Deggendorf Institute of Technology
Algorithms and Data Structures Coursework

Professor Patrick GLAUNER

2024 Summer Term

1 Introduction

In parallel to the lectures, this coursework is offered. The aim of this coursework is to deepen and broaden topics taught in class. You will be able to elaborate your understanding by turning theoretical knowledge into code. You will also have the opportunity to become more familiar with the theory by solving assignments. Most of your solutions are not examined directly, but you are strongly encouraged to work on as many problems as time permits. **The written exam may refer to the problems of this coursework!** Questions and solutions can be discussed during the laboratory sessions.

2 Mandatory problem

Note: This problem is completely independent of the content taught in class. You can start working on this problem whenever you want.

Complete exercise P-13.50 (comparison of three pattern matching algorithms) in Goodrich et al. in order to pass the mandatory problem ("Leistungsnachweis"/"Übungsleistung"). You need to implement your code in a Jupyter notebook[1] and include a report in it that consists of the following parts:

1. Introduction and problem description.
2. Description of the three algorithms and how they work.
3. Description of the text documents that you use for testing your algorithm implementations.
4. Presentation and discussion of experimental results.
5. Conclusions and prospects: summary and what else you could do in order to further improve your project.

Groups of up to 4 students can make a joint submission. **Submit a HTML export of your notebook via iLearn until June 29, 2024.** Your submission must include the names and student IDs of all group members.

3 Exam bonus problem

Successful completion of Problem 12.6 carries a 10% bonus for the exam. You need to implement your code in a Jupyter notebook and include a training set of your choice and visualize the learned tree. Groups of up to 4 students can make a joint submission. **Submit a HTML export of your notebook via iLearn until June 29, 2024.** Your submission must include the names and student IDs of all group members.

[1] `http://jupyter.org/`

4 Programming language

You can choose any programming language to solve the problems. However, Python is the recommended language.

5 Use of AI tools

You are encouraged to use (generative) AI tools, such as ChatGPT for text, when solving the problems. Like this, you will be more productive and possibly achieve far better results. AI tools will provide you very quickly with a structure or draft that you can then amend, improve and extend. The following rules apply:

- You are expected to substantially amend, improve and extend the output of AI tools.
- Critically assess the output of AI tools. They may sound convincing but could be completely wrong.
- Add references in order to support the claims generated by AI tools.
- Clearly state which AI tools you have used for which parts of your coursework.
- Check your coursework for plagiarism. Some AI tools may simply copy entire sentences or paragraphs from other sources.

Read the privacy policy of AI tools before you use them. Do not send confidential data or trade secrets to cloud-based AI tools.

6 Python primer

Problem 6.1

Familiarize yourself with Python using `intro-to-python.py`. Also familiarize yourself with a Python IDE such as PyCharm. Furthermore, try interactive programming using `python -i` or `ipython`.

Problem 6.2

Do exercises R-1.1 to R-1.11, C-1.13, C-1.24, C-1.25, C-1.28 and P-1.29 in Goodrich et al.

Problem 6.3

Consider a circle inscribed in a unit square. Given that the circle and the square have a ratio of areas that is $\frac{\pi}{4}$, the value of π can be approximated using a Monte Carlo method:

1. Draw a square, then inscribe a circle within it.
2. Uniformly scatter objects of uniform size over the square.
3. Count the number of objects inside the circle and the total number of objects.
4. The ratio of the two counts is an estimate of the ratio of the two areas, which is $\frac{\pi}{4}$. Multiply the result by 4 to estimate π.

Implement this Monte Carlo method and run it for a varying number (e.g. 100, $1,000$, $10,000$, $100,000$, ...) of objects. Compare your results to the actual value of π (`math.pi`).

7 Graphs

Problem 7.1

In the standard pouring problem, there are two glasses of different capacity. Legal actions are filling a glass, emptying a glass or pouring as much as possible of a glass into the other one. Glutting is not legal. Given an initial filling, a goal filling must be achieved by applying these actions. The shortest path of actions must be returned or no path in case the problem is not solvable for a certain configuration of start, goal and capacities. Sample configuration:

- Capacities: 418 and 986
- Start: 0 and 0
- Goal: 6 and 0

Implement the pouring problem using the shortest path function defined in class. This allows to focus on the task without having to reimplement the shortest path algorithm. Complete Algorithm 1 and use it for different capacities, start conditions and goals. Can you create a setting that is not solvable?

