optimization

May 15, 2024

1 Lecture 11: Code optimization

- More tricks and algorithms for runtime and memory optimization
- Profiling

1.0.1 Please download the `preprocessing.py` script from OLAT!

1.1 Learning objectives

By the end of this lecture, you should:

- Know how to find bottlenecks in your code
- Know what hash functions are
- Know when and how to use hash functions to so save time or memory
- Be able to implement the reservoir sampling algorithm
- Know how to use caching to optimize runtime
- Know when to use stacks and queues

1.1.1 What happened so far...

Optimizing **runtime**:

- Lecture 6: Time complexity of data structures (`list`, `dict`, `set`)
- Lecture 6: Dynamic programming (Levenshtein distance)

Optimizing **memory usage**:

- Lecture 5+10: Generators
- Lecture 10: Sequential file processing (chunking, `lxml`, `ijson`)
- Lecture 10: Parallelism
- Lecture 10: Memory profiling

1.1.2 In today's lecture...

Optimizing **runtime**:

- Runtime profiling (`cProfile`)
- Hashes
- Caching (`lru_cache`)
- More datastructures (stacks, queues)

Optimizing **memory usage**:

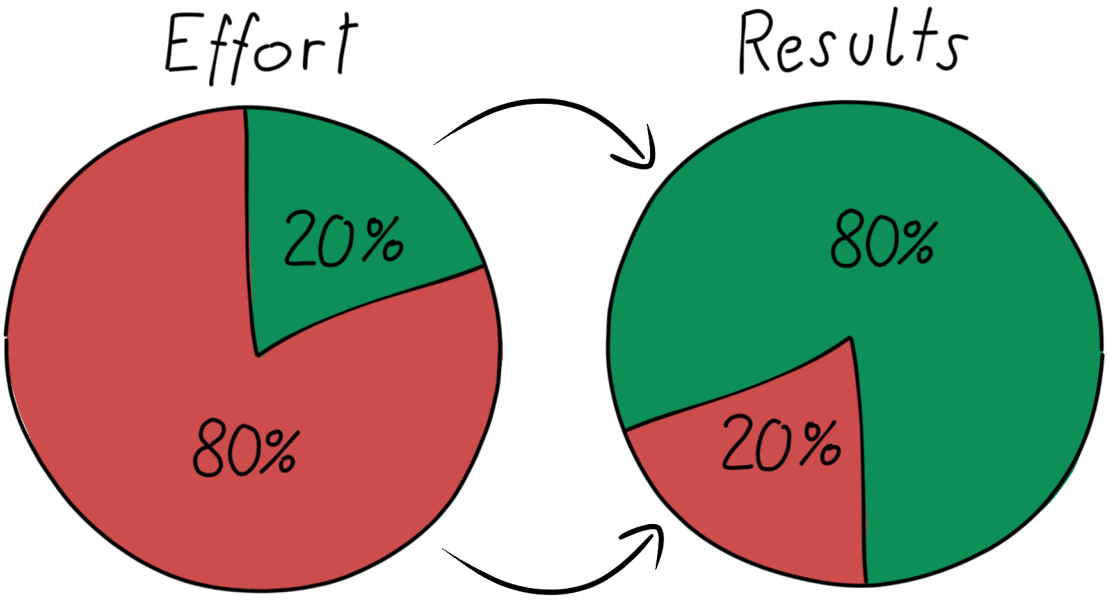
- Hashes
- Reservoir sampling

1.2 Principles for code optimization

``We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.'' -- Donald Knuth, (1970)

Before worrying about performance:

- Make it work
- Make it readable
- Make it maintainable
- Check if optimization is even necessary for your use case



1.2.1 The Pareto Principle

``20% of the effort causes to 80% of the results''

- When optimizing code, there is often a single bottleneck
- Find the bottleneck and forget about the rest
- If the bottleneck can't be optimized, it might not be worth optimizing the rest

1.2.2 Space-time tradeoff

- Optimizing memory usage often leads to slower code
- Optimizing runtime often leads to higher memory usage

→ Know your use case and resource requirements

1.2.3 RAM vs. disk

- Random-access memory (RAM) is fast but limited

- Disk space is abundant but slow to read and write

→ Storing everything in RAM is often faster, but not scalable

1.3 Our example script: `preprocessing.py`

- What does the code do? (Start by looking at the `main` function)
- Are there some obviously inefficient parts?
- Can you guess where the main bottlenecks are?

