DSC40B: Theoretical Foundations of Data Science II

Lecture 3: *More on asymptotic time complexity; Best and Worst case*

Instructor: Yusu Wang

Today

- More on asymptotic complexity
  - Properties, and some cautioning

- Asymptotic time complexity of algorithms
  - Best time? Worst time? Expected time?

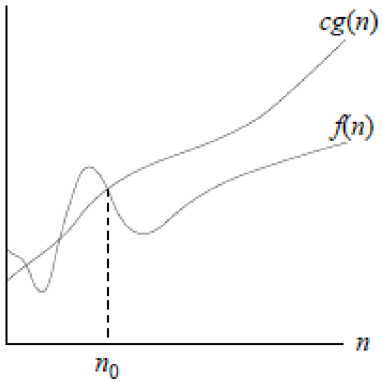
More about Asymptotic complexity

Previously,

Big-O (upper bounded)

We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that for all $n \geq n_0$:

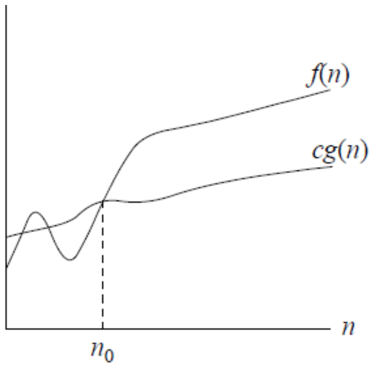
$$f(n) \leq c \cdot g(n)$$



Big-Ω (lower bounded)

We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that for all $n \geq n_0$:

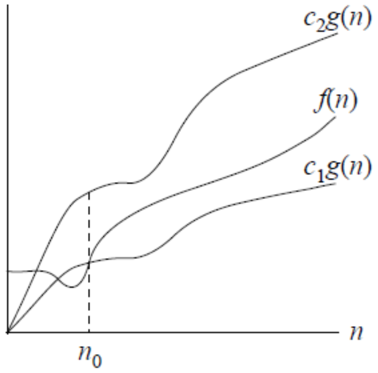
$$f(n) \geq c \cdot g(n)$$



Big-Θ (asymptoticly the same)

We write $f(n) = \Theta(g(n))$ if there are positive constants n_0, c_1, and c_2 such that for all $n \geq n_0$:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$



Another view

- Assume that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists.

$f(n) = O(g(n))$ if there exists $c > 0$ such that:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c.$$

$f(n) = \Omega(g(n))$ if there exists $c > 0$ such that:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c.$$

$f(n) = \Theta(g(n))$ if there exists $c_1, c_2 > 0$ such that:

$$c_1 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c_2.$$

Useful special cases

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then

$$f(n) \in O(g(n)) \text{ but } f(n) \notin \Theta(g(n)).$$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then

$$f(n) \in \Omega(g(n)) \text{ but } f(n) \notin \Theta(g(n)).$$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$ $(c \neq \infty)$, then

$$f(n) \in \Theta(g(n)).$$

Hierarchy

- $\Theta(n^n)$
- $\Theta(3^n)$
- $\Theta(2^n)$
- $\Theta(n^3)$
- $\Theta(n^2)$
- $\Theta(n\log(n))$
- $\Theta(n)$
- $\Theta(n^{0.5})$
- $\Theta(n^{0.1})$
- $\Theta((\log(n))^2)$
- $\Theta(\log(n))$
- $\Theta(1)$

Higher complexities are asymptotic upper bound for lower ones, and there is no big-Θ relation between any two of them.

Complexity decreasing

Some more examples

- $n\sqrt{n} = O(n^2)$?
- $\lg n = O(n)$?
- $\lg n = O(\sqrt{n})$?
- $n \lg n = O(n^{1.5})$?
- $3n^2 - n\sqrt{n} + n \lg n = \Theta(\_\_\_\_)$?
- $10^{10} n = O(n^2)$?

