DSC40B:
Theoretical Foundations of Data Science II

Lecture 2: *Nested loops and asymptotic time complexity*

Instructor: Yusu Wang

Today's Agenda

- Warm up: analyzing nested loops
- What is Θ notation ?

More examples: Nest loops

The median problem

- Design an algorithm for the following problem
  - Input: given n numbers $\mathrm{X} = \{x_1, x_2, \dots, x_n\}$
  - Output: find h minimizing the total absolute loss:
    - $Loss(h) = \sum_{i=1}^{n} |x_i - h|$

- Recall from DSC40A
  - Solution: h^* is a median of numbers in X
    - i.e, the number whose order in X (if sorted) is $\lfloor \frac{n}{2} \rfloor$ or $\lceil \frac{n}{2} \rceil$

- Question:
  - How to compute this median?

One algorithm

- Idea:
  - h^* has to be one of $\mathrm{X} = \{x_1, x_2, \ldots, x_n\}$
  - compute all $Loss(x_1), \ldots, Loss(x_n)$
  - return the one whose loss is smallest

One algorithm

```
def median(X):
        min_h = None
        min_value = float('inf')
        for h in X:
            total_abs_loss = 0
            for x in X :
                total_abs_loss += abs(x - h)
            if total_abs_loss < min_value:
                min_value = total_abs_loss
                min_h = h
        return min_h
```

Inner-for loop:

Outer-for loop:

Total:

One algorithm

```
def median(X):
    min_h = None
    min_value = float('inf')
    for h in X:
        total_abs_loss = 0
        for x in X :
            total_abs_loss += abs(x - h)
        if total_abs_loss < min_value:
            min_value = total_abs_loss
            min_h = h
    return min_h
```

$T(n) = \Theta(n^2)$

We will see how to do better later in class.

- More abstract form

```
def foo_0(n):
    for x in range(n):
        for y in range(n):
            print(x + y)
```

Inner-most line will be called n^2 times.
Thus the time complexity is
$T_0(n) = n^2$

Caution!

- Not all nested loop takes $\Theta(n^2)$ time

```
def foo_1(n):
    for x in range(n):
        for y in range(n, n+10):
            print(x + y)
```

$T_1(n) = \Theta(n)$

```
def foo_2(n):
    for x in range(n):
        for y in range(n, 2n-10):
            print(x + y)
```

$T_2(n) = \Theta(n^2)$

A second example

- Alex is given n sticks. He needs to design an algorithm to compute the tallest pole he can make by stacking two sticks

```
def tallest_pole(heights):
1.   max_height = -float ('inf')
2.    n = len(heights)
3.    for i in range(n):
4.        for j in range(i+1, n):
5.            h = heights[i] + heights[j]
6.            if h > max_height
7.                max_height = h
8.    return max_height
```

On outer iter.#1, inner body runs _____ times

On outer iter.#2, inner body runs _____ times

On outer iter.#3, inner body runs _____ times

⋮

On outer iter.#n, inner body runs _____ times

For the k-th iteration of outer loop, the inner loop takes $c(n-k)$ time.

- How many times does line $5-7$ (inner body) run?

Tallest_pole, cont.

- Total times line 5-7 (inner body) is executed:

$$\underbrace{(n-1)}_{\text{1st outer iter}} + \underbrace{(n-2)}_{\text{2nd outer iter}} + \ldots + \underbrace{(n-k)}_{\text{kth outer iter}} + \underbrace{(n-(n-1)}_{\text{(n-1)th outer iter}} + \underbrace{(n-n)}_{\text{nth outer iter}}$$

$$=$$

$$1+2+3+\ldots+(n-3)+(n-2)+(n-1)$$

$$=$$

- Recall an arithmetic sum:
  - $1+2+3+\cdots+m = \frac{m(m+1)}{2}$

- Algorithm tallest_pole has $T(n) = \Theta(n^2)$
- Another way to see the time complexity:
  - Number of pairs of n objects is $\binom{n}{2} = \frac{\mathrm{n(n-1)}}{2} = \Theta(n^2)$

Exercise: Find a linear-time algorithm for this problem.

Note

- Alex is given n sticks. He needs to design an algorithm to compute the tallest pole he can make by stacking two sticks

```
def tallest_pole(heights):
    max_height = -float (‘inf’)
    n = len(heights)
    for i in range(n):
        for j in range(i+1, n):
            h = heights[i] + heights[j]
            if h > max_height
                max_height = h
    return max_height
```

- Essentially the same as the following

```
def foo_3(n):
    for x in range(n):
        for y in range(i+1, n):
            do sth. in constant time
```

What's the difference?

