31251 – Data Structures and Algorithms

Week 3, Autumn 2020

Xianzhi Wang

In this week's episode:

- Vectors & List
- Templates
- Iterators
- Big-O notation for algorithm analysis

- Arrays are easy to use but unsuitable to be changed dynamically.
- C++ offers two alternative data structures:
  - vector: `#include <vector>`
  - list: `#include <list>`
- Reference: https://thispointer.com/difference-between-vector-and-list-in-c/

- `array`
  - Memory: fixed size, contiguous
  - Features: random access, shift upon deletion
- `vector`
  - Memory: dynamic size, contiguous
  - Implementation: based on `array`
  - Features: random access, shift upon deletion
- `list`
  - Memory: dynamic size, non-contiguous
  - Implementation: based on `doubly linked list`
  - Features: efficient insertion and deletion
- Practice 1

Containers are objects that store data.

- Simple containers:
  - `pair`: simple 2-tuple store
- Sequence containers (ordered collection):
  - `vector`: dynamic array
  - `deque`: double-ended queue
  - `list`: doubly linked list
- Associative containers (unorder collection):
  - `set`: a mathematical set
  - `map`: key-value store
  - `multiset`, `multimap`: allow duplicated elements/keys
  - `hash_set`, `hash_map`, `hash_multiset`, `hash_multimap`: the above containers implemented using `hash table`.

Container Adaptors are classes that use an encapsulated object of a specific container class as its underlying container, providing a specific set of member functions to access its elements.

- `queue`: based on `deque` or `list`
- `priority queue`: based on `vector` or `deque`
- `stack`: based on `deque`, `vector`, or `list`

An example:

```
queue<int, list<int> > q;
```

- Functions used with `vector` and `list`:
  - Iterator: a means of enumerating all elements of the container.
  - Capacity: get or adjust the size of the container.
  - Element access: get an element at a specific position.
  - Modifier: add, remove, replace, or swap elements.
- References:
  - `https://www.geeksforgeeks.org/vector-in-cpp-stl/`
  - `https://www.geeksforgeeks.org/list-cpp-stl/`
- Practice 2

- Wait... what that thing in the angle brackets (<>)?
- If you've used generics in Java, this is the C++ version: templates!
- Templates provide a way to write certain types of code once:
  - If the code doesn't care about the types it's working with.
- So we can easily make each `vector` hold a different data type without rewriting the code.
- Practice 3

Iterators

- Iterators (or limited pointers) are used to point at the memory addresses of STL containers.
- They are primarily used in sequence of numbers, characters, etc.
  - `iterator`: random access containers.
  - `bidirectional iterator`: non random access containers.
- They reduce the complexity and execution time of program.

Why use iterators?

- A flexible way to access data in containers that don't have obvious means of accessing all of the data (e.g., `maps`).
- STL algorithms defined in `<algorithm>` use iterators.

The gotchas?

- No boundary check.
- Can be invalidated if the underlying container is changed significantly.

Practice 4

If we have two algorithms that solve the same problem, what things are we interested in?

- Time Complexity: how long they take.
- Space Complexity: How much space they take up.

How do we reliably compare algorithms?

- Testing gives good information, but is limited to the cases you test and can be resource-intensive.
- How much of that information comes from the choice of computer, programming languages, or test data?

- How do we know what resources an algorithm will use for a huge number, or even an infinite number of inputs?
- We need a way of comparing algorithms using *lower-bound*, *best estimate*, or *upper-bound* measures.
- In practice, we are interested in the upper-bound of algorithms' time complexity, called big-O notation.

Given two functions f and g, we say f is in big-O of g (denoted by $f \in O(g)$) if:

$$\exists c \in \mathbb{R}^{+}, N \in \mathbb{N} \text{ such that } \forall n \geq N, \text{ we have } f(n) \leq c \cdot g(n).$$

That means, given a big enough number n, $f(n)$ is less than or equal to a constant times of $g(n)$.

