

Exam 2022 January

- [Exam 2022 January](#)
 - [Problems](#)
 - [Problem 1: Line - Graph traversal](#)
 - [Problem 2: Pip price - Greedy](#)
 - [Problem 3: Red and white - Flow](#)
 - [Problem 4: Sale - NP-hard](#)
 - [Problem 5: Stack - Dynamic programming](#)
 - [Counterexample](#)
 - [Greedy](#)
 - [Graph Traversal](#)
 - [Dynamic programming](#)
 - [Flow](#)
 - [NP-hard](#)

Problems

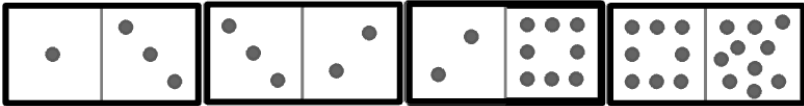
Problem 1: Line - Graph traversal

Line

Problem ID: line

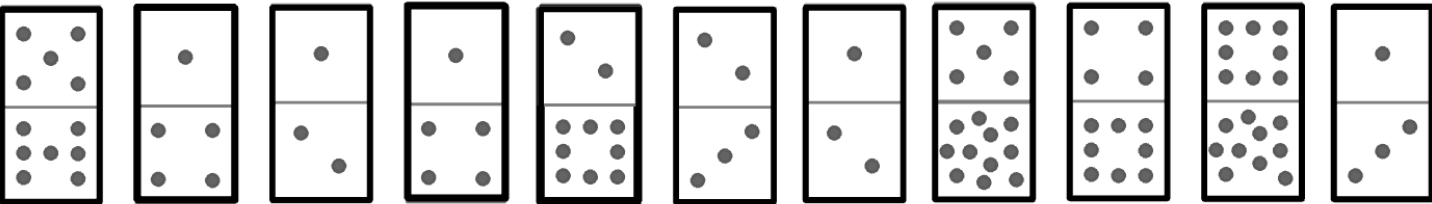
A generalised domino tile (henceforth, *tile*) is a rectangular tile displaying two nonzero positive numbers (a, b) , not necessarily different. Tiles display these numbers as small dots called *pips*.

Horizontally oriented tiles can be arranged into a *line* if the number of pips on their adjacent sides match. For instance, here is a line of four tiles:

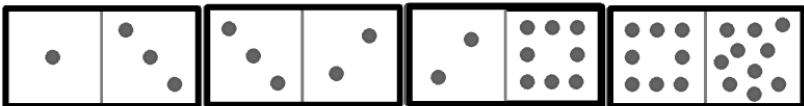


Given T tiles, we want to form a line that begins with a tile whose left side shows 1 pip and ends with a tile whose right side shows the largest possible number of pips.

For instance, if we are given these tiles:



then we can reach a tile containing a side with 10 pips like this:



(Note that we don't care about the number of tiles used – in fact, there is another valid solution reaching 10 that uses only three tiles instead of four.)

Tiles cannot be reused (but there can be duplicate tiles in the input).

Input

On the first line, the number T of tiles.

Then follow T lines, one for every tile. Each of these lines contains two integers a b , which describes a tile containing the numbers a and b .

There is at least one tile with $a = 1$ or $b = 1$. There can be duplicate tiles, and a can be equal to b .

Output

Print a single integer r : the largest number of pips that can appear in a line starting from a 1-pip tile.

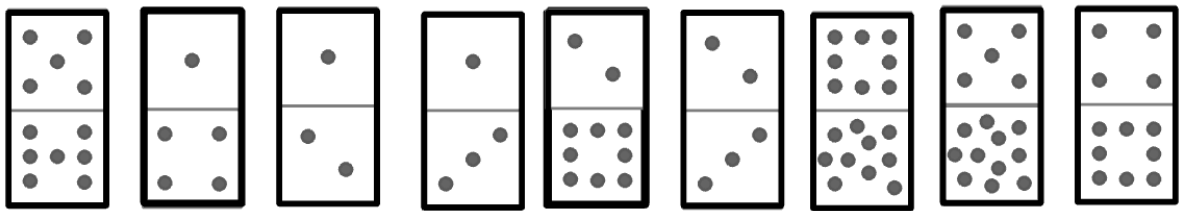
Sample Input 1	Sample Output 1
11 5 7 1 4 1 2 1 4 2 8 2 3 1 2 5 11 4 8 8 10 1 3	10

Problem 2: Pip price - Greedy

Pip price
Problem ID: pipprice

A generalised domino tile (henceforth, *tile*) is a rectangular tile displaying two nonzero positive numbers (a, b) , not necessarily different. Tiles display these numbers as small dots called *pips*.