Algorithm 1 Skeleton of pouring problem

```python
from shortest_path import shortest_path_search

def successors(X, Y):
    def sc(state):
        x, y = state
        assert x <= X and y <= Y
        return {(0, y+x) if y+x <= Y else (x-(Y-y), Y): 'x->y',
                (x+y, 0) if x+y <= X else (X, y-(X-x)): 'x<-y',
                [...]: 'fill x',
                [...]: 'fill y',
                [...]: 'empty x',
                [...]: 'empty y'}
    return sc

if __name__ == '__main__':
    res = shortest_path_search((0, 0), successors(418, 986), lambda state: state ==
         (6, 0))
    print(res)
    print('%s transitions' % (int(len(res) / 2)))
```

Problem 7.2

Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find the minimum number of actions to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. Re-use the shortest path algorithm presented in class for solving this problem. Also, `itertools.combinations_with_replacement` may be helpful when implementing your `successors` function.

8 Complexity analysis

Problem 8.1

Do exercises R-3.2, R-3.3, R-3.8, R-3.17, R-3.18, R3.21, R-3.23 to R-3.27 and C-3.36 in Goodrich et al.

Problem 8.2

Show that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ and $\frac{n(n+1)}{2}$ is $O(n^2)$.

Problem 8.3

Do exercise R-3.1 in Goodrich et al. You are advised to use the `matplotlib`[2] module.

9 Lists

Problem 9.1

Do exercises R-5.1 to R-5.3 in Goodrich et al.

Problem 9.2

Review the list implementation in Algorithm 2. Extend it by implementing `__setitem__(self, k, value)`, `insert(self, k, value)` and `remove(self, k)` methods.

Algorithm 2 Array list

```
class MyArray:
    def __init__(self):
        self._n = 0
        self._capacity = 1
        self._A = self._make_array(self._capacity)
    def __len__(self):
        return self._n
    def __getitem__(self, k):
        if not 0 <= k < self._n:
            raise IndexError('invalid index')
        return self._A[k]
    def append(self, obj):
        if self._n == self._capacity:
            self._resize(2 * self._capacity)
        self._A[self._n] = obj
        self._n += 1
    def _resize(self, c):
        B = self._make_array(c)
        for k in range(self._n):
            B[k] = self._A[k]
        self._A = B
        self._capacity = c
    def _make_array(self, c):
        return (c * ctypes.py_object)()
```

Problem 9.3

Extend the `__getitem__(self, k)` method such that it can handle negative indices.

Problem 9.4

Do exercises C-5.16 and C-5.25 in Goodrich et al.

Problem 9.5

Do exercises R-6.1, R-6.10, R-6.12 and C-6.17 in Goodrich et al.

Problem 9.6

Do exercise P-6.34 (evaluation of postfix notation) in Goodrich et al. Hint: use a stack.

[2] `http://matplotlib.org`

Problem 9.7
Implement a queue using a singly linked list.

Problem 9.8
Do exercises R-7.5 and R-7.9 in Goodrich et al.

10 Recursion

Problem 10.1
Do exercises R-4.1, R-4.2, C-4.9 and C-4.17 in Goodrich et al.

Problem 10.2
A trivial recursive definition of the power function follows from the fact that $x^n = x \times x^{n-1}$ for $n > 0$:

$$\text{power}(x,n) = \begin{cases} 1 & n = 0 \\ x \times \text{power}(x, n-1) & \text{otherwise} \end{cases} .$$

Implement this function and determine its running time complexity.

Problem 10.3
There is a much faster way to compute the power function using an alternative definition that employs a squaring technique:

$$\text{power}(x,n) = \begin{cases} 1 & n = 0 \\ \text{power}(x, \frac{n}{2})^2 & \text{if } n > 0 \text{ is even} \\ x \times \text{power}(x, \lfloor \frac{n}{2} \rfloor)^2 & \text{if } n > 0 \text{ is odd} \end{cases} .$$

Let $\lfloor \frac{n}{2} \rfloor$ denote the floor of the division (expressed as `n // 2` in Python).

Implement this function and determine its running time complexity.

11 Sorting

Problem 11.1
The bubble sort implementation we discussed in class has a best-case running time complexity of $O(n^2)$. Implement a better bubble sort that has a best-case running time complexity of $O(n)$. Hint: stop as early as possible, i.e. if the sequence was not altered at all in the previous iteration.

Problem 11.2
Implement `merge(S1, S2, S)` for merge sort.

Problem 11.3
Implement a randomized quicksort, which randomly picks the pivot element. This algorithm has a worst-case *expected* running time complexity of $O(n \log n)$. Compare its execution time to the quicksort algorithm shown in class. Use various data sets of $10,000$ or more elements, including a data set that is already sorted. What do you observe?