Some useful relations

- For any two constant $a, b > 1$
  - $\log_a n = \Theta(\log_b n) = \Theta(\lg n)$
- $1 + 2 + 3 + \cdots + \ n = \sum_{i=1}^{n} i = \Theta(n^2)$ (Arithmetic sum)
- $1 + 2^2 + 3^2 + \cdots + n^2 = \sum_{i=1}^{n} i^2 = \Theta(n^3)$
- $1 + 2^d + 3^d + \cdots + n^d = \sum_{i=1}^{n} i^d = \Theta\left(n^{d+1}\right)$
- $\lg 1 + \lg 2 + \cdots + \lg n = \lg n! = \Theta(n \lg n)$
- $1 + \frac{1}{2} + \left(\frac{1}{2}\right)^2 + \cdots + \left(\frac{1}{2}\right)^m = \Theta(1)$ (Geometric sum)
- For any $0 < r < 1$, $\ 1 + r + r^2 + \cdots r^m = \frac{1 - r^{m+1}}{1 - r} = \Theta(1)$
- For any $r > 1$, $\ 1 + r + r^2 + \cdots r^m = \frac{r^{m+1} - 1}{r - 1} = \Theta(r^m)$

Properties

- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$, and $f(n) = \Omega\big(g(n)\big)$.

- (Transpose) symmetry:
  - If $f(n) = \Theta(g(n))$, then $g(n) = \Theta\big(f(n)\big)$
  - If $f(n) = O(g(n))$, then $g(n) = \Omega\big(f(n)\big)$. The converse also holds.
    - E.g, $n = O(n \lg n) \Rightarrow n \lg n = \Omega(n)$

- Transitivity:
  - If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O\big(h(n)\big)$.
  - Same for Ω and Θ
  - E.g, $\lg n = O(n)$; $n = O(2^n) \Rightarrow \lg n = O(2^n)$

- Prove the statement that

$$\textit{If } f(n) = \Theta(g(n)), \textit{ then } g(n) = \Theta\big(f(n)\big).$$

- Proof:

Since $f(n) = \Theta\big(g(n)\big)$, by definition, we know that there exist two positive constants c_1 and c_2, as well as integer $n_0 > 0$, s.t.

$$\forall n > n_0, \qquad c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$$

By LHS of the above inequality, we have that $g(n) \le \frac{1}{c_1} f(n)$

By RHS of the above inequality, we have that $g(n) \ge \frac{1}{c_2} f(n)$

Putting these two together, we have that there exist positive constant $b_1 = \frac{1}{c_1}$ and $b_2 = \frac{1}{c_2}$, and integer $n_0 > 0$, s.t.

$$\forall n > n_0, \qquad b_2 \cdot f(n) \le g(n) \le b_1 \cdot f(n)$$

Hence by definition of big-Θ notation, it follows that $g(n) = \Theta\big(f(n)\big)$.

Properties

- Assume all functions we consider are positive functions
- $f(n) + g(n) = \Theta(\max(f(n), g(n)))$
- $f(n) + O(f(n)) = \Theta(f(n))$
- If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then
  - $f_1(n) + f_2(n) = \Theta(g_1(n) + g_2(n)) = \Theta(\max(g_1(n), g_2(n)))$
- Useful in analyzing algorithm with multiple commands.

```
def foo(n):
    bar(n)
    baz(n)
```

- $T_{foo}(n) = T_{bar}(n) + T_{baz}(n)$
- If $T_{bar} = \Theta(n^2)$ and $T_{baz}(n) = \Theta(n^3)$...
- ...then $T_{foo}(n) = \Theta(n^3)$.
- baz is the **bottleneck**.

Properties

- Assume all functions we consider are positive functions
- $f(n) + g(n) = \Theta(\max(f(n), g(n)))$
- $f(n) + O(f(n)) = \Theta(f(n))$
- If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then
  - $f_1(n) + f_2(n) = \Theta(g_1(n) + g_2(n)) = \Theta(\max(g_1(n), g_2(n)))$
- If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then
  - $f_1(n) \times f_2(n) = \Theta(g_1(n) \times g_2(n))$

Example

```
def foo(n):
    for i in range(4*n + 4, 4*n**2 + 5):
        for j in range(100*n, n**2):
            print(i, j)
```

Remark 1:

- In this course, we mostly use asymptotic language to measure time complexity of algorithms
- However, it can be used for other places where a measurement of growth rate is needed.
  - Ex1: $\lg 1 + \lg 2 + \cdots + \lg n = \lg n! = \Theta(n \lg n)$
  - Ex2: CLT says that the sample mean has a normal distribution with standard deviation $\sigma/\sqrt{n}$, we often say that the error in sample mean is $O(1/\sqrt{n})$ with high probability.

Caution 1:

- It is convenient to think that
  - Big-O is smaller than or equal to
  - Big-Ω is larger than or equal to
  - Big-Θ is equal
- However,
  - These relations are modulo constant factor scaling
  - Not every pair of functions have such relations

If $f(n) \notin O(g(n))$, this does not imply that $f(n) = \Omega(g(n))$.

