Algorithms & Data Structures SoSe 25 Notes

Igor Dimitrov

2024-12-18

Table of contents

Preface

1 Reading List

1.1 ADS

Basic & Light Reading

- Unlocking Algorithms. Cormen
- First Course in Algorithms Through Puzzles. Uehara

Intermediate

- Understanding Algorithms and Data Structures. Brunksill
- Algorithms + Data Structures = Programs. Wirth
- Problems on Algorithms. Ian Parberry
- Fundamentals of Algorithmics. Brassard, Bratley
- Foundations of Algorithms. Neapolitan
- Data Structures and Algorithms - a First Course. Adamson
- Algorithms and Data Structures - Design, Correctness, and Analysis. Kingston
- Data Structures and Their Algorithms. Lewis, Denenberg
- Design and Analysis of Algorithms. Smith

Python

- Data Structures and Algorithms in Python. Lafore
- Competitive Programming in Python. Duerr

C

- Algorithms and Data Structures - an Approach in C.
- Programs and Data Structures in C. Ammeraal
- Foundations of Computer Science. Ullman

C++

- Data Structures and Algorithm Analysis in C++. Weiss
- Data Structures and Problem Solving using C++. Weiss
- Principles of Algorithmic Problem Solving. Sannemo

Java

- Fundamentals of OOP and Data Structures in Java. Wiener

Advanced

- How to Think About Algorithms. Jeff Edmonds
- Basic Toolbox. Melhorn
- Algorithms. Erickson

1.2 Discrete Math and Graph Theory

1. Mathematical Structures for Computer Science. Judith Gersting
    1. 3: Recurrence Relations & Analysis of Algorithms
    2. 5: Graphs & Trees
    3. 7: Graph Algorithms
2. Discrete Mathematics. Rosen 3: Algorithms 5: Induction & Recursion 8: Advanced Counting: recurrence relations 10: Graphs 11: Trees
3. Discrete & Combinatorial Mathematics
    1. 4: Mathematical Induction
    2. 5.7, 5.8: Analysis of Algorithms
    3. 10: Recurrence Relations
    4. 11, 12, 13: Graph Theory
4. Diskrete Mathematik fuer Einsteiger. Beutelspacher
5. Discrete Mathematics in Computer Science. Golovnev, Kulikov
6. Concrete Mathematics. Knuth

Graph Specific

1. Graph Theory - A Problem Oriented Approach. Daniel Marcus
2. Algorithmic Graph Theory. Alan Gibbons

Part I

Python

2 Iterables

2.1 `for` Loops and Comprehensions

`for` loops in Python are used to iterate over any iterable (like lists, tuples, strings, sets, dictionaries, generators, etc.).

Syntax:

```
for item in iterable:
    # do something with item
```

Common Use Cases & Idioms:

Basic Iteration

```
names = ["Alice", "Bob", "Charlie"]
for name in names:
    print(name)
```

Iterating with Index (use `enumerate`)

```
for i, name in enumerate(names):
    print(f"{i}: {name}")
```

Iterating Multiple Sequences (use `zip`)

```
ages = [25, 30, 22]
for name, age in zip(names, ages):
    print(f"{name} is {age} years old")
```

Iterating Over Dictionaries

```
person = {"name": "Alice", "age": 25}
for key, value in person.items():
    print(key, value)
```

Nested Loops

```
for i in range(3):
    for j in range(2):
        print(i, j)
```

What Are Comprehensions?

Comprehensions are **concise expressions** for generating new iterables (like lists, sets, or dicts) using the syntax of a `for` loop inside a single line.

Types and Idiomatic Patterns

List Comprehension (most common)

```
squares = [x**2 for x in range(5)]
# Output: [0, 1, 4, 9, 16]
```

Conditional List Comprehension

```
evens = [x for x in range(10) if x % 2 == 0]
# Output: [0, 2, 4, 6, 8]
```

Set Comprehension

```
unique_lengths = {len(word) for word in ["a", "ab", "abc", "ab"]}
# Output: {1, 2, 3}
```

Dict Comprehension

```
words = ["apple", "banana", "cherry"]
lengths = {word: len(word) for word in words}
# Output: {'apple': 5, 'banana': 6, 'cherry': 6}
```

