DTS203TC
Design and Analysis of Algorithms

Lecture 2: Growth of Functions

Dr. Qi Chen
School of AI and Advanced Computing

Time Complexity Analysis

- How fast is the algorithm?
  - Depend on the speed of the computer
  - Waste time coding and testing if the algorithm is slow

- How to measure efficiency?
  - Identify some important operations/steps and count how many times these operations/steps needed to executed
  - Number of operations usually expressed in terms of input size ***n***

Time Complexity Analysis

- Suppose:
  - an algorithm takes n^2 comparisons to sort n numbers
  - we need 1 sec to sort 5 numbers (25 comparisons)

- Now, if we can perform 2500 comparisons in 1 sec (100 times speedup), How many numbers we can sort?
  - 50 numbers (10 times more)

Time Complexity Analysis

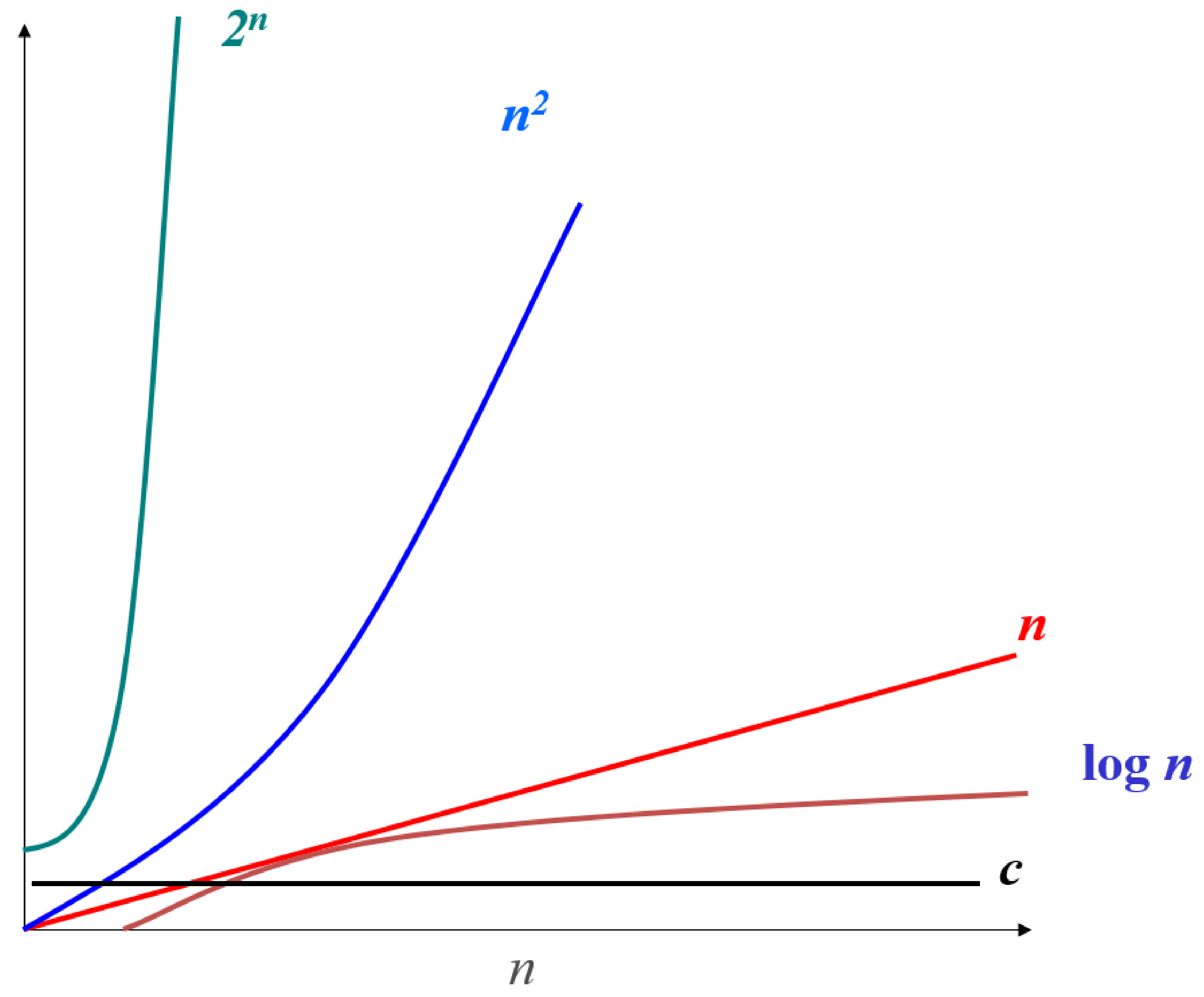
- The time complexity of Insertion Sort is: $O(n^2)$
  - If we doubled the input size, how much longer would the algorithm take?
    - Roughly 4 times
  - If we trebled the input size, how much longer would it take?
    - Roughly 9 times

Time complexity
– Big O notation

Which algorithm is the fastest?