Caution 2

- Provide a unified language to measure the performance of algorithms
  - Give us intuitive idea how fast we shall expect the alg.
  - Can now compare various algorithms for the same problem

- Constants hidden!
  - $O(n)$ $vs.$ $n \lg n$
    - Say, $T_1(n) = 10^6 n$, while $T_2(n) = n \backslash\lg n$,which one would you use in practice?

Caution 3

- Don't include constants, lower-order terms in the notation.
  - Bad: $3n^2 + 2n + 5 = \Theta(3n^2)$
  - Good: $3n^2 + 2n + 5 = \Theta(n^2)$
  - It isn't wrong to do so, just defeats the purpose.
- Don't misinterpret meaning of $\Theta(\cdot)$.
  - $f(n) = \Theta(n^2)$ does not mean that there are constants so that $f(n) = c_1 n^2 + c_2 n + c_3$.
  - E.g, $3\,n^2 - n\sqrt{n} + n \lg n = \Theta(n^2)$

Caution 4

- $O(n) + O(n) = O(n)$

OK

- $T(n) = n + \sum_i^k O(n)$
  $= n + O(n)$

?

k should not depend on n !
It has to be a constant …

Best time complexity, worst time complexity ?

- Now that we are equipped with a language to describe "time complexity", let's use it to analyze algorithms.

- Simple example 1:

```
def mean(arr):
        total = 0
        for x in arr:
                total += x
        return total / len(arr)
```

- No matter what the input is, let n be its size, then we have
  - $T(n) = \Theta(n)$

Simple Example 2

- Search queries
  - say in a database A represented by an array storing n keys (numbers), given a key k, check whether $k \in A$ or not.
    - return the index of this element in A if it is found, None otherwise.

```
def linear_search(A, k):
        for i, x in enumerate(A):
                if x == k:
                        return i
        return None
```

- Running time depends on specific input!
- Best scenario?
  - $\Theta(1)$
- Worst scenario?
  - $\Theta(n)$

- Best-case time complexity
  - How does the time taken in the best case grow as the input gets larger?
  - $T_{best}(n)$: Best time of the algorithm over any input of size n
    - Note: it has to be of any possible input size n. One cannot say the best case is $O(1)$ as we can have an input of size 1. It is about certain structure of input (of any size) that makes the algorithm faster.
  - The asymptotic growth of $T_{best}(n)$ is the algorithm's best-case time complexity
    - E.g, for linear search algorithm: $T_{best}(n) = \Theta(1)$
- Worst-case time complexity
  - How does the time taken in the worst case grow as the input gets larger?
  - $T_{worst}(n)$: Worst time of the algorithm over any input of size n
  - The asymptotic growth of $T_{worst}(n)$ is the algorithm's worst-case time complexity
    - E.g, for linear search algorithm: $T_{worst}(n) = \Theta(n)$

- Often in practice (and in this class)
  - We focus on worst-case time complexity
    - So that we have confidence that even in the worst scenario, the time complexity will still be bounded by a certain value.
- In data analysis
  - Average (expected) case is also important (see next time)

- However, one should be mindful of the existence of the best-case time complexity, and understand that the running time can really depend on the specific input.

Another example

- The Movie problem
  - Input: Given a list of length of movies available, stored in array $movies$, and a flight duration D
  - Output: Return two movies whose total length = D; None otherwise.

A simple approach

```
def find_movies(movies, D):
    n = len(movies)
    for i in range(n):
        for j in range(i + 1, n):
            if movies[i] + movies[j] == D:
                return (i, j)
    return None
```

- Best-case time complexity
  - $T_{best}(n) = \Theta(1)$
- Worst-case time complexity
  - $T_{worst}(n) = \Theta(n^2)$

Exercise:

Can you find an algorithm with better worst-case time complexity ?

Remark

- It is not true that best-case time complexity is always $\Theta(1)$
  - e.g, the mean algorithm has best-case time complexity $\Theta(n)$ as well
- The best-case time complexity analysis is about the structure of input
  - certain structure may lead to faster execution of algorithm
- Knowing both best- and worst- case time complexity can give a more thorough understanding of algorithm performance
- However, note that it is possible that both cases can be biased by only some specific infrequent input

- Next time
  - We will also talk about the average and expected running time
  - However, we again emphasize that in practice, the worst-case time analysis is the most common one, and also the one that we will use later in the course.

FIN