COMP 3711 Fall 2021

Design & Analysis of Algorithms

Written By: Ljm

Topic 1 **Asymptotic**

1 Algorithm

An **algorithm** is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions. To evaluate an algorithm, we measure:

- **Memory (Space Complexity)**: all space used except for holding inputs
- **Running time (Time Complexity)**

In this course, we measure algorithms *analytically*, i.e., depends only on the algorithms, without considering actual implementations, hardwares, etc.

However, it is difficult and rarely that we can say "one algo is better than the other", since that usually depends on input size, input data(even for same size), etc.

2 Time Complexity

Usually, we measure running time(time complexity) as the num of machine instructions, such as addition, multiplication, swap(as used in sorting analysis)... We describe running time as a function of input size: $T(n)$.

There are three commonly-used asymptotic notations:

- **Upper bounds:** $T(n) = O(f(n))$, if $\exists c > 0, n_0 \geq 0$ such that $\forall n \geq n_0, T(n) \leq c \cdot f(n)$.
- **Lower bounds:** $T(n) = \Omega(f(n))$, if $f(n) = O(T(n))$.
- **Tight bounds:** $T(n) = \Theta(f(n))$, if both $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

Here are some notes for above notations: First, more accurate expression should be $T(n) \in O(f(n))$, but we often use $=$ for simplicity, which means "is", not "equal". Second, these notations is not properly definable using limits. One may think that $f(n) = O(g(n))$ is equivalent to $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$, but a counterexample can easily be found like $f(n) = (2 + (-1)^n)g(n)$, in which case the limit does not exist.

I will omit examples here, but I'd like to list some interesting facts. (1) 2^{10n} is not $O(2^n)$, since it is $(2^n)^{10}$. (2) $\Theta(f(n) + g(n)) = \Theta(\max(f(n), g(n)))$. (3) $\sum_{i=1}^{n} \frac{1}{i} = O(\log n)$, which is called *Harmonic Series*. (4) $\log(n!) = \Theta(n \log n)$.

For a certain algorithm, different inputs can cause different performances, even with same input size. For insertion sort, input an already sorted list requires no additional swaps, which gives $\Theta(n)$, and this is called **best case**; input an inversely sorted list gives $T(n) = \sum_{i=2}^{n}(i-1) = \Theta(n^2)$, this is **worst case**; if we average over all possible inputs for a certain size n, assuming same probability distribution on these inputs, then the result running time is called **average case**.

Generally, average case analysis is rather complicated. In insertion sort, we assume each of the $n!$ permutations is distributed equally likely. With some probability knowledge we will know it's $\Theta(n^2)$. I will give brief proof in last page of this note if you are interested.

Let's have a summary of three kinds of analysis: (1) best case is ideal so that it is useless; (2) average case is sometimes used but requires complicated analysis; (3) **worst case** is commonly used, since it gives running time guarantee **independent of actual input**. In this course, **Worst-case analysis is the default**, but it is not perfect: some algorithms with bad worst-case running time actually work very well in practice, since worst case input rarely occurs.

When we say an algorithm's worst case running time is $O(f(n))$, we mean **on all inputs of size** n, the algorithm's running time is $O(f(n))$, but there is no need to really *find* the worst input to prove.

When we say an algorithm's worst case running time is $\Omega(f(n))$, we mean **there exists at least one input of size** n, the algorithm's running time is $\geq c \cdot f(n)$. We mainly use this to prove the big-Oh analysis is tight.

To understand above two paragraphs, again consider insertion sort: it runs in $\leq \frac{n(n-1)}{2}$ time for all inputs of size n, so it is $O(n^2)$, it **requires** $\frac{n(n-1)}{2}$ time if items are reversed, so it is $\Omega(n^2)$. To combine, it runs in $\Theta(n^2)$ time.

This is the end of class note. Last modified: Sep 9
Sep 9: Fixed some typos.

Brief proof for average case time complexity of insertion sort:

Firstly, one can show that the number of "swaps" is equals to the number of **inversions**.(proof by induction in lecture slide divide & conquer)

So now we know the running time for a certain input will be $\Theta(n + I)$, where I is the number of inversions of the original array.

Here, we define X_{ij} to be 1 if $a[i]$ and $a[j]$ form an inversion and 0 otherwise. So an given input of size n will have $n(n-1)/2$ different X_{ij}s.

Now, we can express I as: $I = \sum X_{ij}$. But remember we are interested in the **expected number of inversions** in the array, since we're looking for average running time of all inputs. This is also simple by linearity of expectation: $E(I) = E(\sum X_{ij}) = \sum E(X_{ij})$.

That's a good one, $E(X_{ij})$ is the expected value of X_{ij}, of course it is $1 \cdot P(X_{ij} = 1) = 0.5$, since we have assumed $n!$ permutations are equally likely.

Thus, $E(I) = \sum(1/2)$, and there are $n(n-1)/2$ terms, which gives $E(I) = n(n-1)/4 = \Theta(n^2)$.

To sum up, on expectation the runtime will be $\Theta(n^2 + n) = \Theta(n^2)$, This explains why the average-case behavior of insertion sort is $\Theta(n^2)$.