- Consider a problem that can be solved by 5 algorithms A_1, A_2, A_3, A_4, A_5 using different number of operations.
  - $f_1(n) = \log n$ $\quad$ $(\log n \ \ stand\ for\ \log_2 n)$ $\quad$ $(\log_2 2^x = x)$
  - $f_2(n) = c$ $\quad$ (constant)
  - $f_3(n) = n^2$
  - $f_4(n) = n$
  - $f_5(n) = 2^n$

Relative growth rate



Growth of functions

n	$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3	2^n
2	1	1.4	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	2.8	8	24	64	512	256
16	4	4	16	64	256	4096	65536
32	5	5.7	32	160	1024	32768	4294967296
64	6	8	64	384	4096	262144	1.84×10^{19}
128	7	11.3	128	896	16384	2097152	3.40×10^{38}
256	8	16	256	2048	65536	16777216	1.16×10^{77}
512	9	22.6	512	4608	262144	134217728	1.34×10^{154}
1024	10	32	1024	10240	1048576	1073741824	

Hierarchy of functions

- We can define a hierarchy of functions each having a **greater** order of magnitude than its predecessor:

1	$\log n$	$\underbrace{n \;\; n^2 \; n^3 \; \dots \; n^k \; \dots}$	2^n
constant	logarithmic	polynomial	exponential

- As n increases, the values of the later functions increase more rapidly than the values of the earlier ones.

Hierarchy of functions

- When we have a function, we can assign the function to some function in the hierarchy:
  - For example, $\mathrm{f}(n) = an^2 + bn + c$

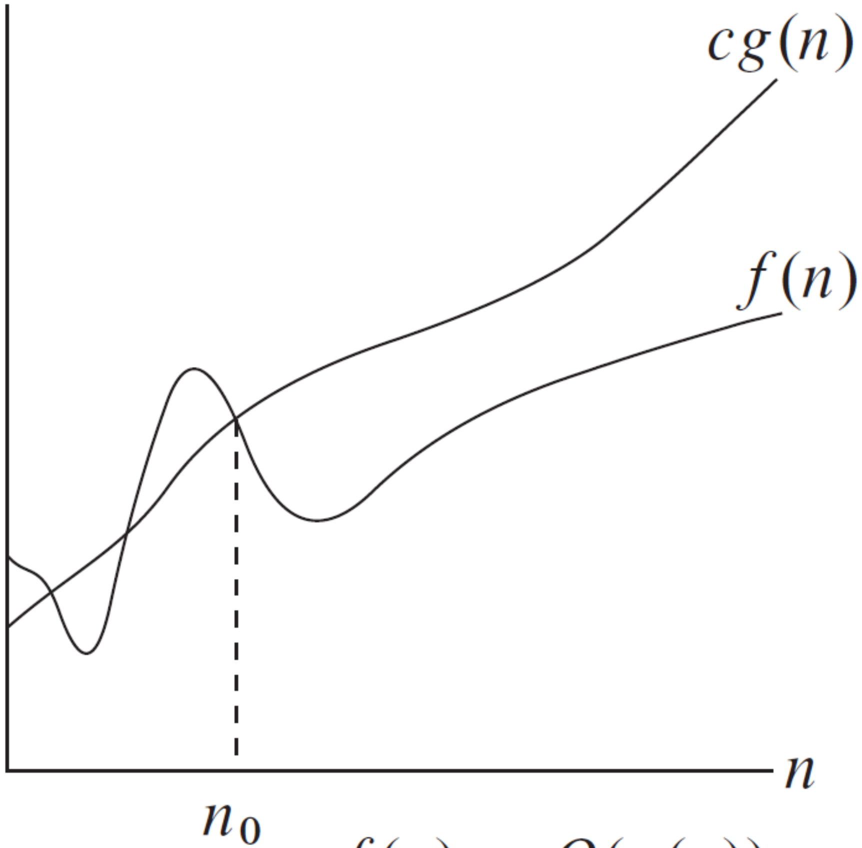
    The term with the highest power is an^2.
    The growth rate of f(n) is dominated by n^2.

