Competitive Programming I

Computational Complexity and C++ STL

Luís Paquete <paquete@dei.uc.pt>

September 18, 2024

Computational Complexity

Computational Complexity

In competitions we usually care about:

- Runtime complexity
- Memory complexity

Computational Complexity

Complexity is often discuss in Big-Oh notation, which characterizes a function according to their growth rate. We focus on the rate at large input values.

Computational Complexity

Complexity is often discuss in Big-Oh notation, which characterizes a function according to their growth rate. We focus on the rate at large input values.

Essential classes of complexity worth knowing

- $O(1)$ — constant
- $O(\log N)$ — logarithmic
- $O(N)$ — linear
- $O(N \log N)$ — linearithmic
- $O(N^2)$ — quadratic
- $O(N^3)$ — cubic
- $O(2^N)$ — exponential
- $O(N!)$ — factorial

Consider the following problem:

We have a sorted vector V with N integers and want to find whether an integer x exists in V or not.

Solutions?

Algorithm Analysis — Example 1

```
bool solve(vector<int> const& v, int x) {
  for (auto const& val : v) {
    if (val == x) {
      return true;
    }
  }
  return false;
}
```

Runtime complexity?

(Extra) Memory complexity?

Algorithm Analysis — Example 1

```cpp
bool solve(vector<int> const& v, int x) {
  for (auto const& val : v) {
    if (val == x) {
      return true;
    }
  }
  return false;
}
```

Runtime complexity? $O(N)$

(Extra) Memory complexity? $O(1)$

Algorithm Analysis — Example 1

```cpp
bool solve(vector<int> const& v, int x) {
  return std::binary_search(v.begin(), v.end(), x);
}
```

Runtime complexity?

(Extra) Memory complexity?

Algorithm Analysis — Example 1

```
bool solve(vector<int> const& v, int x) {
  return std::binary_search(v.begin(), v.end(), x);
}
```

Runtime complexity? $O(\log N)$

(Extra) Memory complexity? $O(1)$

Algorithm Analysis — Example 2

Consider the following problem:

Given a number N find the number of distinct factors of N. Examples:

- $N = 8$ has 4 distinct factors (1, 2, 4, 8)
- $N = 12$ has 6 distinct factors (1, 2, 3, 4, 6, 12).

Solutions?

Algorithm Analysis — Example 2

```
size_t solve(int n) {
  vector<int> v;
  for (int i = 1; i <= n; ++i) {
    if (n % i == 0) {
      v.push_back(i);
    }
  }
  return v.size();
}
```

Runtime complexity?

(Extra) Memory complexity?

Algorithm Analysis — Example 2

```
size_t solve(int n) {
  vector<int> v;
  for (int i = 1; i <= n; ++i) {
    if (n % i == 0) {
      v.push_back(i);
    }
  }
  return v.size();
}
```

Runtime complexity? $O(N)$

(Extra) Memory complexity? $O(N)$

Algorithm Analysis — Example 2

```
size_t solve(int n) {
  size_t count = 0;
  for (int i = 1; i <= n; ++i) {
    if (n % i == 0) {
      count += 1;
    }
  }
  return count;
}
```

Runtime complexity?

(Extra) Memory complexity?

Algorithm Analysis — Example 2

```
size_t solve(int n) {
  size_t count = 0;
  for (int i = 1; i <= n; ++i) {
    if (n % i == 0) {
      count += 1;
    }
  }
  return count;
}
```

Runtime complexity? $O(N)$

(Extra) Memory complexity? $O(1)$

Algorithm Analysis — Example 2

```
size_t solve(int n) {
  size_t count = 0;
  for (int i = 1; i*i <= n; ++i) {
    if (n % i == 0) {
      count += 2;
    }
  }
  return count;
}
```

Is this correct?

Algorithm Analysis — Example 2

```
size_t solve(int n) {
  size_t count = 0;
  for (int i = 1; i*i <= n; ++i) {
    if (n % i == 0) {
      count += 2;
    }
  }
  return count;
}
```

Is this correct?