Nested Comprehensions (2D lists)

```
matrix = [[i * j for j in range(3)] for i in range(3)]
# Output: [[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```

2.2 `enumerate()`

`enumerate()` is a built-in Python function that adds a **counter** to any iterable (like a list, tuple, or string), returning an **enumerate object**, which yields `(index, value)` pairs on iteration.

Type:

```
type(enumerate(['a', 'b', 'c']))  # <class 'enumerate'>
```

Like `zip`, it's a **lazy iterable**, meaning it produces values on demand and can be turned into a list or looped over.

Basic Example:

```
fruits = ['apple', 'banana', 'cherry']

for i, fruit in enumerate(fruits):
    print(i, fruit)
```

Output:

```
0 apple
1 banana
2 cherry
```

Common & Idiomatic Use Cases for `enumerate()`

1. **Avoid Manual Indexing with `range(len(...))`**

Instead of:

```
for i in range(len(fruits)):
    print(i, fruits[i])
```

Do this:

```
for i, fruit in enumerate(fruits):
    print(i, fruit)
```

Cleaner, more Pythonic.

2. **Start Index at a Custom Value**

```
for i, fruit in enumerate(fruits, start=1):
    print(f"{i}. {fruit}")
```

Output:

```
1. apple
2. banana
3. cherry
```

Great for user-friendly numbering (e.g. starting from 1 instead of 0).

3. **Tracking Position in File or Data**

```
with open("file.txt") as f:
    for lineno, line in enumerate(f, start=1):
        print(f"Line {lineno}: {line.strip()}")
```

Common in data processing and log parsing.

4. **Enumerate with Conditional Logic**

```python
colors = ['red', 'blue', 'green', 'blue']
for i, color in enumerate(colors):
    if color == 'blue':
        print(f"'blue' found at index {i}")
```

Helps track positions that meet a condition.

5. **Use with `zip()` for Triple Iteration**

```python
a = ['x', 'y', 'z']
b = [10, 20, 30]
for i, (x, y) in enumerate(zip(a, b)):
    print(f"{i}: {x}-{y}")
```

Combines enumeration with parallel iteration.

Summary:

Function	What it does	Output form
`zip(a, b)`	Combines sequences	`(a[i], b[i])`
`enumerate(x)`	Adds index to an iterable	`(i, x[i])`
`enumerate(x, start=n)`	Like above, but starts at `n`	`(n, x[0])`, `(n+1, x[1])`, …

2.3 `zip()`

The built-in `zip()` function takes **two or more iterables** (like lists, tuples, or strings) and **aggregates elements from each iterable by position** (i.e. index). It returns an **iterator of tuples**, where the *i-th* tuple contains the *i-th* element from each of the input iterables.

```python
zip(iterable1, iterable2, ...)
```

It stops when the shortest input iterable is exhausted.

You can think of `zip()` as:

```python
zip(A, B, C)   [(A[0], B[0], C[0]), (A[1], B[1], C[1]), ...]
```

No matter the input shape, `zip()` **always does the same thing**: *Group elements by position across multiple iterables.*

`zip()` returns a **zip object**, which is an **iterator**. You need to explicitly convert it into a list or tuple to see the full output:

```
list(zip(...))     # common
tuple(zip(...))    # possible
```

Common Use Cases and Idiomatic Patterns

1. **Combining Lists (Zipping)**

```
letters = ['a', 'b', 'c']
numbers = [1, 2, 3]

zipped = list(zip(letters, numbers))
print(zipped)
# Output: [('a', 1), ('b', 2), ('c', 3)]
```

Useful for:

- Pairing related data.
- Iterating in parallel over multiple lists.

2. **Looping Over Zipped Values**

```
names = ['Alice', 'Bob']
scores = [85, 92]

for name, score in zip(names, scores):
    print(f"{name} scored {score}")
# Output:
# Alice scored 85
# Bob scored 92
```

This is an idiomatic way to loop over multiple sequences in sync.

3. **Unzipping (Inverse of zip) with * Unpacking**