The price of a tile is the total number $a + b$ of pips on it. For instance, the tile $(3, 7)$ costs 10 kr. You have k kr. and want to buy as many tiles as you can from T given tiles. For example, if you have 16 kr. then you can buy 3 of the following tiles:



namely the tiles $(1, 4)$, $(2, 3)$ and $(1, 2)$ (which cost $1 + 4 + 2 + 3 + 1 + 2 = 13 \leq 16$ kr.).

Input

On the first line, the number T of tiles and your money k .
Then follow T lines, one for every tile. Each of these lines contains two integers a b , which describes a tile containing the numbers a and b .
There can be duplicate tiles, and a can be equal to b .

Output

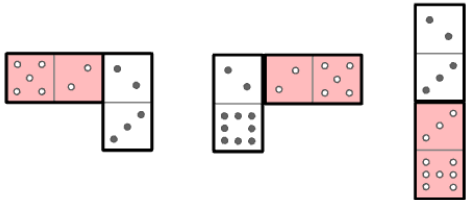
Print a single integer r : the number of tiles you can buy.

Sample Input 1	Sample Output 1
9 16 5 7 1 4 1 2 1 3 2 8 2 3 8 10 5 11 4 8	3
Sample Input 2	Sample Output 2
2 10 1 3 2 2	2
Sample Input 3	Sample Output 3
1 10 6 7	0

Problem 3: Red and white - Flow

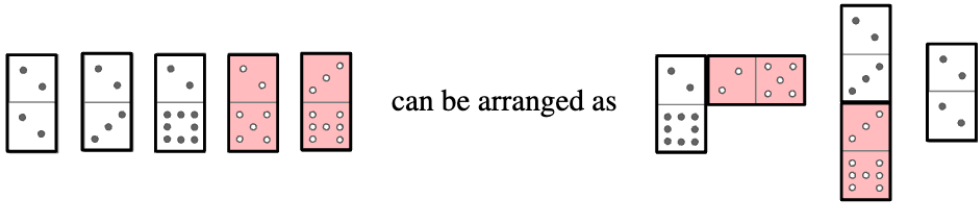
Red and white
Problem ID: redandwhite

A generalised domino tile (henceforth, *tile*) is a rectangular tile displaying two nonzero positive numbers (a, b) , not necessarily different. Tiles display these numbers as small dots called *pips* and come in two colours, red and white. Two tiles of different colours form a *couplet* if they can be placed next to each other such that the numbers agree along the shared edge. Here are three examples of couplets:

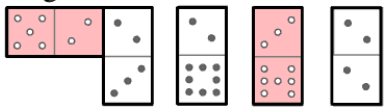


It does not matter if the two tiles in a couplet are arranged horizontally, vertically, form an L-shape, etc. No couplet can consist of more than two tiles.

Given some tiles, how many couplets can be created by an optimal arrangement?
For instance, the five tiles



forming two couplets.
Note the following less-than-optimal arrangement of the same tiles that achieves only one couplet:



Input

On the first line, the number T of tiles. Then follow T lines, one for every tile. Each of these lines contains three values $a\ b\ c$, which describes a tile containing the numbers a and b of colour c , where $c \in \{R, W\}$ (for ‘red’ and ‘white’, respectively). There can be duplicate tiles, and a can be equal to b .

Output

Print a single integer r : the maximum number of couplets that can be achieved.