What if the number has an integer square root? For example for 25, this code would say there are 4 distinct factors, when in reality there are only 3 (1, 5, 25). What is the problem?

Algorithm Analysis — Example 2

```
size_t solve(int n) {
  size_t count = 0;
  int i = 1;
  for (; i*i < n; ++i) {
    if (n % i == 0) {
      count += 2;
    }
  }
  if (i*i == n) {
    count += 1;
  }
  return count;
}
```

Runtime complexity?

(Extra) Memory complexity?

Algorithm Analysis — Example 2

```
size_t solve(int n) {
  size_t count = 0;
  int i = 1;
  for (; i*i < n; ++i) {
    if (n % i == 0) {
      count += 2;
    }
  }
  if (i*i == n) {
    count += 1;
  }
  return count;
}
```

Runtime complexity? $O(\sqrt{N})$

(Extra) Memory complexity? $O(1)$

Algorithm Analysis

We will go over more examples in future classes

Rules of thumb — Time complexity

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g., Enumerating permutations (Section 3.2)
$\leq [17..19]$	$O(2^n \times n^2)$	e.g., DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g., DP with bitmask technique (Book 2)
$\leq [24..26]$	$O(2^n)$	e.g., try 2^n possibilities with $O(1)$ check each
≤ 100	$O(n^4)$	e.g., DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 450	$O(n^3)$	e.g., Floyd-Warshall (Section 4.5)
$\leq 1.5K$	$O(n^{2.5})$	e.g., Hopcroft-Karp (Book 2)
$\leq 2.5K$	$O(n^2 \log n)$	e.g., 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g., Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 200K$	$O(n^{1.5})$	e.g., Square Root Decomposition (Book 2)
$\leq 4.5M$	$O(n \log n)$	e.g., Merge Sort (Section 2.2)
$\leq 10M$	$O(n \log \log n)$	e.g., Sieve of Eratosthenes (Book 2)
$\leq 100M$	$O(n), O(\log n), O(1)$	Most contest problem have $n \leq 1M$ (I/O bottleneck)

Table 1.4: Rule of Thumb for the 'Worst AC Algorithm' for various single-test-case input sizes n, assuming that a year 2020 CPU can compute $100M$ operations in 1 second.

Taken from: Competitive Programming 4 — Book 1 — Section 1.3.3

C++ STL

Arrays (Static and Dynamic)

- Contiguous block of objects (cache friendly)
- $O(1)$ for lookup by index
- $O(1)$ for insertion and deletions at the back
- $O(n)$ for insertions and deletions anywhere else
- Can be stored in the stack (static) or the heap (dynamic)
- Stack is preferable if you know the size of the array

Arrays - C++

```cpp
// Static size array saved on the stack
#include <array>
std::array<int, 3> arr;
arr[i]; // O(1)

// Dynamic size array saved on the heap
#include <vector>
std::vector<int> vec();
vec[i];               // O(1)
vec[i] = ...;         // O(1)
vec.push_back(int); // O(1) amortized
vec.pop_back();       // O(1)
```

Deque

- An array of arrays
- $O(1)$ insertion and deletion at the front and back
- $O(n)$ insertion and deletion anywhere else
- Small penalty in terms of random access (compared to vector/array)
- In `C++` you can use `std::deque`

Linked Lists

- List of dinamically allocated objects (not cache friendly)
- $O(1)$ insertions and deletions at the front and back
- $O(1)$ access, insertion, and deletion with an iterator to the desired position
- $O(n)$ if we need to lookup by index for access, insertion, or deletion
- Useful whenever there are many insertions or deletions in the middle or at the beginning, and we have quick access to the position where we want to insert/delete

Linked Lists - C++

```cpp
#include <list>

std::list<int> l();
l.push_back(int);            // O(1)
l.pop_back();                // O(1)
l.push_front(int);           // O(1)
l.pop_front();               // O(1)
l.insert(iterator, int);     // O(1)
l.erase(iterator);           // O(1)
```

