Assignment 2

March 1, 2024

1 DAT600

1.0.1 Ardijan Rexhaj, Daniel Alvsåker, Ove Oftedal

1.1 Task 1.1:

Using the recursive algorithm from the book, defined as:

$$m[i,j] = \begin{cases} 0 & \text{if} \quad i = 0 \\ min\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}, & \text{if} \quad i > 0 \end{cases}$$

K denotes the positioning of the split and is therefore bounded too:

$$i \leq k < j$$

We apply the algorithm for these matrices:

Matrix	A1	A2	A3	A4	A5
Dimension	30×10	10×20	20×5	5×40	40×20

This gives us the input dimentions:

$$p_n = [30, 10, 20, 5, 40, 20]$$

$$\begin{aligned} &i = 1 \\ &j = 2 \\ &m[1,2] = min\left\{k = 1, \quad m[1,1] + m[2,2] + p_0p_1p_2 = 0 + 0 + 30 * 10 * 20 = 6000\right. \\ &m[1,2] = 6000, k = 1 \end{aligned}$$

$$\begin{aligned} &i = 1 \\ &j = 3 \\ &m[1,3] = min\begin{cases} k = 1, & m[1,1] + m[2,3] + p_0p_1p_3 = 0 + 1000 + 30 * 10 * 5 = 2500 \\ k = 2, & m[1,2] + m[3,3] + p_0p_2p_3 = 6000 + 0 + 30 * 20 * 5 = 9000 \end{cases} \\ &m[1,3] = 2500, k = 1 \end{aligned}$$

$i = 1$

$j = 4$

$$m[1,4] = min \begin{cases} k = 1, & m[1,1] + m[2,4] + p_0p_1p_4 = 0 + 3000 + 30 * 10 * 40 = 15000 \\ k = 2, & m[1,2] + m[3,4] + p_0p_2p_4 = 6000 + 4000 + 30 * 20 * 40 = 34000 \\ k = 3, & m[1,3] + m[4,4] + p_0p_3p_4 = 2500 + 0 + 30 * 5 * 40 = 8500 \end{cases}$$

$m[1,4] = 8500, k = 3$

$i = 1$

$j = 5$

$$m[1,5] = min \begin{cases} k = 1, & m[1,1] + m[2,5] + p_0p_1p_5 = 0 + 6000 + 30 * 10 * 20 = 12000 \\ k = 2, & m[1,2] + m[3,5] + p_0p_2p_5 = 6000 + 6000 + 30 * 20 * 20 = 24000 \\ k = 3, & m[1,3] + m[4,5] + p_0p_3p_5 = 2500 + 4000 + 30 * 5 * 20 = 9500 \\ k = 4, & m[1,4] + m[5,5] + p_0p_4p_5 = 8500 + 0 + 30 * 40 * 20 = 32500 \end{cases}$$

$m[1,5] = 9500, k = 3$

$i = 2$

$j = 3$

$$m[2,3] = min \begin{cases} k = 2, & m[2,2] + m[3,3] + p_1p_2p_3 = 0 + 0 + 10 * 20 * 5 = 1000 \end{cases}$$

$m[2,3] = 1000, k = 2$

$i = 2$

$j = 4$

$$m[2,4] = min \begin{cases} k = 2, & m[2,2] + m[3,4] + p_1p_2p_4 = 0 + 4000 + 10 * 20 * 40 = 12000 \\ k = 3, & m[2,3] + m[4,4] + p_1p_3p_4 = 1000 + 0 + 10 * 5 * 40 = 3000 \end{cases}$$

$m[2,4] = 3000, k = 3$

$i = 2$

$j = 5$

$$m[2,5] = min \begin{cases} k = 2, & m[2,2] + m[3,5] + p_1p_2p_5 = 0 + 6000 + 10 * 20 * 20 = 10000 \\ k = 3, & m[2,3] + m[4,5] + p_1p_3p_5 = 1000 + 4000 + 10 * 5 * 20 = 6000 \\ k = 4, & m[2,4] + m[5,5] + p_1p_4p_5 = 3000 + 0 + 10 * 40 * 20 = 11000 \end{cases}$$

$m[2,5] = 6000, k = 3$

$i = 3$

$j = 4$

$$m[3,4] = min \begin{cases} k = 3, & m[3,3] + m[4,4] + p_2p_3p_4 = 0 + 0 + 20 * 5 * 40 = 4000 \end{cases}$$

