Lab 2 — Programming for Data Science

DS2002 — Programming for Data Science

Summary

This lab is designed to help you practice writing clear and correct implementations of several fundamental algorithms in programming and data science. You will work with sorting algorithms, recursion, and binary search.

The goal of this lab is not just to produce working code, but to understand how and why these algorithms work. Avoid copying existing code directly from AI tools or online sources. Try to write your own implementations and only compare them to other solutions *after* you finish your attempt. Use this lab to build strong programming skills and confidence.

Throughout the lab, write your code in a way that is easy to test and read.

General Instructions

- Use separate classes for every task.
- Test your functions with different inputs.

Sorting Algorithms

To generate test data, you may use Python's `random` library to create random lists of integers. To compare results or verify correctness, you may also use Python's built-in functions such as `sorted()` or `list.sort()`.

Task 1: Bubble Sort

Goal: Implement the bubble sort algorithm from scratch.

What to include:

- Implement a mechanism that counts pairwise comparisons in the Bubble sort.
- At least one test on a random list.
- Printed output before and after sorting alongside the number of pairwise comparisons.

Task 2: Merge Sort

Goal: Implement the merge sort algorithm using recursion.

Merge sort divides the list into two halves, sorts each half separately (also using merge sort), and then merges the two sorted halves into one sorted list. This is an example of the **divide and conquer** strategy.

What to include:

- Implement a mechanism that counts the recursions in the Merge sort.
- At least one test on a random list.
- Printed output before and after sorting alongside the number of recursions.

Task 3: Performance Comparison

Goal: Compare the running time (you may use `time.perf_counter`) and number of pairwise comparisons and recursions of the sorting algorithms.

Your comparison should include:

- Your implementations of bubble sort and merge sort.
- Python's built-in `sort()` or `sorted()`.

Things to consider:

1. Results may be inconsistent because many external factors affect time measurements. Run each test several times with randomized lists and compute the average.
2. Try different input sizes (small, medium, large).
3. You may present results in a simple table or plot.

Extra Credit (Optional)

If you want to go further, try:

- Implement the Quicksort sorting algorithm and include the number of recursion calls in it. You should be able to explain it.
- Extend Task 3 to compare performance on different types of lists:
  - already sorted
  - almost sorted
  - random
  - reverse sorted

Recursion Practice

Task 4: Recursive Binary Search

Goal: Implement a recursive version of binary search.

Implement a mechanism that counts the number of recursions in the search.

Test the code on multiple lists.

What would happen if there are multiple cells with the target of the search?

Tree Search Practice

A Python script has been provided to you that creates a tree and returns its root. Review the script carefully, as you will be working with the `Node` class. You can obtain the root using the following code:

```
# Make sure the script is in the same directory as your own code.
# This line imports the function from the script
from tree import create_the_tree

# You have received the root of the tree.
root = create_the_tree()
```

Figure 1 shows the structure of the tree. Node `0` is the root.

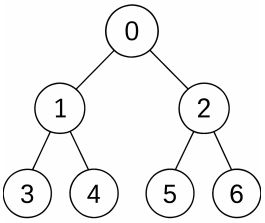

Figure 1: The Tree

Task 5: Tree Search

Write a class that implements Breadth-First Search (BFS) and Depth-First Search (DFS). Run both on the tree and print the order of visited nodes for each.

Question: When should we use BFS instead of DFS, and when is DFS more appropriate?