1.4 Profiling

- **Profiling** means looking at different parts of your code and measuring how long it took or how much memory it used
- Use profiling to find bottlenecks

1.4.1 Measuring runtime of single functions with `cProfile`

```
$ python -m cProfile -s cumtime preprocessing.py
```

```
         53198002 function calls (53197126 primitive calls) in 25.369 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     49/1    0.000    0.000   25.369   25.369 {built-in method builtins.exec}
        1    1.092    1.092   25.369   25.369 preprocess.py:1(<module>)
        1    0.061    0.061   24.232   24.232 preprocess.py:72(main)
        1    4.089    4.089   16.579   16.579 preprocess.py:26(read_segments)
  5935159    3.480    0.000    9.172    0.000 /usr/lib/python3.11/zipfile.py:898(readline)
        1    2.419    2.419    6.789    6.789 preprocess.py:57(write_splits)
   160692    0.654    0.000    4.880    0.000 {function ZipExtFile.readline at 0x7fda195e7100}
   319527    0.251    0.000    3.956    0.000 /usr/lib/python3.11/zipfile.py:914(peek)
   480227    0.464    0.000    3.931    0.000 /usr/lib/python3.11/zipfile.py:932(read)
   160699    0.411    0.000    3.403    0.000 /usr/lib/python3.11/zipfile.py:1014(_read1)
        1    1.315    1.315    2.570    2.570 /usr/lib/python3.11/random.py:376(shuffle)
   160699    2.213    0.000    2.213    0.000 {method 'decompress' of 'zlib.Decompress' objects
  5935165    1.502    0.000    1.502    0.000 {method 'decode' of 'bytes' objects}
  5935045    1.271    0.000    1.271    0.000 {method 'write' of '_io.TextIOWrapper' objects}
  2967519    0.946    0.000    1.254    0.000 /usr/lib/python3.11/random.py:235(_randbelow_wit
  2967579    0.549    0.000    0.916    0.000 <string>:1(<lambda>)
  5935159    0.812    0.000    0.812    0.000 {method 'find' of 'bytes' objects}
        1    0.802    0.802    0.803    0.803 preprocess.py:50(deduplicate_segments)
  5935158    0.728    0.000    0.728    0.000 {method 'strip' of 'str' objects}
       98    0.464    0.005    0.464    0.005 {method '__exit__' of '_io._IOBase' objects}
```

1.4.2 Important caveat when using profilers

- Profilers can slow down your code
- Profilers will take up memory

→ Don't use them in your final code!

1.4.3 Measuring total runtime with `time`

Put the `time` keyword before your command in Bash:

```
$ time python preprocessing.py
python preprocess.py  21.92s user 0.81s system 99% cpu 22.898 total
```

This will be more accurate than the profiler, but it won't tell you where the bottleneck is.

1.4.4 Memory profiling with `memory-profiler`

- Using `@profile` adds significant overhead but gives you the line-by-line memory usage.
- Using `mprof run` is more accurate but only shows total memory usage over time.

```
$ mprof run python preprocessing.py
$ mprof plot
```

1.4.5 Summary

	Total, little overhead	Detailed, significant overhead
Measuring runtime	`time`	`cProfile`
Measuring memory	`mprof`	`@profile`

1.5 Hash functions

Have you ever come across an error like this?

[ ]:
```
my_dict = {
    [1, 2, 3]: "hello",
}
```

Hash functions are what makes lookups in `dict`s and `set`s so fast!

A hash function converts complex objects into simple values with a fixed size (e.g. integers or strings).

