# Exercises: Problem Solving

This document defines the **homework** assignments for the ["Algorithms" course @ Software University](https://softuni.bg/opencourses/algorithms).

# Parking Zones

You want to find the best place to park your car, in order to quickly go to the nearby store. We are given **n** parking zones, each parking zone has **x** and **y** (the coordinates of the top left corner of the parking zone), **width**, **height** and a price **p –** the price for **a minute stay** at the zone. We are also provided **s** free parking spaces in the form of their top left **x** and **y** coordinates, every parking space belonging to a parking zone. We also know that each block on the coordinate system is a square with dimensions exactly **1 x 1**. Additionally, we are given a target block **t** (given as its top left **x** and **y** coordinates)and a constant **k** which tells us the number of seconds needed to traverse **one block** (1 unit on the coordinate system). For the purposes of calculating the distance to **t**, movement is considered possible only in the four **major directions** (north, south, east, west), **moving diagonally is** **NOT allowed**.

The goal is to find the parking space, from which the journey to the target block **t** and backcosts the **least amount of money**, if multiple such spaces exist then we are to find the one that takes us the **least amount of time** to reach **t.** For the purpose of calculating the price for the stay, the minutes needed to reach the point and come back have to be **rounded up**, because for 2 minutes and 5 seconds we are taxed for 3 minutes.

### Input

* On the **first line,** we receive the number **n** that shows us the number of parking zones that we will receive.
* On the next **n** lines, we receive the zones in the format "**[Name of Zone]: [x], [y], [width], [height], [price/min]**"
* On the next line, we receive a list of the free parking spots in the format "**x, y**" separated by "**;**"
* On the next line, we receive the target point **t**, again in format "**x, y**"
* On the last line, the constant **k** is given, showing us **the time in seconds needed to traverse a single block** on the coordinate system.

### Output

* On a single line print the string “**Zone Type: [Zone Name]; X: [x]; Y: [y]; Price: [Price]**”, where **[Zone Name]** is the name of the zone in which the parking space with the best price and time to **t** is, **[x]** and **[y]** are the top left coordinates of the parking space and **[price]** is the price we’ll have to pay for the stay **formatted to 2 decimal places** **(pay attention that the number of minutes needed to reach the point and come back has to be rounded up, because for 2 minutes and 5 seconds we are taxed for 3 minutes).**

### Constraints

* Each parking space will always be part of a defined parking zone.
* Parking zones will never overlap.
* There will never be 2 parking spaces with equal distance from the target in the same zone.
* There will never be 2 parking spaces with the best price and time from the target point.
* The number of zones **n** will always be a whole number in the range **[1…10].**
* The price for staying a minute at a parking zone **p** will always be a real number in the range **[1…1000].**
* All input values except **p** and **n** will always be integers in the range **[1…1 000 000].**
* Allowed time: **100 ms**. Allowed Memory: **16 MB**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| **2 Blue: 0, 0, 10, 10, 1.0 Green: 0, 10, 10, 10, 0.5 1, 1; 5, 5; 1, 11; 5, 16 15, 10 20** | **Zone Type: Green; X: 1; Y: 11; Price: 5.00** |
| **Comments** | |
| We find the closest parking spot to t from each zone.  For the **Blue zone,** this is the parking spot with coordinates **5, 5**. The total blocks we need to walk from it to the target point is **14**. We multiply the blocks by **2** to get the number of blocks needed **to reach t and return**, then we multiply by the time needed for each block **k** (in this case **20**) to receive the number of seconds that we have to stay. **14 \* 2 \* 20 = 560** / 60 (to transform the seconds into minutes) = **9 minutes and 20 seconds** (since we crossed the 9 minutes’ mark, we need to pay for 10 minutes). 10 \* price **p** for stay/min **1** = **10lv.**  For the **Green zone,** this is the parking spot with coordinates **1, 11** (we don't check **[1,1]** and **[5, 16]** because the distance from them is obviously bigger). We do the same calculations, the total number of blocks is again **14**, so we calculate the time **14** **\*** **2** for a round trip **\* 20** for the number of seconds = **560** / 60 **= 9m20s**, which we round up to **10**, this time however the price per minute for the zone is lower, so we get **10 \* 0.5** (p) **= 5lv**. | |

# Parentheses

Given **N pairs** of parentheses, write a program to generate all combinations of well-formed parentheses.