Sample Input 1	Sample Output 1
5 2 2 W 2 3 W 2 8 W 2 5 R 3 7 R	2
Sample Input 2	Sample Output 2
4 2 3 W 3 8 W 2 7 R 2 9 R	1

Problem 4: Sale - NP-hard

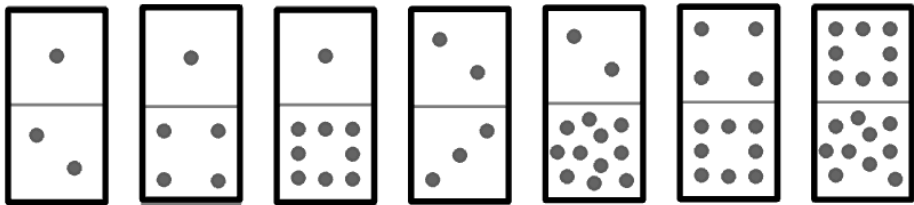
Sale

Problem ID: sale

A generalised domino tile (henceforth, *tile*) is a rectangular tile displaying two nonzero positive numbers (a, b) , not necessarily different.

The Domino Tile Shop is having a sale! For as little as 1 kr. you can buy *all* the tiles containing a specific value, for instance, *all* tiles containing a 4.

For example, you can buy all these tiles:



for only 3 kr. by bying all tiles containing a 2, all tiles containing a 4, and all tiles containing an 8.
Given T tiles, how many kroner do you need to buy all of them?

Input

On the first line, the number T of tiles.
Then follow T lines, one for every tile. Each of these lines contains two integers $a\ b$ with $1 \leq a \leq b$, which describes a tile containing the numbers a and b .
There can be duplicate tiles, and a can be equal to b .

Output

Print a single integer k : the smallest cost of buying all tiles. Then print k values p_1, \dots, p_k : the number on the tiles that you buy. To be precise, if you buy all tiles containing p_1 , then all tiles containing p_2 among the remaining tiles, and so on up to p_k , then you will have bought all T tiles.

Sample Input 1	Sample Output 1
7 1 2 1 4 1 8 2 3 2 11 4 8 8 11	3 1 2 8
Sample Input 2	Sample Output 2
6 1 2 1 3 1 4 2 5 3 6 4 7	3 2 3 4
Sample Input 3	Sample Output 3
3 5 5 5 5 5 5	1 5

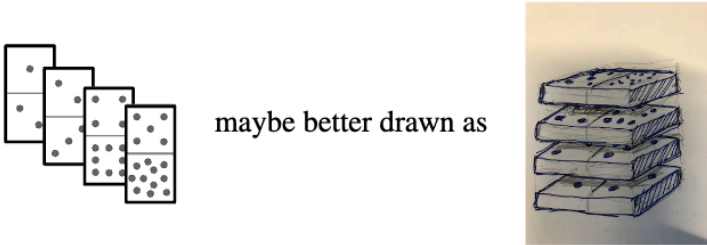
Problem 5: Stack - Dynamic programming

Stack

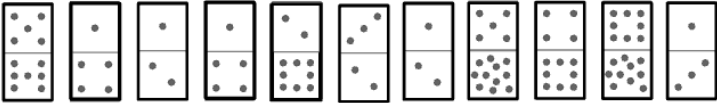
Problem ID: stack

A generalised domino tile (henceforth, *tile*) is a rectangular tile displaying two nonzero positive numbers (a, b) , not necessarily different. Tiles display these numbers as small dots called *pips*.

Tiles can be *stacked* if they are “strictly increasing on both sides” in the following sense: If tile (a, b) is placed below tile (a', b') then we require both $a < a'$ and $b < b'$ or, by turning the topmost tile, $a < b'$ and $b < a'$. For legibility, I’ve drawn a stack below by making the rectangles slightly overlap, but you should think of the four tiles as being neatly stacked exactly on top of each other. For instance, the 4-pip part is exactly below the 5-pip part of the topmost tile:



Given T tiles, what is the highest stack you can build? For instance, if we are given these tiles:



then we can build the stack consisting of $(1, 2)$, $(2, 3)$, $(4, 8)$ and $(5, 11)$ shown in the topmost illustration. (Note that we’ve turned the $(3, 2)$ -tile.) Another way of building a height-4 stack is $(1, 2)$, $(2, 3)$, $(4, 8)$, $(8, 10)$.

Input

On the first line, the number T of tiles. Then follow T lines, one for every tile. Each of these lines contains two integers $a\ b$, which describes a tile containing numbers a and b . There can be duplicate tiles, and a can be equal to b .

Output

Print a single integer r : the highest stack (measured in number of tiles) that can be built.