```python
pairs = [('a', 1), ('b', 2), ('c', 3)]

letters, numbers = zip(*pairs)

print(letters)  # Output: ('a', 'b', 'c')
print(numbers)  # Output: (1, 2, 3)
```

Explanation:

- `*pairs` unpacks the list into separate arguments: `zip(('a', 1), ('b', 2), ...)`
- `zip()` groups by position: first elements, second elements, etc.

This is effectively **transposing a 2D structure**.

4. **Creating Dictionaries**

```python
keys = ['name', 'age']
values = ['Alice', 30]

dictionary = dict(zip(keys, values))
print(dictionary)
# Output: {'name': 'Alice', 'age': 30}
```

A common idiom when you have two separate sequences representing keys and values.

5. **Zipping with Unequal Lengths**

```python
a = [1, 2, 3]
b = ['x', 'y']

print(list(zip(a, b)))
# Output: [(1, 'x'), (2, 'y')]
```

Only pairs up to the shortest iterable. (See `itertools.zip_longest()` if you want padding.)

Unified Understanding: Zip vs. "Unzip"

Why `zip()` seems to do **two very different things**:

1. **Zipping:** Combine separate lists into paired tuples.
2. **Unzipping:** Split paired tuples into separate lists.

clarification:

```
# Zipping
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
zipped = list(zip(list1, list2))
# Output: [('a', 1), ('b', 2), ('c', 3)]

# Unzipping
pairs = [('a', 1), ('b', 2), ('c', 3)]
unzipped = list(zip(*pairs))
# Output: [('a', 'b', 'c'), (1, 2, 3)]
```

Even though the **intent** differs, the **operation** is identical:

> Group elements by position across the given iterables.

- In *zipping*, the elements come from separate sequences.
- In *unzipping*, the unpacking `*` turns a list of tuples into separate positional iterables, and zip groups those.

So:

- `zip(A, B)` zips rows.
- `zip(*rows)` transposes the matrix — an "unzip" operation in spirit, but still just zip applied to unpacked input.

Bonus: Visual Matrix Analogy

Consider this "table" of rows (a list of tuples):

```
rows = [('a', 1),
        ('b', 2),
        ('c', 3)]
```

If you do:

```
zip(*rows)
```

You're transposing it into:

```
[('a', 'b', 'c'), (1, 2, 3)]
```

This is **column-wise grouping**.

Summary

- `zip()` is a fundamental tool for working with **multiple iterables in parallel**.
- Always groups **by index**.
- Use it to zip, loop, unzip, transpose, and build dictionaries.
- When used with `*`, you can reverse its effect by unpacking rows into inputs.

It's simple, powerful, and highly idiomatic in Python.

2.4 `map()` and `filter()`

What is `map()`?

`map(func, iterable)` applies the function `func` to each item in the iterable, returning a **map object** (an iterator).

Basic Use:

```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, nums))
# Output: [1, 4, 9, 16]
```

What is `filter()`?

`filter(func, iterable)` selects items from the iterable **for which `func(item)` is true**, returning a **filter object** (an iterator).

```python
nums = [1, 2, 3, 4]
evens = list(filter(lambda x: x % 2 == 0, nums))
# Output: [2, 4]
```

Idiomatic Use Cases:

Apply Transformation to All Elements

```python
uppercased = list(map(str.upper, ["a", "b", "c"]))
# Output: ['A', 'B', 'C']
```

Filter with Condition

```python
short_words = list(filter(lambda w: len(w) < 4, ["a", "apple", "bat", "cat"]))
# Output: ['a', 'bat', 'cat']
```

Combine with `zip`

```python
a = [1, 2, 3]
b = [4, 5, 6]
summed = list(map(lambda x: x[0] + x[1], zip(a, b)))
# Output: [5, 7, 9]
```

Equivalent List Comprehensions (more Pythonic)

```python
# Instead of map
[x**2 for x in nums]

# Instead of filter
[x for x in nums if x % 2 == 0]
```

Note: While `map` and `filter` are perfectly valid, **list comprehensions** are often preferred in Python due to better readability.

Final Recap Table

Concept	Description	Common Use Cases
`for` loop	Iterates over any iterable	Basic iteration, nested loops
Comprehension	Concise iterable construction	List/set/dict creation, filtering
`map(func, it)`	Apply `func` to all items	Transform elements
`filter(func, it)`	Keep items where `func(item)` is True	Selective filtering

2.5 Extended Unpacking in Python with * and **

Python allows powerful unpacking syntax to **distribute or collect values** in assignments and function calls.

Sequence Unpacking (with *)

Standard unpacking:

