# Hands-on 3 Solution Tommaso Crocetti

## Problem #1, Holiday planning

For problem #1 I adopted a classical dynamic programming approach. The function holiday_**planning** firstly computes the prefix sums of the arrays given as parameters, since we are interested in knowing how many attractions we can visit up to i days for each city.

```
pub fn holiday_planning(n: usize, d: usize, iters: Vec<Vec<u32>>) -> u32
{
let mut prefixes = Vec::with_capacity(n);
    for iter in iters {
        let mut pref = vec![0; d + 1];
        pref[0] = 0;
        for i in 1..d + 1 {
            pref[i] = pref[i - 1] + iter[i - 1];
        }
        prefixes.push(pref);
    }
```

Then the function builds an n+1 x d+1 matrix, where each row represents the number of attractions that can be visited for each number of days from 0 to d, considering only the first i cities. So the first row is filled by 0. Since each column represents the number of attractions that can be visited up to j days, considering cities from 0 to n, even the first column is filled with zeros. The matrix is filled row by row, so consider a new city for each iteration. The subproblem represented by the cell dp[i][j] is the maximum number of attractions that can be visited with the first i cities for j days. So for the i-th city, we have 2 main choices: don't visit it or visit it. If we don't visit city i, we take the maximum number of attractions that can be visited with i-1 cities and up to j days (dp[i-1][j]). If we want to visit city i, we want to know for how many days we want to stay. So we have to consider the number of days we want to stay in city i, between 1 and j. Considering a generic number of days k, the solution is the number of attractions that we can visit considering i-1 cities for j – k days (dp[i-1][j-k]) plus the number of attractions that can be visited for k consecutive days in the city i (prefix[k]). Since we are interested in the maximum number of attractions, we must take the maximum value for this sum over all possible k. The final result is stored in the last cell of the matrix.

```
    let mut dp = vec![vec![0; d+1]; n+1];
```

```
    for i in 1..n+1 {
        for j in 1..d+1 {
            for k in 0..j+1 {
                dp[i][j] = dp[i][j].max(dp[i - 1][j - k] + prefixes[i -
1][k]);
            }
        }
    }
    dp[n][d]
```

The time complexity for this solution depends on the number of operations we must do, multiplied by the time necessary for each one. Since we have to fill a n+1 x d+1 matrix, we perform O(nd) operations. To fill a cell we look back to all the previous cells in the previous row, the worst case scenario is when we want to fill the last cell of a row, and we have to check the d+1 previous row's cell, so we spend O(d) operations for each cell. Final time complexity so is O(n(d^2)). Space complexity is O(nd), the space necessary to allocate the matrix.

# Problem #2, Design a  course

For this problem, I used an approach based on a BST to achieve an O(nlogn) time complexity. The BST data structure (called BSTree) is based on the tree structure used during the course but slightly modified.

## Node structure

A Node keeps track of various information:

- Key, the value that will be maintained ordered in the BSTree, for this problem it will be the beauty of a course.
- Value, it will represent the number of topics that can be selected up to <key> beauty.
- Id_left and id_right: indexes of children node, if any.
- Id_father: index of the father of the current node.

```
pub struct Node {
    key: u32,
    value: u32,
    id_left: Option<usize>,
    id_right: Option<usize>,
    id_father: Option<usize>,
```

```
}

impl Node {
    fn new(key: u32, value: u32, id_father: Option<usize>) -> Self {
        Self {
            key,
            value,
            id_left: None,
            id_right: None,
            id_father
        }
    }
}
```

A BSTree is just a vector of nodes, created by the **with_root** method that creates a Node with the specified key, starting value of 1, and no father id (since it's the root), and then initializes the Node vector with this node.

```
pub struct BSTree {
    nodes: Vec<Node>,
}

impl BSTree {

    pub fn with_root(key: u32) -> Self {
        Self {
            nodes: vec![Node::new(key, 1, None)],
        }
    }
```

The **add_node** method contains a little change: upon creation of the new Node, it uses the current parent_id to set the father_id of the Node.