- The number of iterations of the inner loop depends on the outer loop

```
def foo_0(n):
    for x in range(n):
        for y in range(n):
            print(x + y)
```

```
def foo_3(n):
    for x in range(n):
        for y in range(i+1, n):
            print(x + y)
```

- Inner loop doesn't depend on outer loop iteration #.
- Just multiply: inner body executed n^2 times

- Inner loop depends on outer loop iteration #.
- Cannot just multiply: need to figure out for each outer loop iteration #, how many times inner loop will run.

Depended Nested Loop

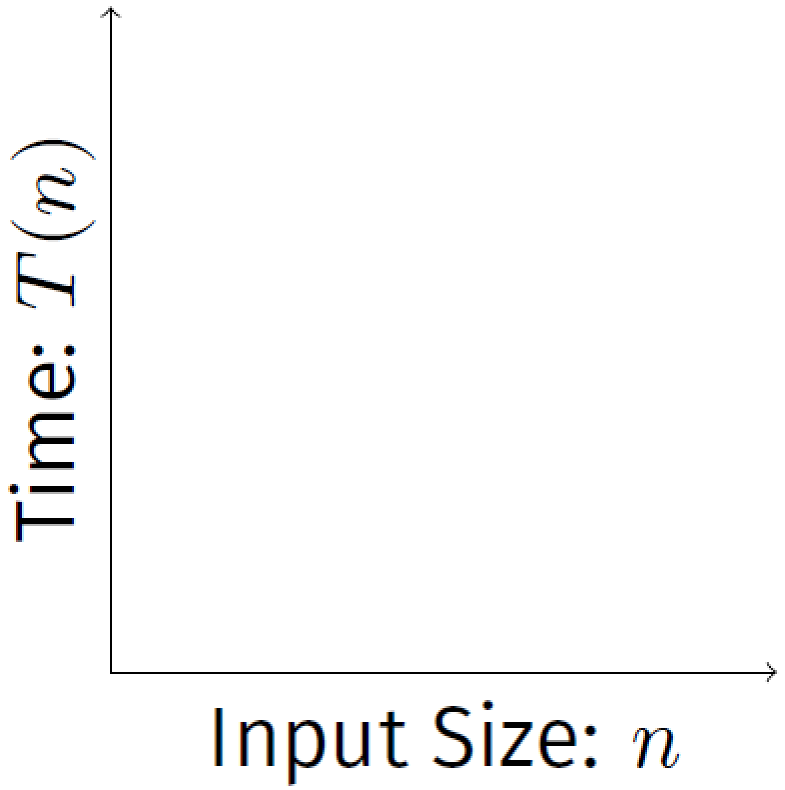
```
def foo_3(n):
    for x in range(n):
        for y in range(i+1, n):
            print(x + y)
```

- Step 1: Find formula $f(k)$ for "number of iterations of inner loop during outer iteration k
- Step 2: Then sum up the total cost as $\sum_{k=1}^{n} f(k)$
  - For this example, $f(k) = n - k$. Thus total cost is $\sum_{k=1}^{n} f(k) = \sum_{k=1}^{n}(n-k) = \frac{\mathrm{n(n-1)}}{2} = \Theta(n^2)$

- What if there are more two layers of nested loops?
  - Always calculate the cost inside-out ! First figure out the cost of inner-most loop
- What if the loops are not just for–loops?
  - We will see examples of while loops soon

Linear vs quadratic growth

Scaling



- $T(n) = \Theta(n)$
  - means "$T(n)$ grows like n"
  - linear growth
- $T(n) = \Theta(n^2)$
  - means "$T(n)$ grows like n^2"
  - quadratic growth

- Suppose your algorithm runs 5 sec on 1000 points
- Linear growth
  - If the input has 100,000 points, then it takes 500 seconds (8.3 min)
- Quadratic growth
  - If the input has 100,000 points, then it will take 50,000 seconds (~14 hours)

Some common growth rates

- $\Theta(1)$: constant
- $\Theta(\log n)$: logarithmic
- $\Theta(n)$: linear
- $\Theta(n \log n)$: linearithmic
- $\Theta(n^2)$: quadratic
- $\Theta(n^3)$: cubic
- $\Theta(2^n)$: exponential