```
a, b, c = [1, 2, 3]
print(a, b, c)
```

Output:

```
1 2 3
```

Extended unpacking:

```
a, *b = [1, 2, 3, 4]
print(a, b)
```

Output:

```
1 [2, 3, 4]
```

```
*a, b = [1, 2, 3, 4]
print(a, b)
```

Output:

```
[1, 2, 3] 4
```

```python
a, *b, c = [1, 2, 3, 4, 5]
print(a, b, c)
```

Output:

```
1 [2, 3, 4] 5
```

Unpacking in Function Calls (with * and **)

Positional unpacking with *:

```python
def add(a, b, c):
    return a + b + c

nums = [1, 2, 3]
print(add(*nums))
```

Output:

```
6
```

Keyword unpacking with **:

```python
def greet(name, greeting):
    return f"{greeting}, {name}!"

data = {'name': 'Alice', 'greeting': 'Hello'}
print(greet(**data))
```

Output:

```
Hello, Alice!
```

Function Definitions with `*args` and `**kwargs`

```
def show_args(*args):
    print(args)

show_args(1, 2, 3)
```

Output:

```
(1, 2, 3)
```

```
def show_kwargs(**kwargs):
    print(kwargs)

show_kwargs(a=1, b=2)
```

Output:

```
{'a': 1, 'b': 2}
```

Mixing Both *args and **kwargs

```
def demo(a, b, *args, **kwargs):
    print(f"a = {a}")
    print(f"b = {b}")
    print(f"args = {args}")
    print(f"kwargs = {kwargs}")

pos = [1, 2, 3, 4]
kw = {'x': 10, 'y': 20}
demo(*pos, **kw)
```

Output:

```
a = 1
b = 2
args = (3, 4)
kwargs = {'x': 10, 'y': 20}
```

Comparing Similar Function Calls

```python
def mixed(a, *rest):
    print(f"a = {a}")
    print(f"rest = {rest}")

l = [1, 2, 3]
a = 0
mixed(a, *l)
```

Output:

```
a = 0
rest = (1, 2, 3)
```

```python
mixed(a, l)
```

Output:

