).
- Thus we only need to determine the day we start allowing fishing.
- Can simulate every 365 possible days (or use Ternary Search).

Extravagant Voyage

Problem

- Given n items with happiness H and volume V.
- Choose items with cumulative weight w.
- Also called 0-1-Knapsack.

Extravagant Voyage

Problem

- Given n items with happiness H and volume V.
- Choose items with cumulative weight w.
- Also called 0-1-Knapsack.

Solution

- Recursive DP solution:
- Go through n items and start with remaining weight $r = w$.

Extravagant Voyage

Problem

- Given n items with happiness H and volume V.
- Choose items with cumulative weight w.
- Also called 0-1-Knapsack.

Solution

- Recursive DP solution:
- Go through n items and start with remaining weight $r = w$.
- Recursively solve:

Extravagant Voyage

Problem

- Given n items with happiness H and volume V.
- Choose items with cumulative weight w.
- Also called 0-1-Knapsack.

Solution

- Recursive DP solution:
- Go through n items and start with remaining weight $r = w$.
- Recursively solve:
  - taking item: r -= W_i; h += H_i.

Extravagant Voyage

Problem

- Given n items with happiness H and volume V.
- Choose items with cumulative weight w.
- Also called 0-1-Knapsack.

Solution

- Recursive DP solution:
- Go through n items and start with remaining weight $r = w$.
- Recursively solve:
  - taking item: r -= W_i; h += H_i.
  - leaving item: r, h unchanged.

Extravagant Voyage

Problem

- Given n items with happiness H and volume V.
- Choose items with cumulative weight w.
- Also called 0-1-Knapsack.

Solution

- Recursive DP solution:
- Go through n items and start with remaining weight $r = w$.
- Recursively solve:
  - taking item: r -= W_i; h += H_i.
  - leaving item: r, h unchanged.
- Save states in dp-table.

Extravagant Voyage

Problem

- Given n items with happiness H and volume V.
- Choose items with cumulative weight w.
- Also called 0-1-Knapsack.

Solution

- Recursive DP solution:
- Go through n items and start with remaining weight $r = w$.
- Recursively solve:
  - taking item: r -= W_i; $h += H_i$.
  - leaving item: r, h unchanged.
- Save states in dp-table.

Gotchas

- Not using a dp-table results in time limit exceeded.

Problem

- Given different train connections, find the shortest time to get from Rostock to the given city.

Going Home

Problem

- Given different train connections, find the shortest time to get from Rostock to the given city.

Solution

- Can be solved using any shortest path algorithms that allows for edge weights e.g. Dijkstra.

Going Home

Problem

- Given different train connections, find the shortest time to get from Rostock to the given city.

Solution

- Can be solved using any shortest path algorithms that allows for edge weights e.g. Dijkstra.

Gotchas

- The input is rather complicated and has to be parsed into a graph structure first.

Going Home

Problem

- Given different train connections, find the shortest time to get from Rostock to the given city.

Solution

- Can be solved using any shortest path algorithms that allows for edge weights e.g. Dijkstra.

Gotchas

- The input is rather complicated and has to be parsed into a graph structure first.
- Need to find the right time to take a connection which is driven multiple times.

Keyboard Robot

Problem

- Given a 6x6 keyboard layout of letters and some text.
- Find a way to move 2 fingers simultaneously such that the time is minimal to type the given text.

Keyboard Robot

Problem

- Given a 6x6 keyboard layout of letters and some text.
- Find a way to move 2 fingers simultaneously such that the time is minimal to type the given text.

Insights

- Insight $\#1$: the text is short, only 200 letters, so the maximum time is $(5 + 5) \cdot 200 = 2000$

Keyboard Robot

Problem

- Given a 6x6 keyboard layout of letters and some text.
- Find a way to move 2 fingers simultaneously such that the time is minimal to type the given text.

Insights

- Insight #1: the text is short, only 200 letters, so the maximum time is $(5 + 5) \cdot 200 = 2000$
- Insight #2: we can simulate it, but need fast way of prioritising interesting states

Keyboard Robot

Problem

- Given a 6x6 keyboard layout of letters and some text.
- Find a way to move 2 fingers simultaneously such that the time is minimal to type the given text.

Insights

- Insight #1: the text is short, only 200 letters, so the maximum time is $(5+5) \cdot 200 = 2000$
- Insight #2: we can simulate it, but need fast way of prioritising interesting states
- Insight #3: grid is unhelpful, save as basic graph instead:
  `dist[(from_pos, to_pos)] = distance`
  `letter_to_pos[letter] = pos`

Keyboard Robot

Solution

- Use a priority queue to track every "reasonable" state

Keyboard Robot

Solution

- Use a priority queue to track every "reasonable" state
- Initial state is `(0, 0, (0, 0), 0, (0, 0), 0)`

Keyboard Robot

Solution

- Use a priority queue to track every "reasonable" state
- Initial state is `(0, 0, (0, 0), 0, (0, 0), 0)`
- A state is `(time, index, pos1, rem1, pos2, rem2)`
  1. *time* is the time since start of simulation
  2. *index* is the index of the current letter to be typed in the text
  3. pos_i is the position as a tuple of the i-th finger
  4. rem_i is the remaining time to move for the i-th finger (if negative it means it has been idle for some time and can be moved retroactively)

Keyboard Robot

Solution

- Use a priority queue to track every "reasonable" state
- Initial state is `(0, 0, (0, 0), 0, (0, 0), 0)`
- A state is `(time, index, pos1, rem1, pos2, rem2)`
  1. *time* is the time since start of simulation
  2. *index* is the index of the current letter to be typed in the text
  3. pos_i is the position as a tuple of the i-th finger
  4. rem_i is the remaining time to move for the i-th finger (if negative it means it has been idle for some time and can be moved retroactively)
- From every state put 2 new states inside the priority queue: what if either finger 1 or finger 2 moves to the next letter

Solution

- Use a priority queue to track every "reasonable" state
- Initial state is `(0, 0, (0, 0), 0, (0, 0), 0)`
- A state is `(time, index, pos1, rem1, pos2, rem2)`
  1. *time* is the time since start of simulation
  2. *index* is the index of the current letter to be typed in the text
  3. pos_i is the position as a tuple of the i-th finger
  4. rem_i is the remaining time to move for the i-th finger (if negative it means it has been idle for some time and can be moved retroactively)
- From every state put 2 new states inside the priority queue: what if either finger 1 or finger 2 moves to the next letter
- Only states where either $rem1$ or $rem2$ are 0 should be put in the priority queue

Keyboard Robot

Solution

- Use a priority queue to track every "reasonable" state
- Initial state is `(0, 0, (0, 0), 0, (0, 0), 0)`
- A state is `(time, index, pos1, rem1, pos2, rem2)`
  1. *time* is the time since start of simulation
  2. *index* is the index of the current letter to be typed in the text
  3. pos_i is the position as a tuple of the i-th finger
  4. rem_i is the remaining time to move for the i-th finger (if negative it means it has been idle for some time and can be moved retroactively)
- From every state put 2 new states inside the priority queue: what if either finger 1 or finger 2 moves to the next letter
- Only states where either $rem1$ or $rem2$ are 0 should be put in the priority queue
- Simulate until some index is at the end of the char sequence