More on nested loops analysis

A first example

```
function func(n)
x ← 0;
i ← 0;
while (i ≤ n) do
    x ← x + i;
    i ← i + 3;
end
return (x);
```

#iteration	value of i	Cost of this iteration

A pseudo-code example

```
function func(n)
x ← 0;
i ← 0;
while (i ≤ n) do
    x ← x + i;
    i ← i + 3;
end
return (x);
```

- Each iteration of the **while** loop takes c time for some constant c
- In the k-th iteration of the **while** loop, the value of i is $i = 3(k-1)$
- The **while** loop terminates when $i > n$, meaning that

$$3(k-1) > n \Rightarrow k > \frac{n}{3} + 1$$

- Thus, the **while** loop runs $\frac{n}{3} + 1$ iterations.
- Hence the total time complexity of the while loop is #iterations $\times$ c. The time complexity of the algorithm is

$$T(n) = c\left(\frac{n}{3} + 1\right) + \Theta(1) = \Theta(n)$$

A second example

```
function func(n)
x ← 0;
i ← 1;
while (i ≤ n) do
    x ← x + i;
    i ← i * 3;
end
return (x);
```

#iteration	value of i	Cost of this iteration

A second example

```
function func(n)
x ← 0;
i ← 1;
while (i ≤ n) do
    x ← x + i;
    i ← i * 3;
end
return (x);
```

- Each iteration of the **while** loop takes c time for some constant c
- In the k-th iteration of the **while** loop, the value of i is $i = 3^{k-1}$
- The **while** loop terminates when $i > n$, meaning that
$$3^{(k-1)} > n \Rightarrow k > \log_3 n + 1$$
- Thus, the **while** loop runs $\log_3 n + 1$ iterations.
- Hence the total time complexity of the while loop is #iterations $\times\, c$. The time complexity of the algorithm is
$$T(n) = \Theta(\log_3 n) = \Theta(\lg n)$$

A third example

```
function func(n)
x ← 0;
for i ← 1 to n do
    j ← 1;
    while (j ≤ n) do
        x ← x + (i − j);
        j ← 2 * j;
    end
end
return (x);
```

- By the same analysis as the previous slide, the inner **while** loop takes $c \lg n$ time for some constant c for each iteration of outer-for loop
- The outer-for loop has n iterations
- Hence the total time complexity of the algorithm is

$$T(n) = n \times (c \lg n) = \Theta(n \lg n)$$

A fourth example

```
function func(n)
x ← 0;
for i ← 1 to n do
    j ← 1;
    while (j ≤ i ) do
        x ← x + (i − j);
        j ← 2 ∗ j;
    end
end
return (x);
```

- By the same analysis as the previous slide, for the i-th iteration of the outer-for loop, the inner **while** loop takes $c\lg i$ time for some constant c

- For the outer-for loop , i changes from 1 to n. Hence the total time complexity is

$$T(n) = \sum_{i}^{n}(c\lg i)$$

$$= c(\lg 1 + \lg 2 + \lg 3 + \cdots \lg n) = c\lg(n!) = \Theta(n\lg n)$$

- We have seen $\Theta(\cdot)$ allows us to ignore unnecessary details and focus on dominating terms of growth
  - Informally, $\Theta(\cdot)$ forgets constant factors, lower-order terms.
    - $5n^3 + 2n - 42 = \Theta(n^3)$
- Simpler, easier to analyze, and focus on key growth rate
- But: what exactly are we ignoring?

- Before we introduce this formally
  - First introduce big-O and big-Ω notations
  - Then big-Θ
  - Intuitively, big-O, big-Ω, and big-Θ "roughly" corresponds to $\leq$, $\geq$, and $=$, respectively (up-to constants)