To insert Node in the BSTree maintaining the ordering on the key value, I defined the functions **insert** and **rec_insert**

- **Insert**: the wrapper function, calls the recursive function on the root's index of the tree (Some(0)) and the key and value to be inserted.
- **Rec_insert**: the recursive function, given a node_id, the key, and value, checks if the key is smaller or equal to the current Node key, in this case, we want to insert in the left subtree, otherwise we insert in the right subtree.

The next step is to check if the current Node already has a left/right child: if yes, rec_insert is recursively called on the child, otherwise, the add_node method is called on the current Node's id.

```
    pub fn insert(&mut self, key: u32, value: u32) {
        self.rec_insert(Some(0), key, value);
    }

    fn rec_insert(&mut self, node_id: Option<usize>, key: u32, value:
u32) {
        if let Some(id) = node_id {
            if key <= self.nodes[id].key {
                if self.nodes[id].id_left == None {
                    self.add_node(id, key, value, true);
                } else {
                    self.rec_insert(self.nodes[id].id_left, key, value);
                }
            } else {
                if self.nodes[id].id_right == None {
                    self.add_node(id, key, value, false);
                } else {
                    self.rec_insert(self.nodes[id].id_right, key, value);
                }
            }
        }
```

An important operation required for this problem is the predecessor search, performed by the **predecessor** and **rec_predecessor** methods.

- **Predecessor**: the wrapper function, calls rec_predecessor on the root index, the key we want to know is the predecessor and the starting result, None.
- **Rec_predecessor**: the recursive function, the base case is a None Node index, for which it returns the computed predecessor, the recursive case is a valid id, so it checks if the current Node's key is smaller than the queried key, in this case the search continues in the right subtree, updating the recursive result to the current key, in the opposite case the search continues in the left subtree keeping the same result.

```
    pub fn predecessor(&self, key: u32) -> Option<usize> {
        self.rec_predecessor(Some(0), key, None)
    }
```

```
    fn rec_predecessor(&self, node_id: Option<usize>, key: u32,
current_pred: Option<usize>) -> Option<usize> {
        if let Some(id) = node_id {
            if key > self.nodes[id].key {
                return self.rec_predecessor(self.nodes[id].id_right, key,
Some(id));
            } else {
                return self.rec_predecessor(self.nodes[id].id_left, key,
current_pred);
            }
        }
        current_pred
    }
```

The next methods, get_max_value, and rec_get_max_value, are used to evaluate the maximum value present in a path node-root, given a certain maximum key.

- **Get_max_value**: the wrapper function, if the Node id is valid, it calls the recursive function passing the id, the maximum key, and the starting result, 0, otherwise the default result is 0**.**
- **Rec_get_max_value**: the recursive function, the base case is a None id, where it returns the current result, the recursive case is a valid Node id, so it compares the current key to the given key passed: if the current key is smaller, then it updates the current result to the maximum between the current result and the value of the current node, in the opposite case the result is unchanged. In both cases, this function calls itself on the father of the current Node.

```
    pub fn get_max_value(&self, node_id: Option<usize>, key: u32) -> u32
{
        if let Some(_id) = node_id {
            return self.rec_get_max_value(node_id, key, 0);
        }
        0
    }

    fn rec_get_max_value(&self, node_id: Option<usize>, key: u32, max:
u32) -> u32 {
        if let Some(id) = node_id {
            if key > self.nodes[id].key {
```

```
                return self.rec_get_max_value(self.nodes[id].id_father,
key, max.max(self.nodes[id].value));
            }
            return self.rec_get_max_value(self.nodes[id].id_father, key,
max);
        }
        max
    }
```

The last couple of methods needed are **update_max_value** and **rec_update_max_value**, which updates the values in a given node-root path of the BSTree.