- $f(n) = n,\ g(n) = 2n \quad \rightarrow \quad f \in O(g)$.
- $f(n) = n,\ g(n) = n^2 \quad \rightarrow \quad f \in O(g)$.
- $f(n) = n^2,\ g(n) = n \quad \rightarrow \quad f \notin O(g)$ (but $g \in O(f)$).
- $f(n) = 50n,\ g(n) = n \quad \rightarrow \quad f \in O(g)$ (and $g \in O(f)$).
- $f(n) = \log n,\ g(n) = n \quad \rightarrow \quad f \in O(g)$.
- $f(n) = n,\ g(n) = 2^n \quad \rightarrow \quad f \in O(g)$.
- $f(n) = 23n^3,\ g(n) = 13n^4 \quad \rightarrow \quad f \in O(g)$.

- These can all be proved using a variety of techniques:
  - Induction
  - Algebraically
  - Limit based definitions.
- Leaving out the proofs, there are a couple of handy rules:
  - $c \cdot n^k \in O(n^{k+1})$ for any c and k.
  - $\log n \in O(n)$.
  - $f(n) + g(n) + h(n) + \ldots \in O(\max\{f(n), g(n), h(n), \ldots\})$.
  - You can always ignore constants, i.e., $c \cdot f(n) == f(n)$.

If

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

then $f \in O(g)$ (where $< \infty$ means any constant or $-\infty$).

Suppose k is a constant, then

- n is "linear".
- n^2 is "quadratic".
- n^3 is "cubic".
- $\log n$ is "logarithmic".
- $(\log n)^k$ for any k is "poly-logarithmic".
- n^k for any k is "polynomial".
- k^n for any k is "exponential".

We use $f = O(n)$ to denote f has linear (time) complexity.

- Any complexity $< O(n)$ is called "sub-linear".
- Linear and sub-linear algorithms have been a constant pursuit in algorithmic research.

How Does This Help Us?

- Back to the two algorithms $\mathcal{A}$ and $\mathcal{B}$:
  - If we can work out functions that describe the running time, we can now compare them and decide which is the fastest (in the long run).
- Given an input size n, if the running time of $\mathcal{A}$ is $T_{\mathcal{A}}(n) = n^2$, and the running time of $\mathcal{B}$ is $T_{\mathcal{B}}(n)$, then we can work out that $T_{\mathcal{B}} \in O(T_{\mathcal{A}})$, i.e., $\mathcal{B}$ is the faster algorithm *asymptotically*.
- $\mathcal{A}$'s running time is always longer for large enough inputs.

- How do we get these functions then?
- In the abstract sense, running time is really the number of steps the algorithm takes for a given input size.
    - This abstracts out programming languages and computers.
- So "all" we need to do is count the number of steps.

```
int main()
{
  a = 1;
  b = 2;
  c = a + b;

  cout << c;
}
```

This code does the same thing for any “input” (it doesn’t really take any), so $T(n) = 4$.

```
void printArray(int a[], size n){
  for (int i = 0; i < n; i++){
    cout << a[i];
  }
}
```

We initialise `i` once and then do n iterations, each

1. checking whether `i` is large enough to stop,
2. printing something out, and
3. adding one to `i`.

Assuming printing is one step, $T(n) = 1 + 3n \in O(n)$.

```
for (int i = 0; i < n; i++){
  for (int j = 0; j < n; j++){
    cout << i << " " << j;
  }
}
```

For the outer loop, we have

- 1 initialisation and n iterations;

Each outer iteration has an inner loop, each having

- 1 initialisation and n iterations; and
- 1 printing for each inner iteration.

Therefore, $T(n) = (n+1) \cdot (n+1) \cdot 1 = n^2 + 2n + 1 \in O(n^2)$.

We care about which algorithm's time consumption grows faster:

1. $O(n^k) < O(n^{k+c})$, s.t., $c > 0$
2. $O(\log n) < O(n^k)$, s.t., $k > 0$
3. $O(n^k) < O(c^n)$, s.t., $k \geq 0, c > 1$
4. $O((\log n)^k) < O(n \log n)$, s.t., $k \geq 0$

Based on the 2nd rule, we have:

5. $O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^2 \log n) < \cdots$

- $O(\cdot)$ is transitive:
  - If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$.
- Equivalence relation:
  - If $f \in O(g)$ and $g \in O(f)$, then $f \in \Theta(g)$ (or $g \in \Theta(f)$).
- The opposite of big-O:
  - If $f \in O(g)$, then $g \in \Omega(f)$.
- Variants of notations:
  - o replaces O if we use $<$ (not $\leq$) in the definition (slide# 13).
  - ω replaces Ω if we use $>$ (not $\geq$) in its definition.