Asymptotic notations

Big-O notation

Big-O notation

Definition

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

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

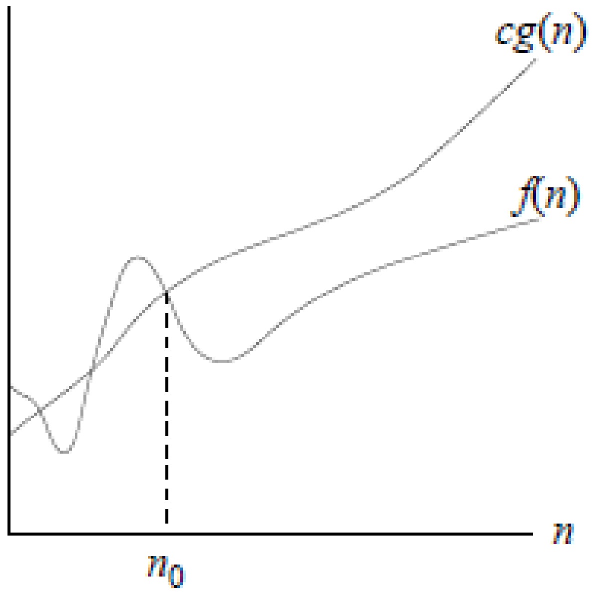
- More precisely, should be $f(n) \in O\big(g(n)\big)$
- Intuitively, $f(n) = O\big(g(n)\big)$ means that
  - $f(n)$ grows at most as fast as g(n) (up to multiplicative constant factor)
  - We also say $g(n)$ is an asymptotic upper bound for $f(n)$ in this case

Big-O notation

Definition

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

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



Examples

- $n^3 - 3n^2 + 5n - 1 = O(n^3)$
- Proof:

- $5n^2 + 6\sqrt{n} + 8 = O(n^2)$?
- Proof:

More examples

- $\sqrt{6n^3 + 7n^2 + 3n} = O(n^2)$?
- $\sqrt{6n^3 + 7n^2 + 3n} = O(n^{1.5})$?
- $\frac{n}{\lg n} = O(n)$?
- $\frac{100\, n}{\lg n} = O(\frac{n}{\lg n})$?
- $n \lg n = O(n)$?
- $\log_2 n = O(\log_{10} n)$?

A useful result

Lemma [Upper Bound]

If $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists, then $f(n) = O\big(g(n)\big)$ if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c \text{, where } c \text{ is a positive constant.}$$

Corollary [Upper Bound]

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

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$, then $f(n) = O\big(g(n)\big)$ does not hold.

More examples

- $n^{100} = O(n^2)$?
- $50 \lg n = O(n)$?
- $n^{100} = O(2^n)$?
- $2^n = O(3^n)$?

Big-Ω notation

Big-Ω notation

Definition

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

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

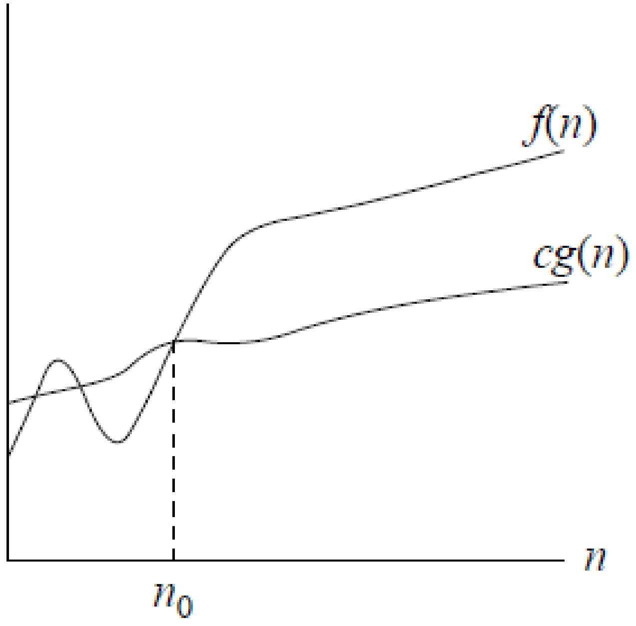
- More precisely, should be $f(n) \in \Omega(g(n))$
- Intuitively, $f(n) = \Omega(g(n))$ means that
  - $f(n)$ grows at least as fast as $g(n)$ (up to multiplicative constant factor)
  - We also say $g(n)$ is an asymptotic lower bound for $f(n)$ in this case

Big-Ω notation

Definition

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

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



Examples

- $n^3 - 3n^2 + 5n \ = \Omega(n^3)$?
- Proof:

A useful result

Lemma [Lower Bound]

If $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists, then $f(n) = \Omega(g(n))$ if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$$, where c is a positive constant.

Corollary [Lower Bound]

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

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0,$ then $f(n) = \Omega(g(n))$ cannot hold.