- This concept is captured by Big-O notation

Big-O notation

- f(n) = O(g(n)): There exists a constant c and n_0 such that
  $$f(n) \le c \times g(n) \text{ for all } n \ge n_0$$

- O-notation provides an asymptotic upper bound on a function



$f(n) = O(g(n))$

Big-O notation

- Examples:
  - $2n^3 = O(n^3)$
  - $2n^3 + n^2 = O(n^3)$
  - $nlogn + n^2 = O(n^2)$

- function on L.H.S and function on R.H.S are said to have the same order of magnitude

Proof of order of magnitude

- Show that $2n^3 + n^2$ is $O(n^3)$
  - Since $n^2 < n^3$ for all n > 1,

    we have $2n^3 + n^2 \leq 2n^3 + n^3 = 3n^3$ for all n > 1.
  - Therefore, by definition $2n^3 + n^2$ is $O(n^3)$. (c = 3, n_0 =1)

- Show that $nlogn + n^2$is $O(n^2)$
  - Since $logn < n$ for all n > 1,

    we have$nlogn + n^2 \leq n^2 + n^2 = 2n^2$ for all n > 1.
  - Therefore, by definition$nlogn + n^2$is $O(n^2)$. (c = 2, n_0 =1)

Exercises

- Prove the order magnitude:
  - Show that $n^3 + 3n^2 + 3$ is $O(n^3)$
  - Show that $4n^2 \log n + n^3 + 5n^2 + n$ is $O(n^3)$

Exercises

- $n^3 + 3n^2 + 3$
  - $3n^2 \leq n^3 \quad \forall n \geq 3$
  - $3 \leq n^3 \quad \forall n \geq 2$
  - $\Longrightarrow n^3 + 3n^2 + 3 \leq 3n^3 \quad \forall n \geq 3$

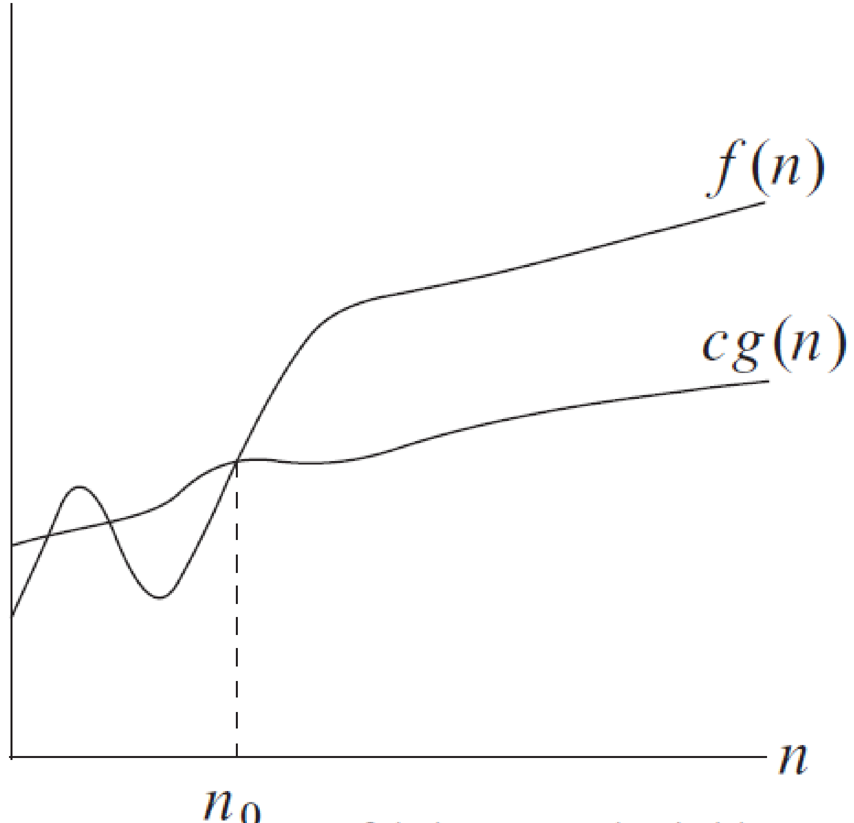
- $4n^2 \log n + n^3 + 5n^2 + n$
  - $4n^2 logn \leq 4n^3 \quad \forall n \geq 1$
  - $5n^2 \leq n^3 \quad \forall n \geq 5$
  - $\mathrm{n} \leq n^3 \quad \forall n \geq 1$
  - $\Longrightarrow 4n^2 logn + n^3 + 5n^2 + n \leq 7n^3 \quad \forall n \geq 5$