$m[3,4] = 4000, k = 3$

$i = 3$

$j = 5$

$$m[3,5] = min \begin{cases} k = 3, \quad m[3,3] + m[4,5] + p_2 p_3 p_5 = 0 + 4000 + 20 * 5 * 20 = 6000 \\ k = 4, \quad m[3,4] + m[5,5] + p_2 p_4 p_5 = 4000 + 0 + 20 * 40 * 20 = 20000 \\ \end{cases}$$

$m[3,5] = 6000, k = 3$

$i = 4$

$j = 5$

$$m[4,5] = min \left\{ k = 4, \quad m[4,4] + m[5,5] + p_3 p_4 p_5 = 0 + 0 + 5 * 40 * 20 = 4000 \right.$$

$m[4,5] = 4000, k = 4$

The A1..A5 matrices thus produce the following memoization matrices:

$m[i,j]$:

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$i=1$	0	6000	2500	8500	9500
$i=2$		0	1000	3000	6000
$i=3$			0	4000	6000
$i=4$				0	4000
$i=5$					0

$s[i,j]$:

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$i=1$		1	1	3	3
$i=2$			2	3	3
$i=3$				3	3
$i=4$					4

1.2 Task 1.2

[ ]:
```
use std::collections::HashMap;


#[derive(Debug, Clone, Copy)]
struct Cost {
    m: u32,
    s: usize,
}

fn calc_and_print_matrix(m: &Vec<usize>) {
    for r in calc_sm_table(&m){
        for v in r.into_iter().rev() {
            if let Some(i) = v{
                print!(", {} | {} ",i.m,i.s);
            }
        }
```

```rust
        print!("\n");
    }
}

fn m(i: usize, j: usize, dimensions: &Vec<usize>, ms_table: &mut Vec<Vec<Option<Cost>>>) -> u32 {
    if i == j {
        return 0;
    }

    if let Some(v) = ms_table[i][j] {
        return v.m;
    }

    let mut choices = (i..j)
        .map(|k| Cost {
            m: m(i, k, dimensions, ms_table)
                + m(k + 1, j, dimensions, ms_table)
                + (dimensions[i - 1] * dimensions[k] * dimensions[j]) as u32,
            s: k,
        })
        .collect::<Vec<Cost>>();

    choices.sort_by(|x, y| x.m.cmp(&y.m));

    ms_table[i][j] = Some(choices[0]);

    choices[0].m
}

fn calc_sm_table(dimensions: &Vec<usize>) -> Vec<Vec<Option<Cost>>> {
    let dim_size = dimensions.len();
    let mut ms_table: Vec<Vec<Option<Cost>>> = vec![vec![None; dim_size]; dim_size];

    for i in 1..dim_size {
        for j in i..dim_size {
            _ = m(i, j, dimensions, &mut ms_table)
        }
    }

    ms_table
}



calc_and_print_matrix(&vec![30, 10, 20, 5, 40, 20]);
```

```
calc_and_print_matrix(& vec![30, 35, 15, 5, 10, 20, 25]);
```

```
, 9500 | 3 , 8500 | 3 , 2500 | 1 , 6000 | 1
, 6000 | 3 , 3000 | 3 , 1000 | 2
, 6000 | 3 , 4000 | 3
, 4000 | 4


, 15125 | 3 , 11875 | 3 , 9375 | 3 , 7875 | 1 , 15750 | 1
, 10500 | 3 , 7125 | 3 , 4375 | 3 , 2625 | 2
, 5375 | 3 , 2500 | 3 , 750 | 3
, 3500 | 5 , 1000 | 4
, 5000 | 5
```

1.3 Task 1.3

No, there is no greedy algorithm that produces an optimal solution for

$$A_0..A_n$$

matrixes, the reason for this is because there is no local greedy choice which will produce an optimal solution.

1.4 Task 2

Binary Knapsack problem: Given set with a fixed capacity and an assortment of items

$$I_0..I_n$$

where each item has a capacity cost and a value associated with it, find the highest valued subset V of I where the entire cost of each element must fit within the capacity of the subset, in addition the sum of all item's cost cannot exceed the capacity.