```
a = 0
rest = ([1, 2, 3],)
```

Summary Table

Context	Syntax	What it Does	Example
Assignment	`*var`	collects excess items into a list	`a, *b = [1,2,3] → b=[2,3]`
Function call	`*seq`	unpacks iterable into positional arguments	`f(*[1,2]) → f(1,2)`
Function call	`**dict`	unpacks dictionary into keyword arguments	`f(**{'x':1}) → f(x=1)`
Function definition	`*args`	collects extra positional arguments as tuple	`def f(*args)`
Function definition	`**kwargs`	collects extra keyword arguments as dictionary	`def f(**kwargs)`

Part II

Exam

3 Ex 2024

3.1 Problem 1: Pseudocode

Let A be the adjacency matrix representation of a directed graph G with n nodes. A node u is called *balanced* if the in-degree and out-degree of the nodes are equal. Let N_A be the amount of balanced nodes in the graph G.

a) Describe an algorithm `CountBalanced` in pseudocode, that computes N_A in optimal asymptotic time.
b) Analyse the run-time of `CountBalanced` and show that it is optimal.

3.2 Problem 2: Heaps

In this problem we look at binary Minheaps and carry out depth first search starting form the root, considering two variants:

- `incrDS`: We visit the children of a node in increasing order, sorted w.r.t. the keys
- `decrDS`: We visit the children of a node in decreasing oder, sorted w.r.t. the keys

a) Consider the following Min-Heap represented as an array $H[1..10]$:

Figure 3.1: heap

Provide the keys in the order that they are visited by `decrDS`:

Key-order: ...

b) In the above heap `incrDS` traverses the keys in a globally increasing order, i.e. it effectively 'sorts' the elements. Therefore we try to construct the following "sorting" algorithm:

Let $A[1..n]$ be an array with n keys. (You may assume that the keys are distinct).

1. Transform A into a Min-Heap array representation
2. Traverse A with `incrDS` and output the elements in the order that they are visited

Prove that this algorithm can not always sort correctly. Provide a short analysis on the run-time of this algorithm.

3.3 Problem 3: Hashing

a) Consider the following hash table of size 13:

Figure 3.2: hash 1

You can assume that each empty cells corresponds to $\perp$.

We consider first open hashing with hash function f, defined as:

$$f(k) := 3k + 2 \mod 13$$

and we use linear search. Insert the elements 1, 2, 3, 4, 5 in this order in the provided table.

b) Consider another hash table of size 10:

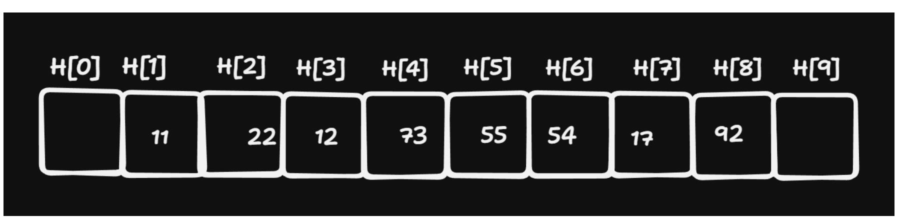

Figure 3.3: hash 2

We again use open hashing this time with the hashing function g defined as:

$$g(k) := k \mod 10$$

Delete the element 22 and provide the state of the hash table after this operation:

3.4 Problem 4: $\mathcal{O}$-Notation and Code Analysis

a) for each of the following cases provide a function $f : \mathbb{N} \to \mathbb{R}$ that satisfies the given asymptotic constraints:

- $f \in \omega\left(\frac{\log x}{\log\log x}\right)$ and $f \in o(\log x)$
- $f \in \omega(\sqrt[3]{m})$ and $f \in o\left(\frac{m}{\log m}\right)$
- $f \in \omega(\log t!)$ and $f \in o(t^2)$

b) Solve the following recurive equations. For all the equations it holds that $T(n) = 1$ and $n \leq 1$. Assume for simplicity that for all equations n is chosen s.t. divisions result without a rest.

- $T(n) = T(n-1) + 2n. \Rightarrow \qquad T(n) \in \Theta(\quad)$
- $T(n) = n \cdot T(n-1). \Rightarrow \qquad T(n) \in \Theta(\quad)$
- $T(n) = 16 \cdot T(\frac{n}{2}). \Rightarrow \qquad T(n) \in \Theta(\quad)$
- $T(n) = 2 \cdot T(\frac{n}{4}) + n. \Rightarrow \qquad T(n) \in \Theta(\quad)$

c) Provide the run-time complexit of the follwoing pseudocode algorithms in Θ notation.

- :

```
read(k)
for i = 1 to k:
 j = k^2
 while j > 1:
    j = j / 3
```

- :

```
read(t)
k = 1
i = 1
while k <= t :
 i = i + 1
 k = k + i
```

- :