c and n_0 could be different when proving the order of magnitude

Ω-notation

- f(n) = Ω(g(n)): There exists a constant c and n_0 such that $c \times g(n) \leq \mathrm{f(n)}$ for all $n \geq n_0$

- Ω-notation provides an asymptotic lower bound.

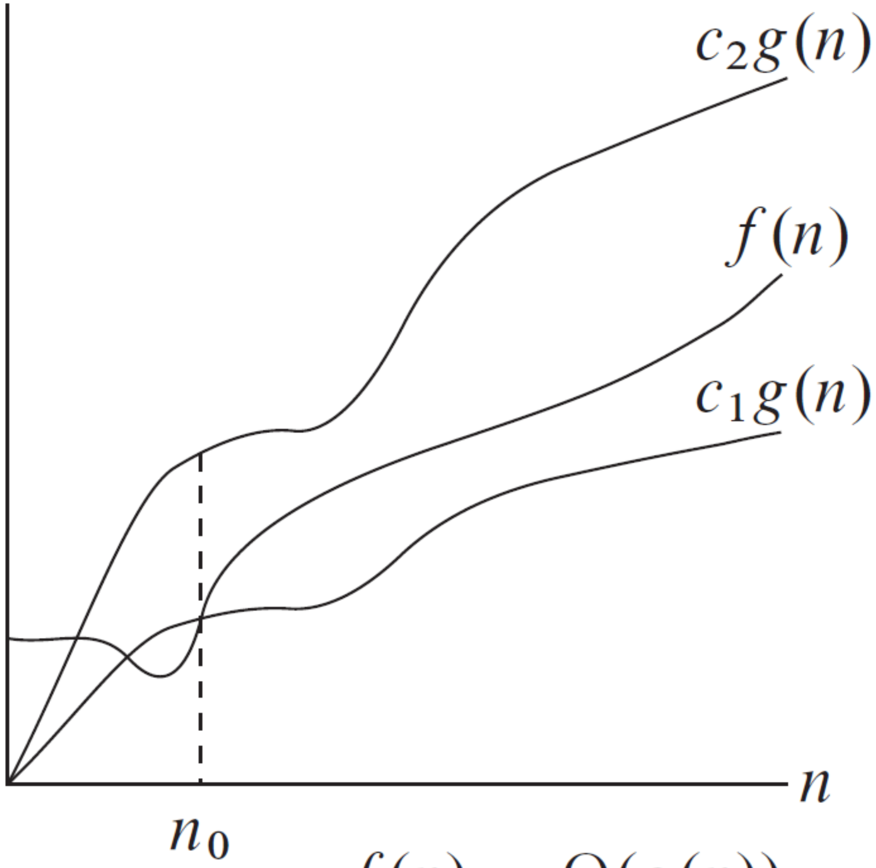


$f(n) = \Omega(g(n))$

Θ-notation

- $f(n) = \Theta(g(n))$: There exists constant c_1, c_2 and n_0 such that $c_1 \times g(n) \leq \mathrm{f(n)} \leq c_2 \times g(n)$ for all $n \geq n_0$

- Θ notation provides an asymptotically tight bound



$f(n) = \Theta(g(n))$

Asymptotic Notations

- Since O-notation describes an upper bound, we usually use it to bound the worst-case running time of an algorithm.
  - $O(n^2)$ bound on worst-case running time of insertion sort also applies to its running time on every input.
  - The $\Theta(n^2)$ bound on the worst-case running time of insertion sort, does not imply $\Theta(n^2)$ bound on the running time of insertion sort on every input. Best-case insertion sort runs in $\Theta(n)$ time.
  - $n = O(n^2)$, BUT O-notation informally describing asymptotically tight upper bounds

Exercises

- Write the computation complexity directly:
  - $n^3 + 3n^2 + 3$ $O(n^3)$
  - $4n^2 \log n + n^3 + 5n^2 + n$ $O(n^3)$
  - $2n^2 + n^2 \log n$ $O(n^2 \log n)$
  - $6n^2 + 2^n$ $O(2^n)$

Time complexity of this?

```
for (i=0;i<n;i++)
{
    stmt
}
```

O(?)