Sample Input 1	Sample Output 1
11 5 7 1 4 1 2 1 4 2 8 3 2 1 2 5 11 4 8 8 10 1 3	4

Sample Input 2	Sample Output 2
5 1 1 5 5 10 10 207 6 208 7	4

1. Counterexample

Looking through the problems on pages 3–7, as soon as Gordon Gecko reads the description of problem “Sale”, he is sure it can be solved by a greedy algorithm.

His rough idea is the following: “First determine which value x appears most often on the tiles.
(Careful with tiles of the form x, x ; you want to count them only once.) Now buy all tiles containing this maximally frequent value x , which costs 1 kr. in total.
Continue doing this until there are no more tiles left.”

It’s clear that Gordon hasn’t really thought a lot about data structures or running times, but you could probably fix that for him. However, his idea is wrong on a more fundamental level.

1.a (3 pt.)

Give a concrete, complete instance (either as a cute drawing of tiles or in the input format specified in Sale) on which Gordon’s algorithm fails to find an optimal solution. Specify a nonoptimal solution that would be found by Gordon’s algorithm, and what an optimal solution would be instead.

1.a - Answer

—	—	—	—	—	—	—
4	1	1	1	2	3	
4	4	3	2	2	3	
—	—	—	—	—	—	—

In the above answer Gordon Geckos algorithm would identify that there are:

Tiles	Count
1	3
2	2
3	2
4	2

Thus his algorithm would buy the tiles in the following order:

1. Buy 1 - leaving us with

—	—	—
4	2	3
4	2	3
—	—	—

2. Buy 2,3, and 4 in any order

Thus resulting in a total cost of 4 kr.

The optimal solution would be to buy the tiles 2, 3, and 4 (in any order) first.
This would thus as result also buy the 1 tiles without any additional cost.
Thus resulting in a total cost of 3 kr.

2. Greedy

One of the problems on pages 3–7 can be solved by a simple greedy algorithm.

2.a (1 pt.)

Which one?

2.a - Answer

"Pip price" can be solved by a simple greedy algorithm.

2.b (2 pt.)

Describe the algorithm, for example by writing it in pseudocode. (Ignore parsing the input.) You probably want to process the input in some order; be sure to make it clear which order this is (increasing or decreasing order of start time, alphabetic, colour, age, size, x-coordinate, distance, number of neighbours, scariness, etc.). In other words, don't just write "sort the input."

2.b - Answer

```
// List of the summed cost of each tile
// So a tile with pips 1 and 2 would be inserted as 3
T = [t_1, t_2, ..., t_n] // List of tiles

// Sort so cheapest tiles are first
// O(n log n)
T.sortAscendingOrder

k = money available
boughtTiles = 0

// O(n)
foreach (cost in T) {
    if (k < 0) return boughtTiles
    k -= cost
}
```

2.c (1 pt.)

State the running time of your algorithm in terms of the input parameters. (It must be polynomial in the input size.)

2.c

The algorithm sorts the tiles in $O(n \log n)$ time and then iterates over the tiles in $O(n)$ time.
Thus the total running time of the algorithm is:

$$O(n \log n)$$

3. Graph Traversal

One of the problems on pages 3–7 can be efficiently solved using (possibly several applications of) standard graph traversal methods (such as breadth-first search, depth-first search, shortest paths, connected components, spanning trees, etc.), and without using more advanced design paradigms such as dynamic programming or network flows.

3.a (1 pt.)

Which one?

3.a - Answer

"Line" can be solved with a standard graph traversal method, such as DFS or BFS.

3.b (2 pt.)

Describe your algorithm. As much as you can, make use of known algorithms. (For instance, don't reinvent a well-known algorithm. Instead, write something like "I will use Blabla's algorithm [KT, p. 342] to find a blabla in the blabla.")

3.b - Answer

I will use the BFS algorithm to find a line from a tile with pip 1 to the tile with the maximum pip value.

We thus create a graph G with each pip number v connected with e to the pip number is written in the input.

This means we do not consider a tile as a node, but instead the pip number on the tile as a node.

We can now run a standard BFS from pip 1.

We should keep track of the which maximum pip value we have seen so far.

Then when we have visited every $v \in V$ we can return the max pip value we have seen.

3.c (1 pt.)

State the running time of your algorithm in terms of the parameters of the input.