The built-in `hash(obj)` is an example of a hash function:

[ ]:
```
my_int = 123
hash(my_int)
```

[ ]:
```
my_str = "123"
hash(my_str)
```

[ ]:
```
my_tuple = (1, 2, 3)
hash(my_tuple)
```

Not all types are hashable:

[ ]:
```
my_list = [1, 2, 3]
hash(my_list)
```

[ ]:
```
my_dict = {"a": 1, "b": 2, "c": 3}
hash(my_dict)
```

When defining your a class, you can override the `__hash__` method to define the hash function for its instances:

[ ]:
```
class Car:
    def __init__(self, color, brand):
        self.color = color            # included in hash
        self.brand = brand            # included in hash
        self.mileage = 0              # NOT included in hash

    def __hash__(self):
        return hash((self.color, self.brand))

    def __eq__(self, other):
        # If `hash(obj1) == hash(obj2)`, then `obj1 == obj2` should also be␣
↪True.
        return isinstance(other, Car) and self.color == other.color and self.
↪brand == other.brand
```

[ ]:
```
my_car1 = Car("red", "Ferrari")
my_car2 = Car("red", "Ferrari")
hash(my_car1), hash(my_car2)
```

[ ]:
```
my_cars = {my_car1, my_car2}
my_cars
```

Important note about custom hashes Hash functions are **pure functions**. The hash value of an object should **never** change across its lifespan. Otherwise, there will be unexpected behavior:

[ ]:
```
ferrari = Car("red", "Ferrari")
prices = {ferrari: 350000}
prices[ferrari]
```

[ ]:
```
print(hash(ferrari))
ferrari.color = "blue"
print(hash(ferrari))
prices[ferrari]
```

→ Usually, hashable types are **immutable** to avoid this (this is why `tuple`s are hashable but `list`s aren't)

1.5.1 Using hashes for equality checking

If we are only interested in checking whether two objects are **equal**, we can just compare their hash values:

[ ]:
```
import random

very_big_tuple1 = tuple(random.random() for _ in range(1000000))
very_big_tuple2 = very_big_tuple1 + (1,)

very_big_tuple1 == very_big_tuple2
```

[ ]:
```
very_big_tuple1_hash = hash(very_big_tuple1)
very_big_tuple2_hash = hash(very_big_tuple2)

very_big_tuple1_hash == very_big_tuple2_hash
```

Comparing the hashes will be much faster:

[ ]:
```
%timeit very_big_tuple1 == very_big_tuple2
```

[ ]:
```
%timeit very_big_tuple1_hash == very_big_tuple2_hash
```

Hash values also take up less memory:

[ ]:
```
from sys import getsizeof

print(getsizeof(very_big_tuple1), "bytes ->", getsizeof(very_big_tuple1_hash),␣
↪"bytes")
print(getsizeof(very_big_tuple2), "bytes ->", getsizeof(very_big_tuple2_hash),␣
↪"bytes")
```

1.5.2 Wait a minute ...

- Two objects that are equal must have the same hash value, but the reverse is not necessarily true
  → Some hash values
- A cryptographic 256-bit hash function will have a 50% chance of collision after generating $2^{128} \approx 10^{38}$ hashes
- Python's built-in `hash()` is much worse, but good enough for most use cases

1.5.3 Use cases

- We cannot get back the original objects once we convert them to hash values
- But we can still use the hash values for equality checking

→ Can you think of scenarios where we can use hash values instead of the original objects?

1.6 Caching

If there are operations whose results we need several times throughout the program, we can **cache** the results in memory instead of computing them repeatedly. For example:

- Storing the return value of pure functions
- Storing part of a database or the results of database queries in memory
- Storing the responses to API requests

A common type is the **least-recently-used** (LRU) cache, which always caches the return values of the last n function calls.