### Input

* On the **first line,** we receive the number **N** that shows us the number of pairs of brackets.

### Output

* Print all correctly formed parentheses. The order of output does not matter.

### Constraints

* **N** will be in the range **[1…20].**
* Allowed time: **150 ms**. Allowed Memory: **32 MB**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| **3** | **((()))**  **(()())**  **(())()**  **()(())**  **()()()** |
| **2** | **(())**  **()()** |

# Towns

You are given **N** towns and you should find a path in them that satisfies few conditions. In each of the towns there are fixed number of **citizens** (between 1 and 1,000,000,000, inclusive).

The towns can be visited **only in the order they are given** on the console. You are not obligated to start the traversing from the first town. You can also **skip some of the towns** but you are not allowed to visit a town more than once.

For example if we have towns Sofia, Plovdiv, Varna and Burgas **you can visit:**

* Sofia, Varna, Burgas (by skipping Plovdiv)
* Plovdiv, Burgas (by starting from Plovdiv and skipping Varna)
* Sofia, Plovdiv, Varna, Burgas (by visiting every town)
* etc.

You are **not allowed** to visit the towns in wrong order. For example:

* Varna, Sofia is **not** allowed because Varna should be visited after Sofia in the given order
* Sofia, Varna, Sofia is **not** allowed because you are visiting Sofia for a second time
* etc.

You should satisfy **one more condition** when traversing towns. If you divide the path into 2 sub-paths that share one common town (part of the original path) such that the last town of the first path is the first town of the second path, then all number of citizens in the first path should be in ascending order and all number of citizens (increasing number of citizens) in the second path should be in descending order (decreasing number of citizens). Yeah.

For example look at the first example below. The path Pleven, Burgas, Varna, Sofia, Ruse, StaraZagora satisfies this condition (in fact, it satisfies all conditions) because we can divide the path into 2 sub-paths (first: Pleven, Burgas, Varna, Sofia; second: Sofia, Ruse, StaraZagora) and the first path has increasing number of citizens and the second path has decreasing number of citizens.

Write a program that finds **the longest path that satisfies the given conditions**.

### Input

* The input data should be read from the console.
* The number of towns **N** will be given on the first input line.
* On each of the next **N** lines there will be information for one of the towns in the order they can be traversed. If you split the line by a single space (‘ ‘) the first part of the line will contain the number of citizens in the town, and the second part of the line will contain the name of the town. See the examples below.
* The input data will always be valid and in the format described. There is no need to check it explicitly.

### Output

* The output data should be printed on the console.
* On the only output line write the longest path that satisfies the given conditions.

### Constraints

* **N** will be an integer between **1** and **1000**, inclusive.
* The name of each town will contain only Latin letters (both lowercase (‘a’-’z’) and uppercase (‘A’-’Z’) are allowed)
* The length of each town name will be between **1** and **20** characters.
* Each town name will be unique.
* Allowed working time for your program: **100 ms**. Allowed memory: **16 MB**.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Explanation** |
| 8  108214 Pleven  339077 Plovdiv  200612 Burgas  334688 Varna  1241396 Sofia  92162 Sliven  151951 Ruse  137907 StaraZagora | 6 | The longest path that satisfies all conditions contains these 6 cities:  Pleven (first town)  Burgas (bigger than Pleven)  Varna (bigger than Burgas)  Sofia (bigger than Varna)  Ruse (smaller than Sofia)  StaraZagora (smaller than Ruse)  (The path can be cut in 2 paths satisfying citizens count condition. Sofia is the town that divides the path into 2 other paths) |
| 4  19906 NikiTown  19832 EvlogiTown  19894 IvoTown  19896 DonchoTown | 3 | You can start from EvlogiTown, then go in IvoTown and then DonchoTown.  If we divide the path into 2 sub-paths  • EvlogiTown, IvoTown and DonchoTown  • DonchoTown  They satisfy the ascending-descending condition. |
| 3  2 HaxorTown  2 LeetTown  2 RoxxorTown | 1 | You can visit any of the 3 towns, but cannot move any further because of the citizens count traversing requirements |

