COMP108
Algorithmic Foundations
Algorithm efficiency

Prudence Wong
http://www.csc.liv.ac.uk/~pwong/teaching/comp108/201617

Learning outcomes

- Able to carry out simple **asymptotic analysis** of algorithms

Time Complexity Analysis

How fast is the algorithm?

Code the algorithm and run the program, then measure the running time

1. Depend on the speed of the computer
2. Waste time coding and testing if the algorithm is slow

Identify some important operations/steps and count how many times these operations/steps needed to be executed

Time Complexity Analysis

How to measure efficiency?

Number of operations usually expressed in terms of input size

- If we doubled/trebled the input size, how much longer would the algorithm take?

Why efficiency matters?

- speed of computation by hardware has been improved
- efficiency still matters
- ambition for computer applications grow with computer power
- demand a great increase in speed of computation

Amount of data handled matches speed increase?

When computation speed vastly increased, can we handle much more data?

Suppose

- an algorithm takes n^2 comparisons to sort n numbers
- we need *1* sec to sort *5* numbers (25 comparisons)
- computing speed *increases by factor of 100*

Using 1 sec, we can now perform ***100x25*** comparisons, i.e., to sort ***50*** numbers

With *100* times speedup, only sort *10* times more numbers!

Time/Space Complexity Analysis

Important operation of summation: ***addition***

How many additions this algorithm requires?

```
sum = 0, i = 1
while i<=n do
begin
  sum = sum + i
  i = i + 1
end
output sum
```

We need **n** additions
(depend on the input size **n**)

We need 3 variables `n`, `sum`, & `i`
$\Rightarrow$ needs 3 memory space

In other cases, space complexity may depend on the input size n

Look for improvement

Mathematical formula gives us an alternative way to find the sum of first n integers:

1 + 2 + ... + n = n(n+1)/2

```
sum = n*(n+1)/2
output sum
```

We only need 3 operations:
1 addition, 1 multiplication, and 1 division
(no matter what the input size n is)

Improve Searching

We've learnt sequential search and it takes n comparisons in the worst case.

If the numbers are pre-sorted, then we can improve the time complexity of searching by **binary search**.

Binary Search

more efficient way of searching when the sequence of numbers is **pre-sorted**

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 (2)

To find 24

3 7 11 12 **15** 19 24 33 41 55 ← 10 nos
24 ← X

19 24 **33** 41 55
24

19 24
24

24
24 found!

Binary Search (3)

To find 30

3 7 11 12 **15** 19 24 33 41 55 ← 10 nos
30 ← X

19 24 **33** 41 55
30

19 24
30

24
30 not found!

Binary Search – Pseudo Code

⌊ ⌋ is the floor function, truncates the decimal part

```
first = 1
last = n
found = false
while (first <= last && found == false) do
begin

// check with no. in middle

end
if (found == true)
  report "Found!"
else report "Not Found!"
```


Binary Search – Pseudo Code

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


Number of Comparisons

Best case:

X is the number in the middle
⇒ 1 comparison

Worst case:

at most $\lceil \log_2 n \rceil + 1$ comparisons

Why?

Every comparison reduces the amount of numbers by at least half

E.g., 16 ⇒ 8 ⇒ 4 ⇒ 2 ⇒ 1

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


Time complexity
- Big O notation ...

Note on Logarithm

Logarithm is the inverse of the power function

$\log_2 2^x = x$

For example,

$\log_2 1 = \log_2 2^0 = 0$

$\log_2 2 = \log_2 2^1 = 1$

$\log_2 4 = \log_2 2^2 = 2$

$\log_2 16 = \log_2 2^4 = 4$

$\log_2 256 = \log_2 2^8 = 8$

$\log_2 1024 = \log_2 2^{10} = 10$

$\log_2 x{*}y = \log_2 x + \log_2 y$

$\log_2 4{*}8 = \log_2 4 + \log_2 8 = 2+3 = 5$

$\log_2 16{*}16 = \log_2 16 + \log_2 16 = 8$

$\log_2 x/y = \log_2 x - \log_2 y$

$\log_2 32/8 = \log_2 32 - \log_2 8 = 5-3 = 2$

$\log_2 1/4 = \log_2 1 - \log_2 4 = 0-2 = -2$

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 (time complexity).