```
def fun(n) :
 read(n)
 if n < 10^6 : return
 f(n / 2)
 i = 0
 while i < n : i = i + 1
 f(n / 2)
```

3.5 Problem 5: Short Proofs

Show or refute following propositions:

a) Let $T = (V, E)$ be an arbitrary (a, b)-Tree. Then T is also a $(a, b+1)$-Tree
b) For any weighted, undirected, connected graph $G = (V, E)$ any shortest-path tree is a minimal spanning tree.
c) For any $n > 1000$ there is a binary search tree that stores n keys $\{1, \dots, n\}$

3.6 Problem 6: Induction

For a set X, $\mathcal{P}(A)$ is the set of all subsets of X, called the power set.

Shows via mathematical induction that:

$$\text{for any set } X \text{ s.t. } |X| = n \text{ it holds that } |\mathcal{P}(X)| = 2^n$$

3.7 Problem 7: Graphs

a) Consider the following graph G:

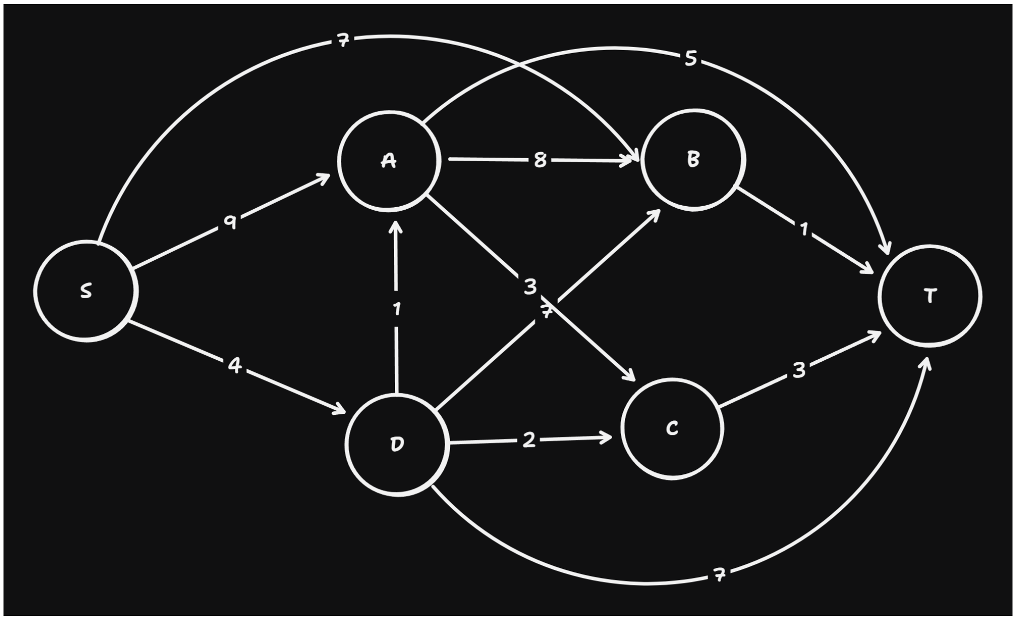

Figure 3.4: graph-1

Carry out the Dijkstra algorithm to find the shortest paths starting from the start node S.

Provide each (Predecessor, Distance)-combination that is assigned to the node T by the algorithm:

b) provide the adjacency field representation of the following graph (CSR). Use the 1 as the smallest index and sort the neighbours in increasing order w.r.t. the index.

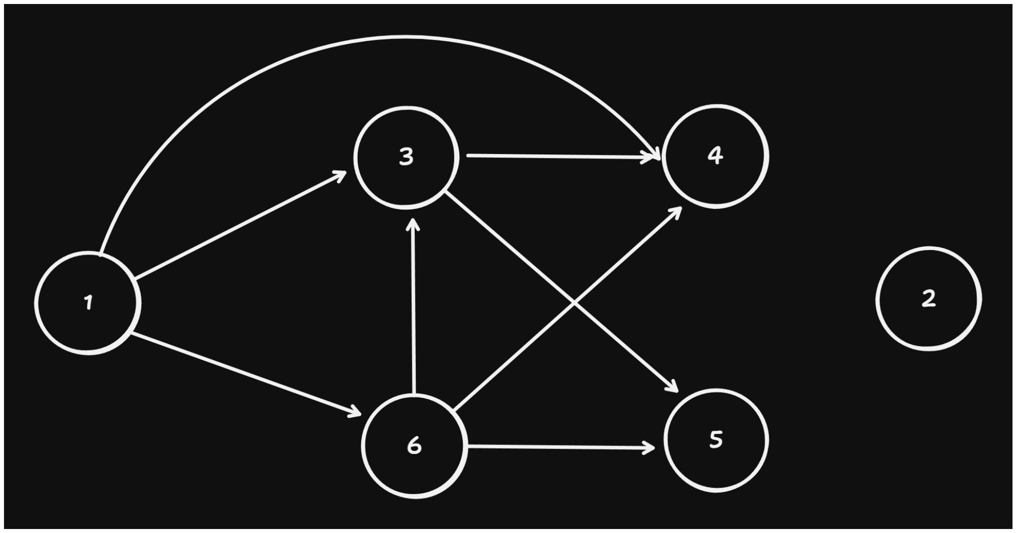

Figure 3.5: graph-2

c) Describe in pseudocode how the k-th neighbor of the i-th node is represented in the adjacency-field representation, using 1 as the smallest index. Name the invariants in the code that guarantee that required entry exists in the data structure. Provide the asymptotic analsysis of the Θ run-time of the algorithm.