[ ]:
```
from functools import lru_cache

@lru_cache
def factors(number: int) -> list[int]:
    return [
        i
        for i in range(1, number + 1)
        if number % i == 0
    ]
```

[ ]:
```
%time factors(13497459)
```

[ ]:
```
%time factors(random.randint(500000, 1000000))
factors.cache_info()
```

Remember the recursive definition of Levenshtein distance from Lecture 6:

[ ]:
```
def levenshtein(a: str, b: str) -> int:
    if a == "":
        return len(b)
    if b == "":
        return len(a)
    if a[-1] == b[-1]:
        return levenshtein(a[:-1], b[:-1])
    return 1 + min(
        levenshtein(a[:-1], b),
        levenshtein(a, b[:-1]),
        levenshtein(a[:-1], b[:-1]),
    )

%timeit -n 1 -r 1 levenshtein("very looong string", "another very long string")
```

Adding a cache ($\rightarrow$ **memoization**):

[ ]:
```
from functools import lru_cache

@lru_cache(maxsize=100)
def levenshtein_cached(a: str, b: str) -> int:
```

```
    if a == "":
        return len(b)
    if b == "":
        return len(a)
    if a[-1] == b[-1]:
        return levenshtein_cached(a[:-1], b[:-1])
    return 1 + min(
        levenshtein_cached(a[:-1], b),
        levenshtein_cached(a, b[:-1]),
        levenshtein_cached(a[:-1], b[:-1]),
    )

%timeit -n 1 -r 1 levenshtein_cached("very looong string", "another very long␣
 ↪string")
```

1.6.1 Important remarks

- Only use `lru_cache` for **pure functions**!
- By default, `lru_cache` will keep the results of the 128 most recent function calls in memory. You can increase this to infinity:

  ```
  @lru_cache(maxsize=None)
  ```

  But this inevitably results in a **memory leak** and is usually a terrible idea!

1.7 Reservoir sampling

In NLP, we often need to split a corpus into **training**, **development**, and **test sets**.

What is the problem with the following code?

```
import random

with open(f"very-large-file.txt", encoding="utf-8") as file:
    lines = file.readlines()
random.shuffle(lines)

with open("test.txt", "w", encoding="utf-8") as file:
    file.writelines(lines[:1000])
with open("train.txt", "w", encoding="utf-8") as file:
    file.writelines(lines[1000:])
```

How can we do this *without* loading and shuffling all of the data in memory?

- Keep a **reservoir** of 1000 lines (the test set)
- For each line in the corpus, randomly decide to put it in the reservoir or not
- The probability of putting a line in the reservoir continuously decreases