$f_1(n) = 50n + 20$ $f_2(n) = 10\ n \log_2 n + 100$

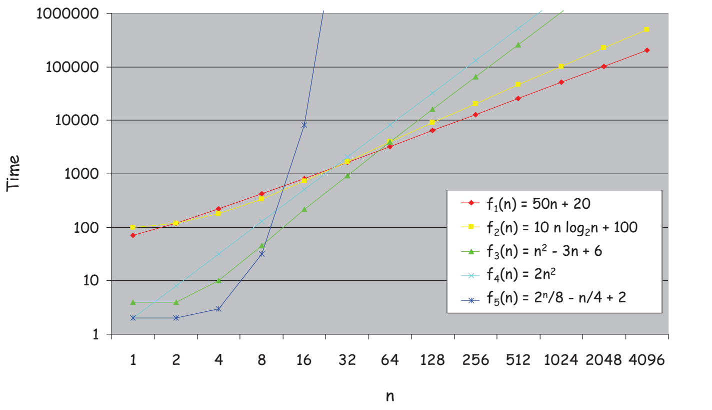
$f_3(n) = n^2 - 3n + 6$ $f_4(n) = 2n^2$

$f_5(n) = 2^n/8 - n/4 + 2$

n	1	2	4	8	16	32	64	128	256	512	1024	2048
$f_1(n) = 50n + 20$	70	120	220	420	820	1620	3220	6420	12820	25620	51220	102420
$f_2(n) = 10\ n \log_2 n + 100$	100	120	180	340	740	1700	3940	9060	20580	46180	102500	225380
$f_3(n) = n^2 - 3n + 6$	4	4	10	46	214	934	3910	16006	64774	3E+05	1E+06	4E+06
$f_4(n) = 2n^2$	2	8	32	128	512	2048	8192	32768	131072	5E+05	2E+06	8E+06
$f_5(n) = 2^n/8 - n/4 + 2$	2	2	3	32	8190	5E+08	2E+18					

Quickest: $f_5(n)$ ⟷ $f_3(n)$ ⟷ $f_1(n)$

Depends on the size of the input!



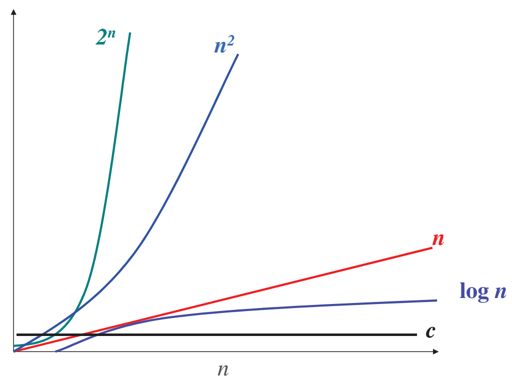
What do we observe?

- There is huge difference between
  - functions involving powers **of** n (e.g., n, n^2, called **polynomial** functions) and
  - functions involving powering **by** n (e.g., 2^n, 3^n, called **exponential** functions)
- Among polynomial functions, those with same order of power are more comparable
  - e.g., $f_3(n) = n^2 - 3n + 6$ and $f_4(n) = 2n^2$

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	

Relative growth rate



Hierarchy of functions

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

1	log n	n n^2 n^3 ... n^k ...	2^n
↑	↑	↑	↑
constant	logarithmic	polynomial	exponential

- We can further refine the hierarchy by inserting **n log n** between **n** and **n^2**, **n^2 log n** between **n^2** and **n^3**, and so on.

Hierarchy of functions (2)

1	log n	n n^2 n^3 ... n^k ...	2^n
↑	↑	↑	↑
constant	logarithmic	polynomial	exponential

Note: as we move from left to right, successive functions have **greater order of growth** than the previous ones.

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

⇒ Relative growth rates increase

Hierarchy of functions (3)

$(\log n)^3$

What about $\log^3 n$ & n?
Which is higher in hierarchy?

Remember: $n = 2^{\log n}$
So we are comparing $(\log n)^3$ & $2^{\log n}$
$\therefore$ $\log^3 n$ is lower than n in the hierarchy

Similarly, $\log^k n$ is lower than n in the hierarchy, for any constant k

Hierarchy of functions (4)

1	log n	n n^2 n^3 … n^k …	2^n
constant	logarithmic	polynomial	exponential

- Now, when we have a function, we can classify the function to some function in the hierarchy:
  - For example, $f(n) = 2n^3 + 5n^2 + 4n + 7$
    The term with the highest power is $2n^3$.
    The growth rate of f(n) is dominated by n^3.
- This concept is captured by **Big-O notation**

Big-O notation

$f(n) = O(g(n))$ *[read as f(n) is of order g(n)]*

- Roughly speaking, this means f(n) is at most **a constant times g(n)** for all large n
- Examples
  - $2n^3 = O(n^3)$
  - $3n^2 = O(n^2)$
  - $2n \log n = O(n \log n)$
  - $n^3 + n^2 = O(n^3)$

Exercise

Determine the order of **growth** of the following functions.

1. $n^3 + 3n^2 + 3$
2. $4n^2 \log n + n^3 + 5n^2 + n$
3. $2n^2 + n^2 \log n$
4. $6n^2 + 2^n$

Look for the term highest in the hierarchy

More Exercise

Are the followings correct?

1. $n^2\log n + n^3 + 3n^2 + 3$ — $O(n^2\log n)$?
2. $n + 1000$ — $O(n)$?
3. $6n^{20} + 2^n$ — $O(n^{20})$?
4. $n^3 + 5n^2 \log n + n$ — $O(n^2 \log n)$?

Some algorithms we learnt

Sum of 1st n integers

```
input n
sum = n*(n+1)/2
output sum
```

O(?)

```
input n, sum = 0
while i <= n do
begin
  sum = sum + i
  i = i + 1
end
output sum
```

O(?)