- **Update_max_value**: the wrapper function, calls the recursive function on the index of the last inserted Node and its key and value.
- **Rec_update_max_value**: the recursive function, it updates the current Node value to the maximum between the current value and the given value if the current Node key is greater or equal to the given key, then it recurs on the father of the current Node.

```
    pub fn update_max_value(&mut self) {
        let key = self.nodes[self.nodes.len() - 1].key;
        let value = self.nodes[self.nodes.len() - 1].value;
        self.rec_update_max_value(Some(self.nodes.len() - 1), key,
value);
    }

    fn rec_update_max_value(&mut self, node_id: Option<usize>, key: u32,
value: u32) {
        if let Some(id) = node_id {
            if key <= self.nodes[id].key {
                self.nodes[id].value = (self.nodes[id].value).max(value);
            }
            self.rec_update_max_value(self.nodes[id].id_father, key,
value)
        }
    }
```

For all these four pairs of methods, time complexity is O(logn), since they perform a simple path root-node or node-root in the BST (considering a  balanced BST).

The function **design_a_course** uses a BSTree and its methods to solve problem #2. The first step is to sort the input array of the topics for increasing difficulty: in this way, we can consider only the beauty of the topics in the next phase since we will process the topics always with increasing difficulty. If some topics have the same difficulty, they are ordered by decreasing beauty, in this way, we can build a correct solution because the BSTree will store Node with the same key (beauty) in the left subtree. The BSTree is initiated with a Node that contains the beauty of the first topic, then for all the remaining topics it computes 5 operations:

1. Find the index of the predecessor of the current topic with the **predecessor**
2. Find the maximum value of Node's value in the path predecessor-root, considering only Node with lower key (**get_max_value**). The value of a Node does not necessarily represent the maximum number of topics considering itself, but it considers the topics up to its beauty, so the value of a Node will be the maximum value contained in the left subtree. Initially, the value of the current topic could be the value of its predecessor (the largest of the smaller topic stored) + 1, but we may miss sequences with even lower beauty but inserted after the predecessor because they have a greater difficulty than the predecessor. **Get_max_value** allows to check the correct values starting from the predecessor.
3. Inserts, with **insert**, a new Node that represents the current topic, the key is equal to the beauty of the topic and the value equals the previously found maximum value + 1.
4. Updates the result variable with the maximum between the current result and maximum value + 1.
5. Updates the values in the tree, so that every Node in the BSTree has a consistent value that allows correct evaluations, with **update_max_values**.

```
pub fn design_a_course(_n: usize, mut topics: Vec<Vec<u32>>) -> u32 {
    topics.sort_by(|a, b| a[1].cmp(&b[1]).then_with(|| b[0].cmp(&a[0])));
    let mut tree = BSTree::with_root(topics[0][0]);
    let mut result = 1;
    for topic in &topics[1..] {
        let id_pred = tree.predecessor(topic[0]);
        let max_value = tree.get_max_value(id_pred, topic[0]);
        tree.insert(topic[0], max_value + 1);
        result = result.max(max_value + 1);
        tree.update_max_value();
    }
    result
```

```
}
```

The overall time complexity is O(nlogn) since for the n-1 topic we compute operation at most logarithmic (still assuming a balanced BST).

# Main

The main functions for both problems (respectively holiday_planning.rs and design_a_course.rs) read the input from the given file (it must be passed with clap, see the Usage section). The file must respect the structure specified in the hands-on website.

For problem #1, it reads the number of cities (n) and the number of days (D) from the first line, then for n times, it reads the following line as an array of dimension D and stores it. At last, it calls **holiday_planning**.

For problem #2 it's used the same approach. The main first reads the number of topics (n), then for n times, it reads the following line as a 2-dimensional array and stores it. At last, it calls **design_a_course**

## Usage

The shared zip file contains the shown code in the following structure:

- src: contains all the code, grouped as:
  - lib: a folder containing the lib_dp.rs file, that includes the code described.
  - bin: a folder containing the two main files for the two problems.
  - lib.rs: file that makes accessible the library in lib.
- test_1: a folder containing input and output given for problem #1.
- test_2: a folder containing input and output given for problem #2.

After applying cargo build, the command to run one of the two main files is:

cargo run --bin <main file> -- --file .\test_<1 || 2>\input<input_number>.txt

# Note: dependencies

Suggested dependencies: clap = { version = "4.5.26", features = ["derive"] }