Note on randomized quicksort: Its absolute worst-case is still $O(n^2)$, yet it is very unlikely: "The probability that quicksort will use a quadratic number of compares when sorting a large array on your computer is much less than the probability that your computer will be struck by lightning."[3].

[3]Source: `http://algs4.cs.princeton.edu/23quicksort/`

12 Trees

Problem 12.1

Review the tree implementation in Algorithm 3. Extend it by implementing traversal methods for pre-order and post-order, both as generators and without generators, respectively.

Algorithm 3 Tree

```
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data
    def insert(self, data):
        if self.data:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                else:
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
        else:
            self.data = data
    def print(self):
        if self.left:
            self.left.print()
        print(self.data)
        if self.right:
            self.right.print()
    def inorder(self):
        if self.left:
            yield from self.left.inorder()
        yield self.data
        if self.right:
            yield from self.right.inorder()
```

Problem 12.2

Implement a function `invert(root)` that inverts a tree. Hint: using recursion, your code will be less than 10 lines.

Problem 12.3

You can represent a tree as a list of list, e.g. `[[1], [2, 3], [4, [5, 6, [7, 8, [9, [10, 11, 12]]]]]]`. Implement a function `flatten(tree)` that flattens a tree. The sample tree would then be turned into the following list: `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]`. Hint: using a `yield` expression and a `yield from` expression, your code will be less than 10 lines.

Problem 12.4

Create a simple decision tree composed of binary attributes for a game of your choice and write a program to evaluate it based on the values of the attributes concerned.

Problem 12.5

Perform the decision tree learning algorithm manually on the following data set or a data set of your choice, write out intermediate steps and the result.

	sky	air	humid	wind	water	forecast	attack
1	sunny	warm	normal	strong	warm	same	+
2	sunny	warm	high	strong	warm	same	+
3	rainy	cold	high	strong	warm	change	-
4	sunny	warm	high	strong	cool	change	+
5	sunny	warm	normal	weak	warm	same	-

Problem 12.6

Implement Algorithm 4 for binary attributes and run it on a training set of your choice.

Algorithm 4 Decision tree learning

```
function DT-LEARNING($examples$, $attributes$, $parent\_examples$)
    if empty($examples$) then return PLURALITY-VAL($parent\_examples$)
    else if all $examples$ have same classification then return the classification
    else if empty($attributes$) then return PLURALITY-VAL($examples$)
    else
        $A \leftarrow \underset{a \in attributes}{\text{argmax}}$ IMPORTANCE($a$, $examples$)
        $tree \leftarrow$ a new decision tree with root test $A$
        for $v_k \in A$ do
            $exs \leftarrow \{e | e \in examples \wedge e.A = v_k\}$
            $subtree \leftarrow$ DT-LEARNING($exs$, $attributes - A$, $examples$)
            add a branch to $tree$ with label ($A = v_k$) and subtree $subtree$
        end for
        return $tree$
    end if
end function
```

13 Maps and hash tables

Problem 13.1

Do exercises R-10.1 and R-10.3 in Goodrich et al.

Problem 13.2

Do exercises R-10.9 and R-10.10 in Goodrich et al.

Problem 13.3

Test the `ChainHashMap` class for hash tables of different types of key-value pairs and different sizes. See the `Resources.zip` file for an implementation of `ChainHashMap` and its superclasses.

Problem 13.4

Review the generic hash table implementation in Algorithm 5. Explain why attribute `_n` is updated in `__delitem__(self, k)` but not updated in `__setitem__(self, k, v)`.

Algorithm 5 Hash table

```
class HashMapBase(MapBase):
    def __init__(self, cap=11, p=109345121):
        self._table = cap * [None]
        self._n = 0
        self._prime = p
        self._scale = 1 + randrange(p-1)
        self._shift = randrange(p)
    def _hash_function(self, k):
        return (hash(k)*self._scale + self._shift) \
                    % self._prime % len(self._table)
    def __len__(self):
        return self._n
    def __getitem__(self, k):
        j = self._hash_function(k)
        return self._bucket_getitem(j, k)
    def __setitem__(self, k, v):
        j = self._hash_function(k)
        self._bucket_setitem(j, k, v)
        if self._n > len(self._table) // 2:
            self._resize(2 * len(self._table) - 1)
    def __delitem__(self, k):
        j = self._hash_function(k)
        self._bucket_delitem(j, k)
        self._n -= 1
    def _resize(self, c):
        old = list(self.items())
        self._table = c * [None]
        self._n = 0
        for (k, v) in old:
            self[k] = v
```

14 Selected topics

Problem 14.1

The recurrence relation for naive matrix multiplication is as follows:

$$\begin{aligned} T(n) &= 8T(n/2) + \Theta(n^2) \\ T(1) &= 1. \end{aligned}$$

Derive the worst-case running time complexity of naive matrix multiplication.