There is also `std::forward_list<T>` that can only move forward

Stacks and Queues

- Stacks are used to implement last-in first-out (LIFO) mechanisms, which can be useful, for example, to simulate recursion
- Queues are used to implement first-in first-out (FIFO) mechanisms
- They can be easily implemented with arrays and deques or linked lists. Alternatively in the `STL` you also have:
- `std::stack` for a stack
- `std::queue` for a queue

Self-balancing binary trees

- $O(\log n)$ for insertion, deletion and lookup
- Are often tricky to implement (luckily C++ does it for us)

Self-balancing binary trees - C++

```cpp
#include <set>

std::set<int> s();
s.insert(int);      // O(log n)
s.find(int);        // O(log n)
s.lower_bound(int); // O(log n) smallest element >= inpu
s.upper_bound(int); // O(log n) smallest element >  inpu
s.erase(int);       // O(log n)
```

There is also `std::map<K, V>` for different keys and values. Also, you have `std::multiset<V>` and `std::multimap<K, V>` if you need to support repeated values in the tree.

Heap Priority Queue

- The minimum (or maximum) element can be accessed or removed in $O(1)$, insertions are $O(\log n)$
- You can use `std::priority_queue` in C++

Hash Tables

- $O(1)$ for insertion, deletion and lookup on average
- $O(n)$ in the worst case
- A good hashing function is very important

Hash Tables - C++

```cpp
#include <unordered_set>

std::unordered_set<int> s;
s.insert(int);  // O(1) best, O(n) worst
s.find(int);    // O(1) best, O(n) worst
s.erase(int);   // O(1) best, O(n) worst
```

There is also `std::unordered_map<K, V>` for different keys and values.

Summary - Usage guidelines

Use static size arrays (`std::array`) when:

- You want a fixed number of items
- You want a stack-allocated array (when supported by the programming language)

Use dynamic size arrays (`std::vector`) when:

- You want a re-sizable array
- You want an heap-allocated array
- You will mostly be inserting and removing from the end (or near the end)

Summary - Usage guidelines

Use a deque (`std::deque`) when:

- You want a dynamic size array, but
- You want to insert/remove from both ends of the array (or near them)

Use linked lists (`std::list`, `std::forward_list`) when:

- You need to constantly insert and remove from the middle of array in known locations
- You don't need random access by index

Summary - Usage guidelines

Use priority queues (`std::priority_queue`) when:

- You only ever need the largest or smallest element of the collection

Use self-balancing binary trees (`std::set`, `std::map`) when:

- You want to keep elements sorted
- You need to find if elements exists or not in a list
- You need to find the elements that are closest to another

Use hash tables (`std::unordered_set`, `std::unordered_map`) when:

- You need to see if elements exist or not in a list and you do not care about order

Summary

- The right choice for a data structure depends on how many insertions, deletions, or lookups have to be performed
- There are other, more specific, data structures (segment trees, tries, suffix trees, ...) that are not covered by the `STL`. However, these should cover most standard cases.
- Explore `https://en.cppreference.com/w/cpp/container`

Summary

Container	Access	Insert	Erase	Find
Array	$O(1)$: Index	N/A	N/A	$O(n)$
Vector	$O(1)$: Index	$O(1)$: Back $O(n)$: Other	$O(1)$: Back $O(n)$: Other	$O(n)$
Deque	$O(1)$: Index	$O(1)$: Back/Front $O(n)$: Other	$O(1)$: Back/Front $O(n)$: Other	$O(n)$
List	$O(n)$: Index	$O(1)$: Iter/B/F $O(n)$: Index/Other	$O(1)$: Iter/B/F $O(n)$: Index/Other	$O(n)$
Set/Map	$O(\log n)$: Key $O(n)$: Index	$O(\log n)$	$O(\log n)$	$O(\log n)$
Hash Set/Map	$O(1)$: Avg $O(n)$: Worst	$O(1)$: Avg $O(n)$: Worst	$O(1)$: Avg $O(n)$: Worst	$O(1)$: Avg $O(n)$: Worst

Algorithms

The STL also includes many common simple algorithms, e.g.