3.c - Answer

As we use a standard BFS algorithm the running time of the algorithm is:

$$O(V + E)$$

4. Dynamic programming

One of the problems on pages 3–7 is solved by dynamic programming.

4.a (1 pt.)

Which one?

4.a - Answer

"Stack" can be solved by dynamic programming.

4.b (3 pt.)

Following the book's notation, let $\text{OPT}(\dots)$ denote the value of a partial solution. (Maybe you need more than one parameter, like $\text{OPT}(i, v)$. Who knows?)

Anyway, tell me what the parameters are—vertices, lengths, etc. and what their range is.

Use words like "where $i \in \{1, \dots, k^2\}$ denotes the length of BLABLA" or "where $v \in R$ is a red vertex."

Give a recurrence relation for OPT , including relevant boundary conditions and base cases.

Which values of OPT are used to answer the problem?

4.b - Answer

Assume all tiles (a, b) such that $a \leq b$.

Make DAG (Directed Acyclic Graph) from tiles, where:

- $V = \{\text{tiles}\}$
- $(u, v) \in E$ if tile u can directly stack below tile v , satisfying:
 $(a_u < a_v \wedge b_u < b_v) \vee (a_u < b_v \wedge b_u < a_v)$

The tiles now have topological ordering (which means that v connected to v' by $(v, v') \in E$ implies that $v < v'$).

$\text{OPT}(v)$ denotes the highest possible stack height that can be achieved and v being the end node.

$$\text{OPT}(v) = \begin{cases} 1 & \text{if } v \text{ has no predecessors} \\ 1 + \max_{(u,v) \in E} \text{OPT}(u) & \text{otherwise} \end{cases}$$

Solution is thus:

$$\max_{v \in V} \text{OPT}(v)$$

4.c (1 pt.)

State the running time and space of the resulting algorithm in terms of the input parameters.

4.c - Answer

As we have to build a DAG (which can be represented as an adjacency list) the space complexity would be $O(T + E) \equiv O(T + E)$. But as we know that there is a maximum of $2T$ edges we can simplify to $O(T)$

Constructing G involves checking all pairs of tiles, resulting in $O(T^2)$ operations in the worst case.

Evaluating $\text{OPT}(v)$ for all tiles $v \in V$, iterating over incoming edges (u, v) , takes $O(T + T^2)$, where T accounts for nodes and T^2 for edges.

Thus results in a total time complexity of $O(T^2)$.

Space complexity: $O(T)$

Time complexity: $O(T^2)$

5. Flow

One of the problems on pages 3–7 is easily solved by a reduction to network flow.

5.a (1 pt.)

Which one?

5.a - Answer

"Red and white" can be solved by a reduction to network flow.

5.b (3 pt.)

Explain the reduction.

Start by drawing the graph corresponding to Sample Input 1.

Be ridiculously precise about which nodes and arcs there are, how many there are (in terms of size measures of the original problem), how the nodes are connected and directed, and what the capacities are.

Describe the reduction in general (use words like "every node corresponding to a giraffe is connected to every node corresponding to a letter by an undirected arc of capacity the length of the neck").

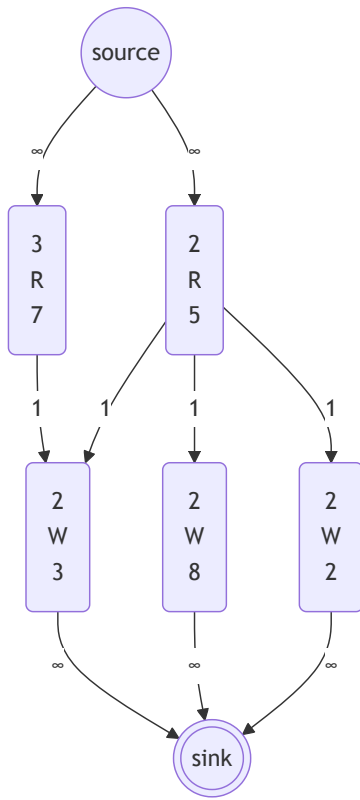
What does a maximum flow mean in terms of the original problem, and what size does it have in terms of the original parameters?