Problem 14.2

The recurrence relation for the Strassen algorithm is as follows:

$$\begin{aligned} T(n) &= 7T(n/2) + \Theta(n^2) \\ T(1) &= 1. \end{aligned}$$

Derive the worst-case running time complexity of the Strassen algorithm.

Problem 14.3

Given a sequence of matrices, find the most efficient way to multiply these matrices together[4]. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications. We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices **A**, **B**, **C** and **D**, we would have:

$$(\mathbf{ABC})\mathbf{D} = (\mathbf{AB})(\mathbf{CD}) = \mathbf{A}(\mathbf{BCD}) = \ldots .$$

However, the order in which we parenthesize the product affects the number of simple multiplications needed to compute the product, or the efficiency. For example, suppose **A** is a 10×30 matrix, **B** is a 30×5 matrix and **C** is a 5×60 matrix. Then

$$(\mathbf{AB})\mathbf{C} = (10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500 \text{ multiplications.}$$

How many multiplications does it take to compute $\mathbf{A}(\mathbf{BC})$? Which of the two parenthesizations requires less number of multiplications?

Problem 14.4

Given a list `p` which represents the chain of matrices such that the i-th matrix $\mathbf{A}_i$ is of dimension $\texttt{p}[i-1] \times \texttt{p}[i]$. Implement a function `matrix_chain_order(p)` that returns the minimum number of simple multiplications needed to multiply the chain.
Examples:

1. Input: `p = (10, 20, 30)`
   Output: 6000
   There are only 2 matrices of dimensions 10×20 and 20×30. So there is only one way to multiply the matrices, i.e. $10 \times 20 \times 30$ multiplications.

2. Input: `p = (40, 20, 30, 10, 30)`
   Output: 26000
   There are 4 matrices of dimensions 40×20, 20×30, 30×10 and 10×30. Let the input 4 matrices be **A**, **B**, **C** and **D**. The minimum number of multiplications are obtained by putting parentheses in following way $(\mathbf{A}(\mathbf{BC}))\mathbf{D} \implies 20 \times 30 \times 10 + 40 \times 20 \times 10 + 40 \times 10 \times 30$.

3. Input: `p = (10, 20, 30, 40, 30)`
   Output: 30000
   There are 4 matrices of dimensions 10×20, 20×30, 30×40 and 40×30. Let the input 4 matrices be **A**, **B**, **C** and **D**. The minimum number of multiplications are obtained by putting parentheses in following way $((\mathbf{AB})\mathbf{C})\mathbf{D} \implies 10 \times 20 \times 30 + 10 \times 30 \times 40 + 10 \times 40 \times 30$.

[4]Source: `http://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/`

A simple solution is to place parentheses at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parentheses in $n-1$ ways. For example, if the given chain is of 4 matrices, let the chain be $\mathbf{ABCD}$, then there are 3 ways to place first set of parentheses outer side: $(\mathbf{A})(\mathbf{BCD})$, $(\mathbf{AB})(\mathbf{CD})$ and $(\mathbf{ABC})(\mathbf{D})$. So when we place a set of parentheses, we divide the problem into subproblems of smaller size. The problem can be easily solved using recursion.

Problem 14.5

What is the worst-case running time complexity of the naive recursive approach?

Problem 14.6

Draw the recursion tree for a matrix chain of size 4.

Problem 14.7

We can see that there are many subproblems being called more than once. Improve your implementation using dynamic programming so that it has a polynomial worst-case running time complexity.

Problem 14.8

What is the worst-case running time complexity of your improved implementation?

Problem 14.9

Apply different values for r and x_0 to the logistic map. Plot each instance using the `matplotlib` module.

Problem 14.10

Amend the random number generator discussed in class such that it works without a seed. For example, initialize it with a value that depends on your system clock. You can use the `time` module for reading your system clock.

Problem 14.11

Implement Newton's method for computing $\sqrt{x}$ and $\frac{1}{\sqrt{x}}$, respectively.

Problem 14.12

Implement the bit manipulation approach of Quake III Arena for computing $\frac{1}{\sqrt{x}}$.

Problem 14.13

Run the different prime number generation algorithms discussed in class for different sizes of n and compare their running time.