- `shuffle` — shuffle elements in a sequence (e.g. vector)
- `sort` — sort elements in vectors, arrays, or deques
- `lower_bound` / `upper_bound` — find lower/upper bound on sorted sequences
- `binary_search` — check if a value exists in O(log n) on sorted sequences
- `merge` — merge two sorted sequences
- `max` / `min` — find max/min value in sequence
- `equal` — check if two sets of elements are equal

And many others, explore
`https://en.cppreference.com/w/cpp/header/algorithm`

Iterators

Many algorithms in `std::algorithm` use iterators to define the range of values that they apply to.

We can think of iterators as a pointer to the value, e.g.

- `std::vector<T>::begin()` — returns an iterator that points to the first element of the vector
- `std::vector<T>::end()` — returns a special iterator that has no value, but represent the end of the object

If `begin() == end()` it means that the vector is empty.

Iterators

Using iterators

- `std::sort(v.begin(), v.end())` — sort the whole vector from beggining to end
- `auto it = set.find(value)` — returns an iterator that points to the position with the value. If the value does not exist, `it` will be equal to `set.end()`
- `++it` increments the iterator, `--it` decrements the iterator
- `std::next(it)` returns the following iterator, `std::prev(it)` returns the previous iterator

Iterators

Accessing the value associated with an iterator can done with the *
operator

Code

```
vector<int> v{4,3,2,1};
auto it = v.begin();
cout << *it << "\n";
++it;
cout << *it << "\n";
cout << *std::next(it) << "\n";
```

Result

```
4
3
2
```

Iterators

Looping with iterators

Code

```
vector<int> v{4,2,3,1};
for (auto it = v.begin(); it != v.end(); ++it) {
        cout << *it << "\n";
}
```

Result

```
4
2
3
1
```

Iterators

Looping with iterators (implicitly)

Code

```
vector<int> v{4,2,3,1};
for (auto value : v) {
        cout << value << "\n";
}
```

Result

```
4
2
3
1
```

Iterators

For `map` and `unordered_map` iterators return a pair of (key, value)

Code

```
map<int, char> m;
m.emplace(1, 'a');
m.emplace(2, 'b');
m.emplace(3, 'c');

for (auto it = m.begin(); it != m.end(); ++it) {
   cout << it->first << " " << (*it).second << "\n";
}
```

Result

```
1 a
2 b
3 c
```

Iterators

Check more about iterators

- `https://en.cppreference.com/w/cpp/iterator`
- `https://en.cppreference.com/w/cpp/container#Iterator_invalidation`

Algorithms with custom functions

Some algorithms make use of custom functions. Sort for example

Code

```
struct Point {
  int x, y;
};

vector<Point> v{{2,2}, {0,3}, {1,5}, {0, 2}};

// Sort first by x, and in case of tie by y
sort(v.begin(), v.end(), [](auto const& lhs, auto const& rhs) {
  return lhs.x < rhs.x || (lhs.x == rhs.x && lhs.y < rhs.y);
});

for (auto const& p : v) {
   cout << p.x << ", " << p.y << "\n";
}
```

Algorithms with custom functions

Result

```
0, 2
0, 3
1, 5
2, 2
```

Define custom operators

Another alternative is to define operators for the object

Code

```
struct Point {
int x, y;
bool operator<(Point const& other) const {
  return x > other.x || (x == other.x && y > other.y);
} };

vector<Point> v{{2,2}, {0,5}, {1,5}, {0, 2}};
// Sort using the operator< function by default
sort(v.begin(), v.end());

for (auto const& p : v) {
        cout << p.x << ", " << p.y << "\n";
}
```

Define custom operators

Result

```
2, 2
1, 5
0, 3
0, 2
```

Conclusion

C++ is a powerful language where many things are possible

However, it is also a big language with many advanced concepts

Explore the documentation
`https://en.cppreference.com/w/cpp`

And ask us!