Optimization and Maintainability

David Grellscheid
Typical scientific workflow

Correctness is main concern

Start coding without much planning

First version that looks like it works is kept

Sub-optimal choices only noticed later on (if at all)
Typical scientific workflow

Correctness is main concern

A friend of my friend said that you should never do XYZ, because the code will be slower!

Sub-optimal choices only noticed later on (if at all)
Donald Knuth, December 1974:

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

Runtime is not the only factor to consider, need to think about trade off between time spent in:

- development
- debugging
- validation
- portability
- runtime in your own usage
- other developers’ time (now/future)
- total runtime for all users
Runtime is not the only factor to consider, need to think about trade off between time spent in:

- development
- debugging
- validation
- portability
- runtime in your own usage
- other developers’ time (now/future)
- total runtime for all users

CPU time much cheaper than human time!
Optimization points

Someone else already solved (part of) the problem:

LAPACK, BLAS
GNU scientific library
C++ Boost
Numpy, Scipy, Pandas
...

Develop googling skills, evaluate what exists.
Quality often much better than self-written attempts
Optimization points

Choice of programming language

Be aware of what exists

Know strengths / weaknesses

But: needs to fit rest of project

take a look at Haskell, Erlang, JS
findLongestUpTo :: Int -> (Int,Int)
findLongestUpTo mx = maximum ( map f [1 .. mx] )
    where f x = (collatzLength x,x)

collatzLength :: Int -> Int
collatzLength 1 = 1
collatzLength n = 1 + collatzLength (collatzStep n)

collatzStep :: Int -> Int
collatzStep n
    | even n     = n `div` 2
    | otherwise  = 3 * n + 1
Optimization points

Program design

First version: understand the problems
start again

Second version: you know what you’re doing
refactor / clean up / make reusable

Done :-}
Optimization points

Algorithm / data structure choice

can get orders of magnitude in speed

Local and hardware-specific optimisations

- later lecture -
Complexity basics

Much simplified, skipping formal derivation
Complexity basics

Much simplified, skipping formal derivation

```python
while not is_sorted(xs):
    random.shuffle(xs)
```
Complexity basics

Much simplified, skipping formal derivation

```
while not is_sorted(xs):
    random.shuffle(xs)
```

Scaling behaviour with size $N$ of problem set:
- $O(1)$ - constant time independent of $N$
- $O(N)$ - linear with $N$
- $O(N^2)$ - quadratic in $N$
Complexity basics

Much simplified, skipping formal derivation

while not is_sorted(xs):
    random.shuffle(xs)

$O(N N!)$

Scaling behaviour with size $N$ of problem set:
$O(1)$ - constant time independent of $N$
$O(N)$ - linear with $N$
$O(N^2)$ - quadratic in $N$
Merge Sort

2 7 5 1 4 3 6 8
Merge Sort
Merge Sort
Merge Sort
Merge Sort
Merge Sort
Merge Sort
Merge Sort

2 7 5 1 4 3 6 8

1 2 5 7

4 3 6 8

2 7
1 5
3 4
6 8

merge
merge
merge
merge
Merge Sort

```
2 7 5 1 4 3 6 8
```

```
1 2 5 7
```

```
3 4 6 8
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
merge
```

```
Merge Sort

1 2 3 4 5 6 7 8

merge

1 2 5 7

merge

2 7
merge

2 7

1 5
merge

5 1

3 4
merge

4 3
merge

6 8
merge

6 8

3 4 6 8
Merge Sort

$O(N \log N)$
Merge Sort

$O(N \log N)$

15 Sorting Algorithms in 6 Minutes
http://youtu.be/kPRA0W1kECg
http://bigocheatsheet.com/

Data structure complexity
Data Structures

Sequence

10  2  7  5  1  4  9  3  6  8

e.g., C-Arrays, std::vector, std::deque, std::list

Associative

key       value
5          Alpha
3          Beta
7          Gamma

e.g., C-Arrays, std::map, std::set, std::unordered_map
Data Structures

• Operations:
  • Insertion
  • Searching
  • Deletion

• Variants:
  • Ordered
  • Unordered
Sequential Containers

Arrays, Lists, Queues, Stacks
STL Containers

• Sequence Containers
  • vector (flexible sequence)
  • deque (double-ended queue)
  • list (double linked list)
  • array (fixed sequence, C++11)
  • forward_list (single linked list, C++11)
C-Arrays

- Simplest Sequence Data Structure
- Data stored in range [0, numElements)
- Fixed Size, Wasteful
- Consecutive Memory (efficient access)

```c
int a[10000];
int numElements = 0;

// insertion at end O(1)
a[numElements++] = new_value;

// insertion at beginning O(n)
for (int i = numElements; i > 0; i--) a[i] = a[i-1];
a[0] = new_value;
numElements++;
```
```
#include <iostream>
#include <vector>

using namespace std;

// empty construction
vector<int> a;

// sized construction
vector<int> a(10);

// sized construction with initial value
vector<int> a(100, -1);

// C++ 11 initializer lists
vector<int> a { 3, 5, 7, 9, 11 };

// insertion at end
a.push_back(3);
a.push_back(5);
a.push_back(7);

// delete at end
a.pop_back();

// insertion at beginning
a.insert(a.begin(), new_value);

// accessing elements just like arrays
for(int i = 0; i < a.size(); i++) {
    cout << a[i] << endl;
}