5.b - Answer

If we consider all tiles as vertices $v \in V$ and then create directed edges from the red to white, with capacity of 1, between all equal pips with different colours $e \in E$ if (v_1, v_2) where $v_1.\text{pips} = v_2.\text{pips} \wedge v_1.\text{colour} \neq v_2.\text{colour}$.

We also need to introduce a source and a sink node which respectively will connect to the red nodes and the white nodes and the edges will have a capacity of infinity.

Then we can create the following graph from **sample input 1**:



Now we can then run max flow on the graph.

The resulting flow will be the maximum number of couplets we can create.

5.c (1 pt.)

State the running time of the resulting algorithm, be precise about which flow algorithm you use.

(Use words like “Using Krampfmeier–Strumpfnudel’s algorithm ((5.47) in the textbook), the total running time will be $O(r^{17} \log^3 \epsilon + \log^2 k)$, where r is the number of frontozes and k denotes the maximal weight of a giraffe.”)

5.c - Answer

As we first have to find the max-flow we can use Ford-Fulkerson.

Specifically we can use the Edmonds-Karp algorithm.

Edmonds-Karp runs in $O(VE^2)$.

Dinitz’s algorithm further reduces this to $O(V^2E)$.

Thus for this problem we will have number of tiles T as nodes and number of edges as $2T$, as we can maximum have all red tiles connected to all white tiles and the source and sink connected to all red and white tiles.

Thus the running time of the algorithm is:

$$O(T^2)$$

6. NP-hard

One of the problems on pages 3–7 is NP-hard.

6.a (1 pt.)

Which problem is it? (Let’s call it P_1 .)

6.a - Answer

“Sale” is NP-hard.

Thus $P_1 = \text{Sale}$

6.b (1 pt.)

The easiest way to show that P_1 is NP-hard is to consider another well-known NP-hard problem (called P_2). Which?

6.b - Answer

We can consider the NP-hard problem "Set cover problem".

Thus $P_2 = \text{Set cover}$

6.c (0 pt.)

Do you now need to prove $P_1 \leq_p P_2$ or $P_2 \leq_p P_1$?

6.c - Answer

We need to prove that "Set cover", P_2 , can be reduced to "Sale", P_1 .

Thus that $P_2 \leq_p P_1$

6.d (3 pt.)

Describe the reduction.

Do this both in general and for a small but complete example.

In particular, be ridiculously precise about what instance is **given**, and what instance is **constructed** by the reduction, the parameters of the instance you produce (for example, number of vertices, edges, sets, colors) in terms of the parameters of the original instance, what the solution of the transformed instance means in terms of the original instance, etc.

For the love of all that is Good and Holy, please start your reduction with words like "Given an instance to BLABLA, we will construct an instance of BLABLA as follows."

6.d - Answer

Given "Set cover" instance $U = \{e_1, e_2, \dots, e_n\}$ and subsets S_1, S_2, \dots, S_m where U is the universe of elements and e_i is an element in the universe.

Goal is then to find the minimum number of subsets to cover the universe U .

Construct instance to "Sale" as follows:

- For each element $e_i \in U$ create a tile - this will be represented as a tuple $(e_i.a, e_i.b)$, where both $e_i.a$ and $e_i.b$ are placeholders for e_i itself (no actual splitting into separate values).
- For each subset S_i , create a subset of all tiles corresponding to elements $e_i \in S_i$.
- In sale select as few subsets $|S|$ as possible to cover all elements in U .

Example: $U = \{e_1, e_2, e_3, e_4\}$

$S_1 = \{e_1, e_2\}$, $S_2 = \{e_2, e_3\}$, $S_3 = \{e_3, e_4\}$

Tiles = e_1, e_2, e_3, e_4

Pips = $\{(e_1.a : 1), (e_1.b : 1), (e_2.a : 1), (e_2.b : 2), (e_3.a : 2), (e_3.b : 3), (e_4.a : 3), (e_4.b : 3)\}$

- **Tiles:** e_1, e_2, e_3, e_4
- **Pips:**

$\{(e_1.a : e_1, e_1.b : e_1), (e_2.a : e_2, e_2.b : e_2), (e_3.a : e_3, e_3.b : e_3), (e_4.a : e_4, e_4.b : e_4)\}$

The solution for this is to buy subsets S_1 and S_3 which will cover all elements in U .

Thus costing us 2 kr.

This reduction ensures $P_2 \leq_p P_1$