Min value among n numbers

```
loc = 1, i = 2
while i <= n do
begin
  if (a[i] < a[loc]) then
    loc = i
  i = i + 1
end
output a[loc]
```

O(?)

Time complexity of this?

```
for i = 1 to 2n do
  for j = 1 to n do
    x = x + 1
```

O(?)

The outer loop iterates for **2n** times.
The inner loop iterates for **n** times for each i.
Total: $2n * n = 2n^2$.

What about this?

```
i = 1
count = 0
while i < n
begin
  i = 2 * i
  count = count + 1
end
output count
```

O(?)

suppose **n=8**

(@ end of) iteration	i	count
	1	0
1	2	1
2	4	2
3	8	3

suppose **n=32**

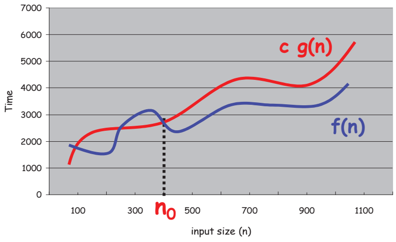
(@ end of) iteration	i	count
	1	0
1	2	1
2	4	2
3	8	3
4	16	4
5	32	5

Big-O notation - formal definition

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

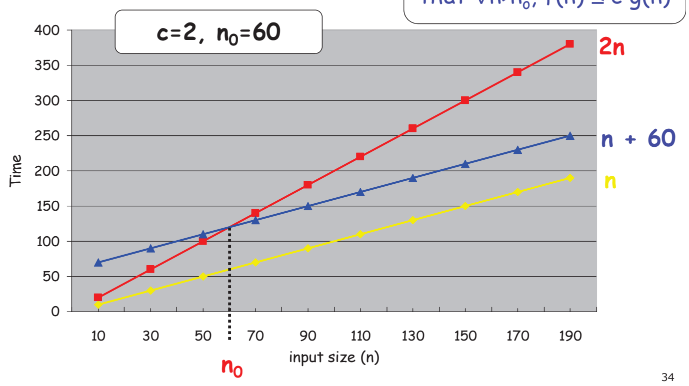
- There exists a constant c and n_0 such that $f(n) \le c\ g(n)$ for all $n > n_0$
  - $\exists c\ \exists n_0\ \forall n > n_0$ then $f(n) \le c\ g(n)$

Graphical Illustration



Example: n+60 is O(n)

$\exists$constants c & n_0 such that $\forall n > n_0$, $f(n) \le c\ g(n)$



Which one is the fastest?

Usually we are only interested in the *asymptotic* time complexity

- i.e., when n is large

$O(\log n) < O(\log^2 n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(2^n)$

Proof of order of growth

- Prove that $2n^2 + 4n$ is $O(n^2)$
  - ✓ Since $n \le n^2\ \forall n \ge 1$,

    we have

    $2n^2 + 4n \le 2n^2 + 4n^2$

    $= 6n^2 \quad \forall n \ge 1.$
  - ✓ Therefore, by definition, $2n^2 + 4n$ is $O(n^2)$.

Note: plotting a graph is NOT a proof

- Alternatively,
  - ✓ Since $4n \le n^2\ \forall n \ge 4$,

    we have

    $2n^2 + 4n \le 2n^2 + n^2$

    $= 3n^2 \quad \forall n \ge 4.$
  - ✓ Therefore, by definition, $2n^2 + 4n$ is $O(n^2)$.

Proof of order of growth (2)

- Prove that $n^3 + 3n^2 + 3$ is $O(n^3)$
  - ✓ Since $n^2 \le n^3$ and $1 \le n^3\ \forall n \ge 1$,

    we have

    $n^3 + 3n^2 + 3 \le n^3 + 3n^3 + 3n^3$

    $= 7n^3 \quad \forall n \ge 1.$
  - ✓ Therefore, by definition, $n^3 + 3n^2 + 3$ is $O(n^3)$.
- Alternatively,
  - ✓ Since $3n^2 \le n^3\ \forall n \ge 3$, and $3 \le n^3\ \forall n \ge 2$

    we have

    $n^3 + 3n^2 + 3 \le 3n^3 \quad \forall n \ge 3.$
  - ✓ Therefore, by definition, $n^3 + 3n^2 + 3$ is $O(n^3)$.

Challenges

Prove the order of growth

1. $2n^3 + n^2 + 4n + 4$ is $O(n^3)$
   a) $2n^3 = 2n^3 \quad \forall n$
   b) $n^2 \le ?? \quad \forall n \ge ?$
   c) $4n \le ?? \quad \forall n \ge ?$
   d) $4 \le ?? \quad \forall n \ge ?$

   $\Rightarrow 2n^3 + n^2 + 4n + 4 \le ?? \quad \forall n \ge ?$
2. $2n^2 + 2^n$ is $O(2^n)$
   a) $2n^2 \le ?? \quad \forall n \ge ?$
   b) $2^n = 2^n \quad \forall n$

   $\Rightarrow 2n^2 + 2^n \le ?? \quad \forall n \ge ?$

Plot: