Tutorial 8
Algorithm Analysis And Sorting Algorithms

Zheng Hanjun (USTF)
(SDS, 122090797)

Contents

- A simple review of time complexity
- More on sorting algorithms

A Quick Recap of Complexity

Assume we want to calculate the sum of integers from 1 to 100.

- **Naive Way**

1 + 2 + 3 + 4 + … + 99 + 100

- **Another Method**

(1+100)*100/2

Assume addition takes 1 unit of time and multiplication/division takes 10 units of time.

The naive way needs 99 units of time

The second method needs 21 units of time

A Quick Recap of Complexity

Assume we want to calculate the sum of integers from 1 to N.

- **Naive Way**

1 + 2 + 3 + 4 + … + (N-1) + N

- **Another Method**

(1+N)*N/2

Assume addition takes 1 unit of time and multiplication/division takes 10 units of time.

The naive way needs (N-1) units of time

The second method needs 21 units of time

A Quick Recap of Complexity

Which one is better?

I think most people will agree the later one is a better algorithm.

When N is small, (N-1) might be smaller 21.

When N is large, (N-1) will definitely be much larger than 21.

And in computer science, we always deal with cases when N is large

When $N \to \infty$, all other terms are negligible:

$N-1 \approx N$

$21 \approx 1$

Precise Definition and Notations

More precisely, we can say: N-1 = Θ (N) and 21 = Θ (1)

Big-Theta Notation: For $f(n)$ and $g(n)$, we say $f(n) = \Theta(g(n))$, iff $\exists c_1, c_2, n_0 > 0, \text{s.t.} \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n).$

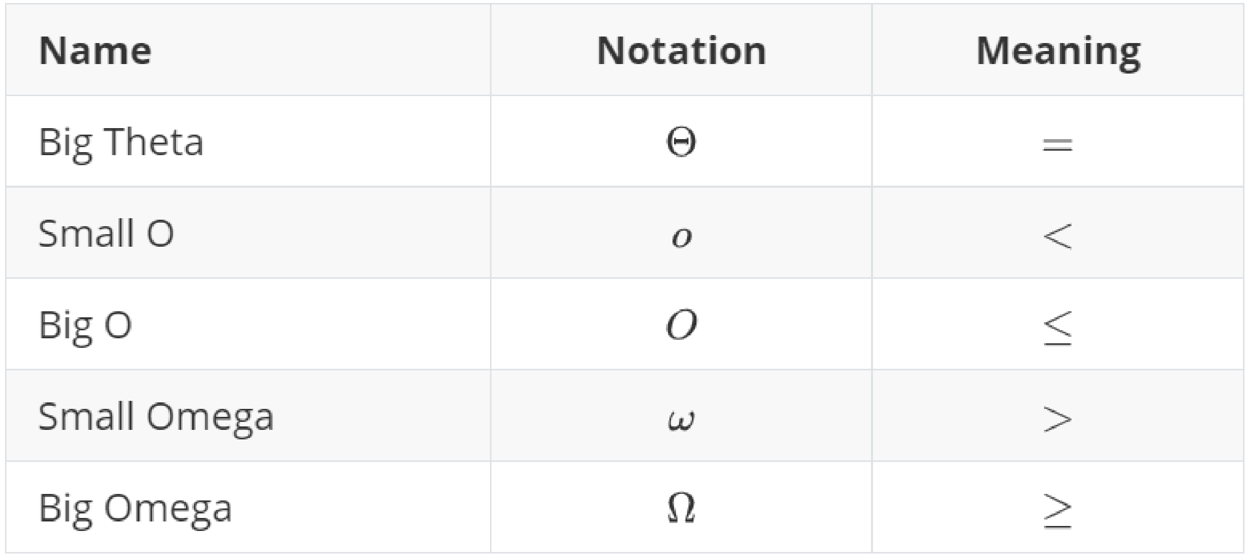
That basically means f(n) and g(n) are of the same “order”.

Example: $2N^3 - 5N^2 + 210 = \Theta(N^3)$

Similarly, there are some other notations: O, o, Ω, ω

Precise Definition and Notations

An easy way to help you remember them:



Name	Notation	Meaning
Big Theta	Θ	$=$
Small O	o	$<$
Big O	O	$\leq$
Small Omega	ω	$>$
Big Omega	Ω	$\geq$

Conventionally, we use the “Big-O” notation at most of the time, because we care more about the upper bound of the complexity.

Average Complexity

For most algorithms, the number of basic operations it needs isn’t always the same.

For example: Linear Search

If we are lucky, the element we want is at the first position, then it only requires 1 access.

If we are unlucky, we might need N access to find the element.

→ Average time complexity (Expectation)

$$1 \times \frac{1}{N} + 2 \times \frac{1}{N} + \cdots + N \times \frac{1}{N} = O(N)$$

Amortized Complexity

Consider the push_back() method in std::vector

1. Initialized with a certain length of memory space.
2. If there is still room, directly put the new element in the end. $O(1)$
3. If the space is full, apply for a space with doubled size and copy every old elements into the new location. $O(m)$

For totally m operations:

The total cost of insertion: $1 \times m = O(m)$

The total cost of copying: $1 + 2 + 4 + \cdots + 2^{k-1} = 2^k - 1 = O(m)$

The amortized cost: $O(1)$

Sorting Algorithms

Sorting algorithms you learned in the lectures:

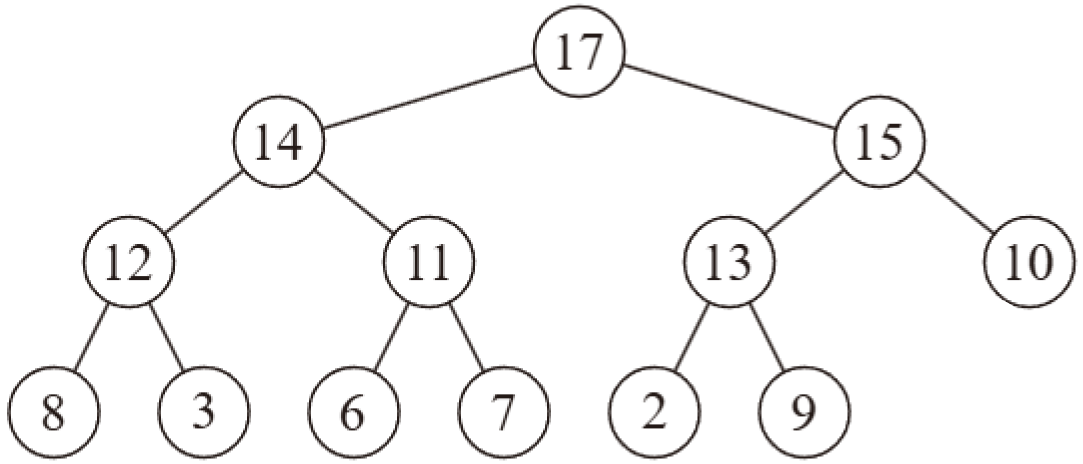
Algorithm	Average Complexity	Worst Case Complexity
Selection Sort	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n^2)$

Heap Sort

Heap sort is a sorting algorithm utilizing the binary heap data structure.

In a max heap, it keeps two properties:

1. It is a complete binary tree
2. Heap Property: The key value of any parent node is greater than or equal to that of its child nodes. (Similarly we can define a min heap)



Heap Operations

A max(min) heap support these operations:

Operation	Complexity
Find Max(Min)	$O(1)$
Delete Max(Min)	$O(\log n)$
Insertion	$O(\log n)$

Find Max: Just get the top element

Insertion: (1) Put the new element at the end (2) Swap it up

Delete Max: (1) Swap the top element and the last one (2) Delete the last element
(3) Swap the top element down

Use these operations, we can easily develop an $O(n \log n)$ sorting algorithm.

Find the k-th element

Given an unsorted array of n elements, find the k-th smallest element.

A naive way:

(1) Sort the given array. $\mathrm{O}(n \log n)$ (if we adopt quick sort)

(2) Simply get the k-th element. $O(1)$

We only need to find the k-th element, but we waste a lot of time sorting other elements. Can we do better?

Find the k-th element

Let’s take a look at the process of quick sort:

(1) Divide the array into two parts and guarantee that all elements in the first part is smaller than that in the second part.

(2) Sort both two part.

Find the k-th element

Let’s take a look at the process of quick sort:

(1) Divide the array into two parts and guarantee that all elements in the first part is smaller than that in the second part.

(2) Sort both two part. ← Problem is here

Find the k-th element

Let's take a look at the process of quick sort:

(1) Divide the array into two parts and guarantee that all elements in the first part is smaller than that in the second part.
(2) Determine which part the k-th element is in.
(3) Only sort one part that we need.

Thus, we can reach an average time complexity of $O(n)$

C++ Standard Library

For the algorithms mentioned above, you don’t need to implement them by yourself. They’re already provided in the C++!

We will introduce some of the fundamental usage of them. But for more details, you should learn by yourself.

STFW

RTFM

RTFSC

std::priority_queue

Header file: #include <queue>

std::priority_queue provide similar functionality to a heap and is usually implemented by heap. By default, it create a “max heap” based on vector.

Usage	Meaning
`std::priority_queue<int> pq`	Declare a "max heap" of integers
`pq.empty()`	Check if it's empty
`pq.push(x)`	Push x into the "heap"
`pq.top()`	Get the max value
`pq.pop()`	Delete the max value

std::sort

Header file: #include <algorithm>

Usage: std::sort(first, last) or std::sort(first, last, cmp)

It sort the element in [first, last), and the default order is from smaller ones to greater ones.

Old C++ standards only require it to have an average $O(n \log n)$ time complexity.

C++11 or newer standards require it to have a worst case $O(n \log n)$ time complexity.

And it’s probably much faster than the quick sort you write.

How does it do it?

std::sort

Well, C++ standard doesn’t specify its implementation, so it depends on your compiler!

It is often implemented as a variant of introspective sort, a combination of insertion sort, quick sort and heap sort.

(1) When the data size is large, use quick sort.
(2) When the data size becomes small, change to insertion sort. Since it is fast with small cases.
(3) When the quick sort unfortunately meet the $O(n^2)$ worst cases, changing to heap sort to guarantee the $O(n \log n)$ worst case time complexity.

std::nth_element

By the way, the algorithm we introduced to find the k-th element is also provided.

But its called “std::nth_element”.

Header file: #include <algorithm>

Usage: std::nth_element(first, nth, last) or std::nth_element(first, nth, last, cmp)

It “partially sort” the element in [first, last), make sure the (*nth) is the correct element. And all element in [first, nth) is smaller than (*nth), all elements in (nth, last) is greater than (*nth).

Q&A