# Shop Keeper

You are the shop keeper of a store, you will be given a sequence of numbers - **S** representing the **products** **you have in stock** (each number representing a certain product ex. 1 = bread, 2 = flour and so on). You will also be given a **list of orders** placed by customers – **O**, represented as a sequence of numbers (each number in the list represents an order for a product of that type ex. 1 = a customer has ordered a bread, 2 = customer has ordered flour and so on). Fulfilling an order does not remove the desired product from your stock (imagine we have infinite amounts of all products in stock). After each order, you are allowed to **change a single product** in your stock to any product you wish, however the total amount of product types in your stock should remain constant. Knowing all orders in advance, your task is to calculate the **minimum amount of changes** you need to do in your stock, in order to service all of them.

### Input

* On the first line, we receive the sequence **S**, each element of **S** will be separated from the others by a single space.
* On the second line, we receive the list of orders **O**, each element of **O** will be separated from the others by a single space.

### Output

* On a single line print the **minimum number of changes needed to service all orders**, if it's impossible to service all of them write "**impossible**" instead.

### Constraints

* The possible product types will be integer numbers in the range **[1…3000].**
* The sequence of products in stock - **S** will contain only **unique** products.
* The sequence of the products in stock - **S** will contain between **[1…1500]** products.
* The sequence of the orders - **O** will contain between **[1…12000]** orders.
* Allowed time: **140 ms**. Allowed Memory: **32 MB**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 2 3 4  1 2 2 5 1 3 | 1 |
| **Comments** | |
| We start with item types 1, 2, 3 and 4 in stock, the first order comes and it wants an item type **1**, checking our stock we have the item so we fulfill the order. After fulfilling an order we have the option to change an item type in our stock, looking ahead in the list of orders we see that the next order is for an item type **2**, as we already have that item type in stock we decide not to change anything.  **After Order 1 -> No exchange**  The second order comes in, since we have **2** in stock, we fulfill that order. We are again offered the chance to change an item type in stock, looking ahead we see the next order is for a **2** as well, having the **2** in stock we decide not to change anything again.  **After Order 2 -> No exchange**  The third order comes in, it is for item type **2** again, so we fulfill it. Looking ahead in the order list we see the next order is for a **5**, since we don't have it in stock we have to exchange an item type for it. In this case we have two good choices, we can either change the **2** or the **4**, as we won't need them anymore. We arbitrarily choose to exchange the **2**.  **After Order 3 -> Exchange 2 for 5**.  **Stock after exchange:** **1 5 3 4**  The fourth order comes – it's for a **5**, since we exchanged the **2** for it we manage to service the order. Looking ahead, the next item is a **1**, since we have it we don't change anything.  **After Order 4 -> No exchange**  The fifth order comes in for a **1**, we have it in stock so we fulfill it. Looking ahead the last order is for a **3**, as we also have that in stock we skip any changes again.  **After Order 5 -> No exchange**  The sixth and last order comes in for a **3**, we have it in stock so we fulfill it. Having fulfilled all orders we write the number of changes we did **1**. | |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 3 1 8  8 10 13 13 5 3 1 13 3 8 14 10 | 5 |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 6 7 8  1 6 6 6 8 | impossible |
| **Comments** | |
| We take a look at the first order – 1, we don't have it in stock so we immediately fail the order, thus we print "**impossible**". | |