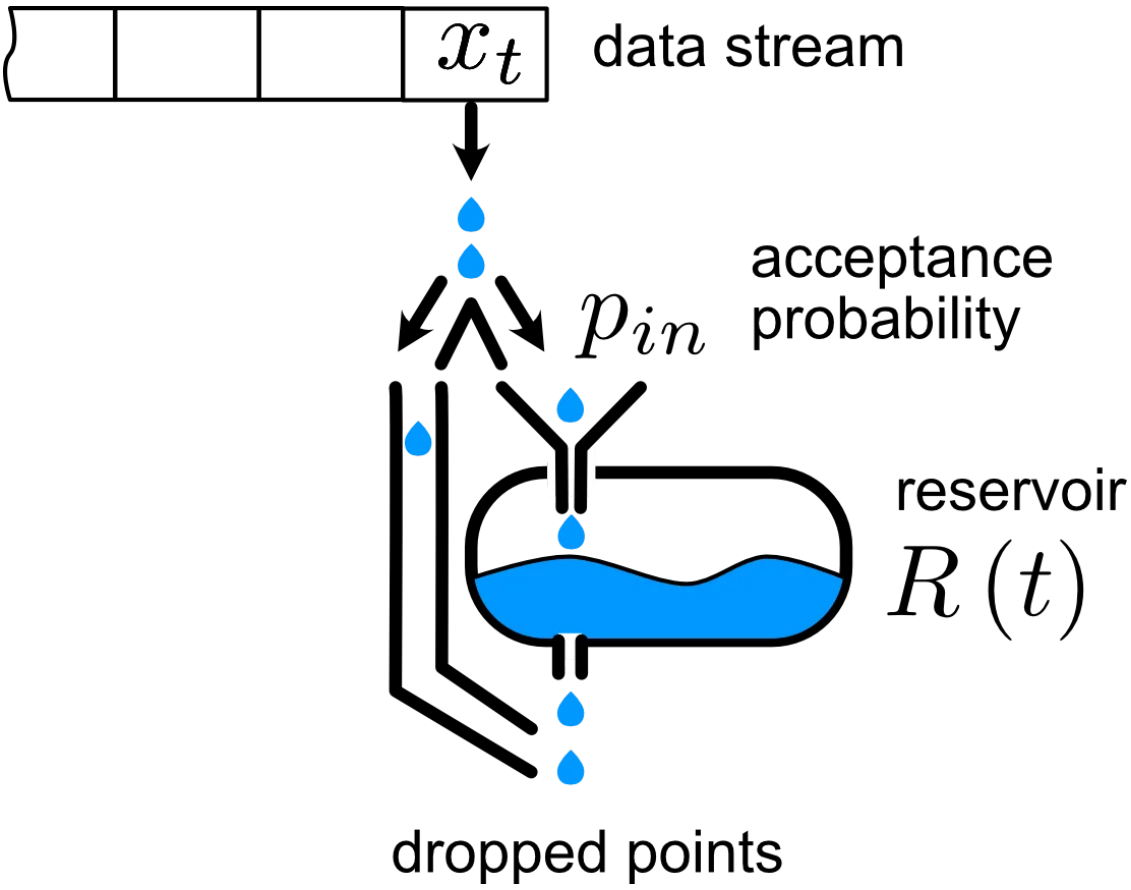


Image source

[ ]:
```
import random
from typing import Iterable

def reservoir_sample(iterable: Iterable, reservoir_size: int):
    reservoir = []
    for t, item in enumerate(iterable):
        if t < reservoir_size:
            reservoir.append(item)
        else:
            # acceptance probability = reservoir_size / t
            random_number = random.randint(0, t)
            if random_number < reservoir_size:
                reservoir[random_number] = item
    return reservoir
```

[ ]:
```
numbers = range(10000)

reservoir_sample(numbers, 10)
```

This allows us to sample a pre-defined number of lines from a file **without knowing how many**

lines there are. Each line will have the same probability of being picked (→ **uniform distribution**).

[ ]:
```
from collections import Counter
import pandas as pd
import seaborn as sns

# Sample 10 numbers from 0 to 99, 100000 times
numbers = range(100)
counts = Counter()
for _ in range(100000):
    counts.update(reservoir_sample(numbers, 10))

# Plot the counts
pd.DataFrame(sorted(counts.items()), columns=["number", "count"])
sns.barplot(x="number", y="count", data=pd.DataFrame(sorted(counts.items()),␣
↪columns=["number", "count"]))
```