O(n)

Time complexity of this?

```
for (i=n;i>0;i--)
{
     stmt
}
```

O(?)

O(n)

Time complexity of this?

```
for (i=0;i<n;i=i+2)
{
     stmt
}
```

O(?)

O(n)

Time complexity of this?

```
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
    {
        stmt
    }
```

O(?)

$\mathbf{O}(n^2)$

Time complexity of this?

```
for (i=0;i<n;i++)
{
      stmt
}
for (j=0;j<n;j++)
{
      stmt
}
```

O(?)

O(n)

Xi'an Jiaotong-Liverpool University 西交利物浦大学

Time complexity of this?

```
for (i=0;i<n;i++)
    for (j=0;j<i;j++)
    {
        stmt
    }
```

O(?)

O(n^2)

Time complexity of this?

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

O(?)

$\mathbf{O}(\sqrt{n})$

Time complexity of this?

```
for (i=1;i<n;i=i*2)
{
    stmt
}
```

O(?)

O(logn)

Time complexity of this?

```
k=0
for (i=1;i<n;i=i*2)
{
      k++;
}
for (j=1;j<k;j=j*2)
{
      stmt
}
```

O(?)

O(logn)

Time complexity of this?

```
for (i=0;i<n;i++)
{
    for (j=1;j<n;j=j*2)
    {
        stmt
    }
}
```

O(?)

O(nlogn)

Some algorithms we learnt

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

O(?)

$$\mathbf{O}(n^2)$$

Some algorithms we learnt

```
for i = 1 to n-1:
    min = i
    for j = i+1 to n do
        if a[j] < a[min]
            min = j
    swap a[i] and a[min]
```

O(?)

$\mathbf{O}(n^2)$

Searching

- Input: n numbers $a_1, a_2, \ldots, a_n$ and a number X
- Output: determine if X is in the sequence or not

Xi'an Jiaotong-Liverpool University 西交利物浦大学

Sequential search

To find 7

- 12 34 2 9 7 5
  7
- 12 34 2 9 7 5
  7
- 12 34 2 9 7 5
  7
- 12 34 2 9 7 5
  7
- 12 34 2 9 7 5
  7 found!

Sequential search

To find 10

- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10 not found!

Sequential search

```
i = 1
found = false
while (i<=n && found==false)
{
        if X == a[i] then
                found = true
        else
                i = i+1
}
```

Best case: X is 1st no.
$\Rightarrow$ 1 comparison $\Rightarrow$ $O(1)$

Worst case: X is last OR X is not found $\Rightarrow$ n comparisons $\Rightarrow$ $O(n)$

How to improve Searching?

- Time complexity of Sequential searching is $O(n)$.

- If a sorted array is given, can we improve the time complexity?

Binary search

- Input: a sequence of n **sorted** numbers $a_1, a_2, \ldots, a_n$ in ascending order and a number X

- Idea of algorithm:
  - compare X with number in the middle
  - then focus on only the first half or the second half (depend on whether X is smaller or greater than the middle number)
  - reduce the amount of numbers to be searched by half

Binary Search

To find 24

3 7 11 12 **15** 19 24 33 41 55
24

19 24 **33** 41 55
24

19 24
24

24
24

found!

Binary Search

To find 30

3 7 11 12 **15** 19 24 33 41 55
30

19 24 **33** 41 55
30

19 24
30

24
30

not found!

Binary Search – Pseudo Code

```
first = 1, last = n, found = false
while (first <= last && found == false)
{
        mid = ⌊(first+last)/2⌋
        if (X == a[mid])
                found = true
        else
                if (X < a[mid])
                        last = mid-1
              else
                        first = mid+1
}
if (found == true)
        report "Found"
else
        report "Not Found"
```

Best case: X is the number in the middle $\Rightarrow$ 1 comparison $\Rightarrow$ O(1)

Worst case: at most (logn+1) comparisons $\Rightarrow$ O(logn)

Why?
Every comparison reduces the amount of numbers by at least half
E.g., 16 $\Rightarrow$ 8 $\Rightarrow$ 4 $\Rightarrow$ 2 $\Rightarrow$ 1

Learning outcomes

- Understand asymptotic complexity and notation
- Carry out simple asymptotic analysis of algorithms