More Examples

- $5n^2 + 6n + 8 = \Omega(n^3)$?
- $5n^2 + 6n + 8 = \Omega(n^2)$?
- $n^2 = \Omega(100\, n^2)$?
- $\sqrt{6n^3 - 7n^2 + 3n} = \Omega(n^{1.5})$?
- $2^n = \Omega(n^2)$?
- $3\lg n = \Omega(n)$?
- $3^n = \Omega(2^n)$?
- $2^n = \Omega(3^n)$?
- $\log_{10} n = \Omega(\log_2 n)$?

Big-Θ notation

Big-Θ notation

Definition

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

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

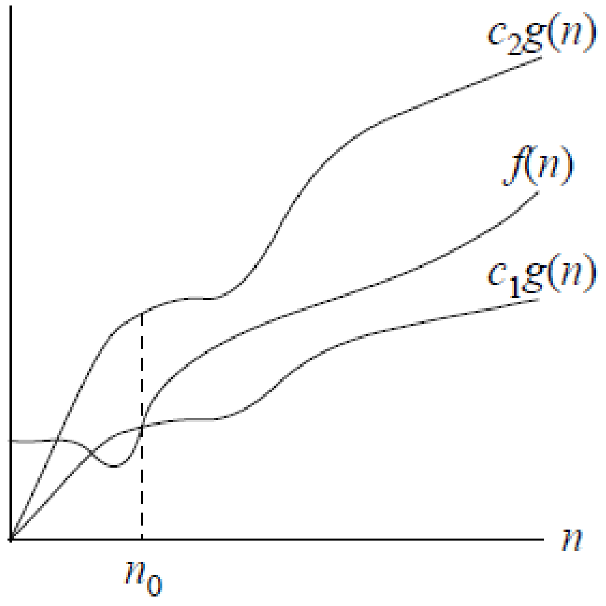
- More precisely, should be $f(n) \in \Theta(g(n))$
- Intuitively, $f(n) = \Theta(g(n))$ means that
  - $f(n)$ grows like $g(n)$ (up to multiplicative constant factor)

Big-Θ notation

Definition

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

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



A useful result

Lemma [Big-Theta]

If $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists, then $f(n) = \Theta\big(g(n)\big)$ if and only if

$c_1 \le \lim_{n\to\infty} \frac{f(n)}{g(n)} \le c_2$, where c_1 and c_2 are positive constants.

Corollary [Big-Theta]

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some positive constant c, then $f(n) = \Theta\big(g(n)\big)$.

Examples

- $2n^3 - 3n^2 = \Theta(n^3)$
- Proof:

More Examples

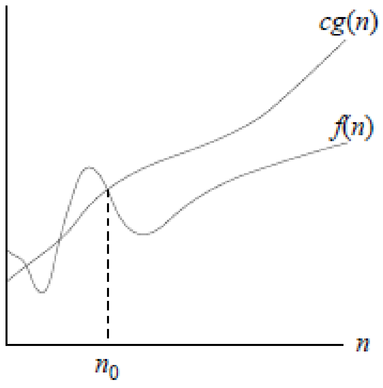
- $5n^2 + 6n + 8 = \Theta(n^2)$?
- $\sqrt{6n^3 - 7n^2 + 3n} = \Theta(n^{1.5})$?
- $n^2 - \lg n = \Theta(n^2)$?
- $3^n = \Theta(2^n)$?
- $\log_{10} n = \Theta(\log_2 n)$?

In summary

Big-O (upper bounded)

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

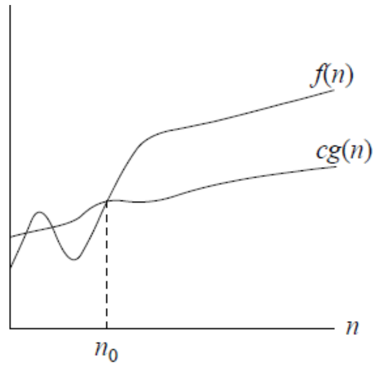
$$f(n) \leq c \cdot g(n)$$



Big-Ω (lower bounded)

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

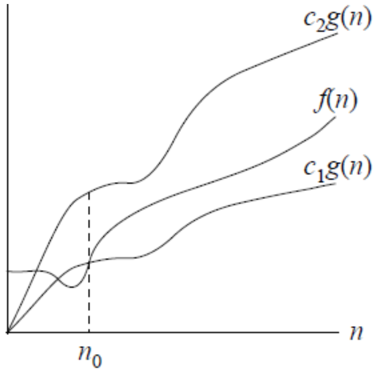
$$f(n) \geq c \cdot g(n)$$



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 $n \geq n_0$:

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



FIN