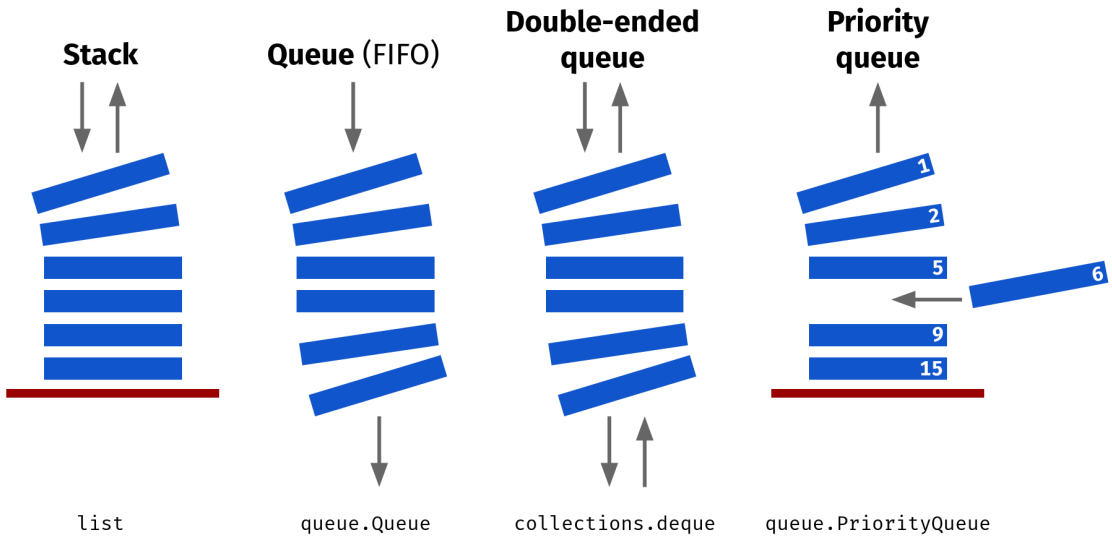
1.7.1 Don't trust the profiler!

Script	Without `cProfile`	With `cProfile`
`preprocess.py`	14.940 sec	26.401 sec
`preprocess_optimized.py`	12.820 sec	30.451 sec

→ Use profilers to find bottlenecks, but don't trust the numbers when comparing different implementations!

1.8 Advanced data structures

Apart from the basic built-in data structures (`list`, `tuple`, `dict`, `set`), Python's standard library provides several other data structures for specific use cases.



1.8.1 Double-ended queue (`collections.deque`)

`deque` works like `list`, but appending and popping at the beginning also have $O(1)$ time complexity.

[ ]:
```
l = ["a", "b", "c", "d", "e"]
l.insert(0, "x")  # O(n)
l.pop(0)  # O(n)
```

[ ]:
```
from collections import deque

d = deque(["a", "b", "c", "d", "e"])
d.appendleft("x")  # O(1)
d.popleft()  # O(1)
```

[ ]:
```
l = list(range(1000000))
%timeit l.insert(0, "x"); l.pop(0)
```

[ ]:
```
d = deque(range(1000000))
%timeit d.appendleft("x"); d.popleft()
```

Example: Keeping the n last lines before a matching line (like `grep -B 5`)

[ ]:
```
from typing import Callable, Generator, Iterable

def search(iterable: Iterable, n: int, match: Callable) -> Generator:
    """Search for the first match in an iterable and return the match and the␣
 ↪previou n items."""
    previous_items = deque(maxlen=n)
    for item in iterable:
        if match(item):
            yield item, list(previous_items)
        previous_items.append(item)
```

[ ]:
```
with open("metamorphosis.txt", encoding="utf-8") as file:
    for match, previous_items in search(file, 5, lambda line: "Gregor" in line):
        print("".join(previous_items), "->", match)
        print("-" * 80, end="\n\n")
```

1.9 Quiz: Code optimization

pwa.klicker.uzh.ch/join/asaeub

1.10 Conclusion: our toolbelt for code optimization

- **Finding bottlenecks:**
  - Profiling (`cProfile`, `memory-profiler`)
  - Memory monitoring
- **Optimizing runtime**:
  - Data structures (`list`, `dict`, `set`, `deque`, …)

  - Dynamic programming (Levenshtein, Dijkstra)
  - Hash functions
  - Caching (`lru_cache`)
  - Parallelism (`multiprocessing`)
- **Optimizing memory usage**:
  - Generators, sequential file processing (`lxml`)
  - Reservoir sampling
  - Hash functions

1.11 Next week's lecture

- Multimodal data processing
- Podcast only

1.12 Final lecture (May 29)

- Information about the exam
- Tell us what topics we should recap and ask questions: https://forms.gle/79oHGwTDXFe1SyPg8

1.13 Help us improve the course!

- Lecture: https://www.uzh.ch/qmsl/en/YJTCC
- Tutorial: https://www.uzh.ch/qmsl/en/7NAAK