1.4.1 2.1

[ ]:
```
#[derive(Debug, Copy, Clone)]
struct ItemInt {
    capacity_cost: usize,
    value: usize,
}

fn knapsack(capacity: usize, items: &[ItemInt]) -> (usize, Vec<ItemInt>) {
    let item_count = items.len();
    let mut cost_table: Vec<Vec<(usize, Vec<ItemInt>)>> =
        vec![vec![(0, vec![]); capacity + 1]; item_count + 1];

    for item in 1..=item_count {
        for cap in 0..=capacity {
```

```
            if items[item - 1].capacity_cost <= cap {
                let v_1 = cost_table[item - 1][cap].clone();
                let mut v_2 = cost_table[item - 1][cap - items[item - 1].
 ↪capacity_cost].clone();

                cost_table[item][cap] = if v_1.0 > v_2.0 + items[item - 1].
 ↪value {
                    v_1
                } else {
                    v_2.0 += items[item - 1].value;
                    v_2.1.push(items[item - 1]);
                    v_2
                }
            } else {
                cost_table[item][cap] = cost_table[item - 1][cap].clone();
            }
        }
    }

    // for i in cost_table.clone() {
    //     println!("{:?}", i);
    // }

    cost_table[item_count][capacity].clone()
}



let items = vec![
    ItemInt {
        capacity_cost: 15,
        value: 20,
    },
    ItemInt {
        capacity_cost: 100,
        value: 20,
    },
    ItemInt {
        capacity_cost: 5,
        value: 200,
    },
    ItemInt {
        capacity_cost: 30,
        value: 20,
    },
    ItemInt {
        capacity_cost: 10,
```

```
        value: 10,
    },
    ItemInt {
        capacity_cost: 20,
        value: 50,
    },
];

for i in 2..=4 {
    let capacity = i*20;
    let max_val = knapsack(capacity,&items);
    println!{"Capacity: {} gives a max value of: {:#?}",capacity, max_val}
}
```

```
Capacity: 40 gives a max value of: (
    270,
    [
        ItemInt {
            capacity_cost: 15,
            value: 20,
        },
        ItemInt {
            capacity_cost: 5,
            value: 200,
        },
        ItemInt {
            capacity_cost: 20,
            value: 50,
        },
    ],
)
Capacity: 60 gives a max value of: (
    280,
    [
        ItemInt {
            capacity_cost: 15,
            value: 20,
        },
        ItemInt {
            capacity_cost: 5,
            value: 200,
        },
        ItemInt {
            capacity_cost: 10,
            value: 10,
        },
        ItemInt {
            capacity_cost: 20,
```

```
            value: 50,
        },
    ],
)
Capacity: 80 gives a max value of: (
    300,
    [
        ItemInt {
            capacity_cost: 15,
            value: 20,
        },
        ItemInt {
            capacity_cost: 5,
            value: 200,
        },
        ItemInt {
            capacity_cost: 30,
            value: 20,
        },
        ItemInt {
            capacity_cost: 10,
            value: 10,
        },
        ItemInt {
            capacity_cost: 20,
            value: 50,
        },
    ],
)
```

[ ]: ()

1.4.2 2.2

Fractional knapsack problem: Any fractional amount of an item can be included in V.

[ ]:
```
#[derive(Debug, Copy, Clone)]
struct Item {
    capacity_cost: f32,
    value: f32,
}

impl Item {
    fn value_pr_cost(&self) -> f32 {
        self.value / self.capacity_cost
    }
}
```

```
type Amount = f32;

fn fractional_knapsack_greedy(mut capacity: f32, mut items: Vec<Item>) -> Vec<(Amount, Item)> {
    items.sort_by(|a, b| a.value_pr_cost().partial_cmp(&b.value_pr_cost()).unwrap());
    items.reverse();

    let mut results = vec![];
    for i in items {
        let remaining_capacity = capacity - i.capacity_cost;
        if remaining_capacity <= 0. {
            results.push((capacity, i));
            break;
        }

        results.push((i.capacity_cost, i));
        capacity = remaining_capacity;
    }

    results
}
```

```
let items = vec![
    Item {
        capacity_cost: 20.,
        value: 20.,
    },
    Item {
        capacity_cost: 100.,
        value: 20.,
    },
    Item {
        capacity_cost: 20.,
        value: 200.,
    },
    Item {
        capacity_cost: 30.,
        value: 20.,
    },
    Item {
        capacity_cost: 20.,
        value: 10.,
    },
```

```rust
    Item {
        capacity_cost: 20.,
        value: 50.,
    },
];

for i in 2..4 {
    let capacity = i as f32 *20.;
    let max_val = fractional_knapsack_greedy(capacity, items.clone());
    println!{"Capacity: {} gives a max value of: {:#?}",capacity, max_val}
}
```

```
Capacity: 40 gives a max value of: [
    (
        20.0,
        Item {
            capacity_cost: 20.0,
            value: 200.0,
        },
    ),
    (
        20.0,
        Item {
            capacity_cost: 20.0,
            value: 50.0,
        },
    ),
]
Capacity: 60 gives a max value of: [
    (
        20.0,
        Item {
            capacity_cost: 20.0,
            value: 200.0,
        },
    ),
    (
        20.0,
        Item {
            capacity_cost: 20.0,
            value: 50.0,
        },
    ),
    (
        20.0,
        Item {
            capacity_cost: 20.0,
            value: 20.0,
```

```
        },
    ),
]
```

[ ]: ()

1.5 Task 3.1

A greedy solution to find the fewest coins needed to achieve the amount N in a coin system where:

$$c_1 < c_2 < ... < c_n$$

And

$$c_n \mod c_{n-1} = 0$$

Would be to the use the largest coins

$$c_n$$

where

$$c_{n_1} <= N$$

, we use as many coins as possible of the largest ones before moving on to the next smaller, and repeat until the remainder

$$R = c_x \ R/c_x$$

becomes 0.

1.6 Task 3.3

[ ]:
```rust
fn minimum_coins(
    remainder: usize,
    coins: &Vec<usize>,
    mem_table: &mut HashMap<usize, (usize, Vec<usize>)>,
) -> (usize, Vec<usize>) {
    if let Some(result) = mem_table.get(&remainder) {
        return result.clone();
    }

    if remainder == 0 {
        return (0, vec![0; coins.len()]);
    }

    let mut min_count = (usize::MAX, vec![0; coins.len()]);

    for i in 0..coins.len() {
        let coin = coins[i];
```

```
        if coin <= remainder {
            let mut new_count = minimum_coins(remainder - coin, coins,␣
↪mem_table);
            new_count.0 += 1;

            min_count = if min_count.0 < new_count.0 {
                min_count
            } else {
                new_count.1[i] += 1;
                new_count
            }
        }
    }

    mem_table.insert(remainder, min_count.clone());
    min_count
}

let coins = vec![1, 5, 11];
let mut mem_table: HashMap<usize, (usize, Vec<usize>)> = HashMap::new();

let (val, choices) = minimum_coins(36, &coins, &mut mem_table);
println!("min coins: {:?}", val);
for (i, choice) in choices.iter().enumerate() {
    println!("Coin {}, amount: {}", coins[i], choice);
}
let (val, choices) = minimum_coins(111, &coins, &mut mem_table);
println!("min coins: {:?}", val);
for (i, choice) in choices.iter().enumerate() {
    println!("Coin {}, amount: {}", coins[i], choice);
}
```

```
min coins: 6
Coin 1, amount: 3
Coin 5, amount: 0
Coin 11, amount: 3
min coins: 11
Coin 1, amount: 1
Coin 5, amount: 0
Coin 11, amount: 10
```

[ ]: ()

1.7 3.4

Yes the norwegian coin system is greedy because each smaller coin wholely divides all larger coins. Also can be proven by:

```
19 = 10+5+1+1+1+1
```

```
18 = 10+5+1+1+1
17 = 10+5+1+1
16 = 10+5+1
15 = 10+5
14 = 10+1+1+1+1
13 = 10+1+1+1
12 = 10+1+1
11 = 10+1
10 = 10
9  = 5+1+1+1+1
8  = 5+1+1+1
7  = 5+1+1
6  = 5+1
5  = 5
4  = 1+1+1+1
3  = 1+1+1
2  = 1+1
1  = 1
```

1.8 3.5

If N is the value we are looking to achieve, and M is the number of coins.

For the greedy proposition, the worst case running time would be the number of coins in the currency system

$$O(M)$$

.

For the dynamic solution which works for any coin system that includes the smallest coin of 1 the running time would be

$$O(N \cdot M)$$

.