// using iterators
for(auto i = a.begin(); i != a.end(); ++i) {
    cout << *i << endl;
}

// C++11 for each
for(auto element : a) {
    cout << element << endl;
}
```
Linked-List

- List Elements connected through pointers
- First Element (head) and last element (tail) are always known
- Insertion/Deletion at both ends in $O(1)$
- Insertion in the middle is also cheaper
  - Finding insertion location is $O(n)$ compared to $O(1)$ with C-Arrays
  - But insertion itself happens in $O(1)$ instead of $O(n)$ copies
- Dynamic Size
- Distributed in memory

**Single Linked-List:**
only pointer of next element

```
10 -> 2 -> 5 -> 7
```

head

**Double Linked-List:**
pointer of previous and next element

```
10 <-> 2 <-> 5 <-> 7
```

head
tail

**struct** Node {
    Node * prev;
    Node * next;
    int data;
}
std::list

```cpp
#include <iostream>
#include <list>

using namespace std;

// empty construction
list<int> a;

// sized construction
list<int> a(10);

// sized construction with initial value
list<int> a(100, -1);

// C++ 11 initializer lists
list<int> a { 3, 5, 7, 9, 11 };

// insertion at beginning
a.push_front(3);

// insertion at end
a.push_back(3);

// delete at beginning
a.pop_front();

// delete at end
a.pop_back();

// access front element
int first = a.front();

// access last element
int last = a.back();

// using iterators
for(auto i = a.begin(); i != a.end(); ++i) { cout << *i << endl; }

// C++11 for each
for(auto element : a) { cout << element << endl; }
```
Queue

• First-In-First-Out (FIFO) data structure

• Implementations:
  • Double-Linked-List

• Operations:
  • **enqueue**: put element in queue (insert at tail)
  • **dequeue**: get first element in queue (remove head)
Stack

• Last-In-First-Out (LIFO) data structure

• Implementations:
  • C-Array
  • Single-Linked-List

• Operations:
  • **push**: put element on stack (insert as first element)
  • **pop**: get first element on stack (remove head)
Associative Containers

Dictionaries, Maps, Sets

key value

5 Alpha
3 Beta
7 Gamma
**Associative Containers**

- Map a key to a value
- Searching for a specific element in unsorted sequential containers takes **linear** time $O(n)$
- Getting a specific element from an associative container can be as fast as **constant** time $O(1)$
STL Containers

- Associative Containers
  - map
  - set
  - multimap
  - multiset
  - unordered_map (C++11)
  - unordered_set (C++11)
  - unordered_multimap (C++11)
  - unordered_multiset (C++11)
C-Array as Associative Container

- Simplest associative data structure
- maps **integer number** to data
  - 0 -> a[0]
  - 1 -> a[1]
  - ...
- efficient access in O(1)
- inefficient storage
- limited to positive integer numbers as keys
Ordered maps

• Maps **arbitrary keys** (objects, basic types) to **arbitrary values** (objects, basic types)

• Basic idea: if keys are sortable, we can store nodes in a data structure sorted by its keys. Sorted data structures can be searched more quickly, e.g. with binary search in $O(\log(n))$

• Elements ordered by key

• **Worst case lookup time is $O(\log(n))$**
std::map

#include <iostream>
#include <map>
#include <string>

using namespace std;

map<string, string> capitals;

// setting value for key
capitals["Austria"] = "Vienna";
capitals["France"] = "Paris";
capitals["Italy"] = "Rome";

// getting value from key
cout << "Capital of Austria: " << capitals["Austria"] << endl;
string & capital_of_france = capitals["France"];
cout << "Capital of France: " << capitals["France"] << endl;

// check if key is set
if (capitals.find("Spain") != capitals.end()) {
    cout << "Capital of Spain is " << capitals["Spain"] << endl;
} else {
    cout << "Capital of Spain not found!" << endl;
}
```cpp
std::map

// iterate over all elements
for (map<string, string>::iterator it = capitals.begin(); it != capitals.end(); ++it) {
    string & key = it->first;
    string & value = it->second;
    cout << "The capitol of " << key << " is " << value << endl;
}

// C++11: iterate over all elements
for (auto it = capitals.begin(); it != capitals.end(); ++it) {
    string & key = it->first;
    string & value = it->second;
    cout << "The capitol of " << key << " is " << value << endl;
}

// C++11: iterate over all elements
for (auto & kv : capitals) {
    string & key = kv.first;
    string & value = kv.second;
    cout << "The capitol of " << key << " is " << value << endl;
}
```
Unordered maps / Hash maps

- Maps arbitrary keys (objects, basic types) to arbitrary values (objects, basic types)
- On average accessing a hash map through keys takes $O(1)$
- In general unordered structure - you can’t get out objects in the same order you inserted them.
- A number, called a hash code, is generated using a hash function based on key in $O(1)$
- Each hash code can be mapped to a location called a bin
- A bin stores nodes with keys which map to the same hash code
- Lookup therefore consists of:
  - Determining the hash code of the key $O(1)$
  - Selecting the correct node inside the bin is in the worst case $O(n)$

On average lookup times are $O(1)$. But this is only true if there are only few hash collisions.

Hash maps require a good hashing function, which reduces the amount of hash collisions.
Reusability is an efficiency!

If the student after you has to start from 0, nothing gained