COMP108
Data Structures and Algorithms

Algorithm Efficiency (Part I)

Professor Prudence Wong

pwong@liverpool.ac.uk

2020-21

Outline

Measuring algorithm efficiency

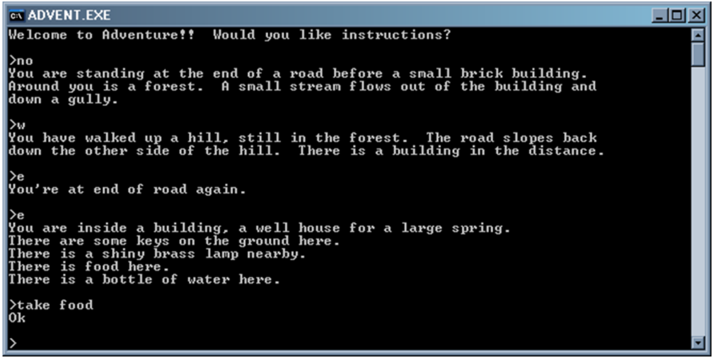
- Why efficiency matters?
- Big-O notation
- Examples

Learning outcome:

- Able to carry out simple asymptotic analysis of algorithms

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



Time complexity analysis

How fast is the algorithm?

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

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

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?
- If we doubled/trebled the speed of computation, how much more data we can handle?

Amount of data handled matches speed increase?

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

Assume this initial scenario:

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

Amount of data handled matches speed increase?

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

Assume this initial scenario:

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

Now suppose that

- computing speed increases by factor of 100

speed ↑ ×100

1 sec ⇒ 25×100

= 2500 comparisons

⇓

50 numbers

Amount of data handled matches speed increase?

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

Assume this initial scenario:

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

Now suppose that

- computing speed increases by factor of 100

What happens now?

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 complexity analysis

Important operation of summation: *addition*

How many additions this algorithm requires?

sum $\leftarrow$ 0, i $\leftarrow$ 1
while i $\leq$ n do
begin
    sum $\leftarrow$ sum + i
    i $\leftarrow$ i + 1
end
output sum

Time 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 $2n$ additions (depend on the input size n)

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).

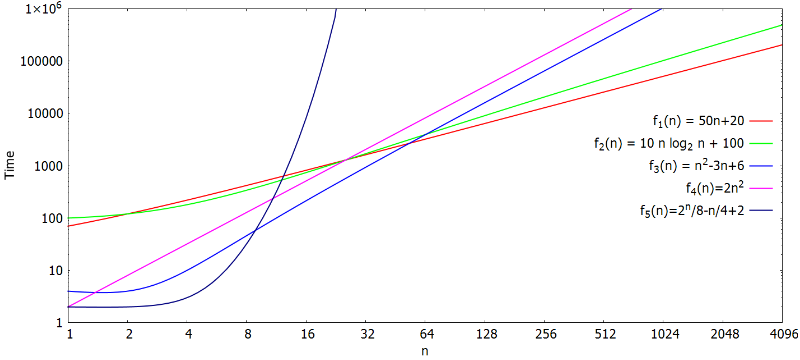
$$f_1(n) = 50n + 20 \qquad f_2(n) = 10n\log_2 n$$
$$f_3(n) = n^2 - 3n + 6 \qquad f_4(n) = 2n^2$$
$$f_5(n) = 2^n/8 - n/4 + 2$$

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

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

Depend on the input size!

Which algorithm is the fastest?



What do we observe?

There is huge difference between functions involving

- powers of n (e.g., n, n^2) called polynomial functions and
- powering by n (e.g., 2^n, 3^n) called exponential functions

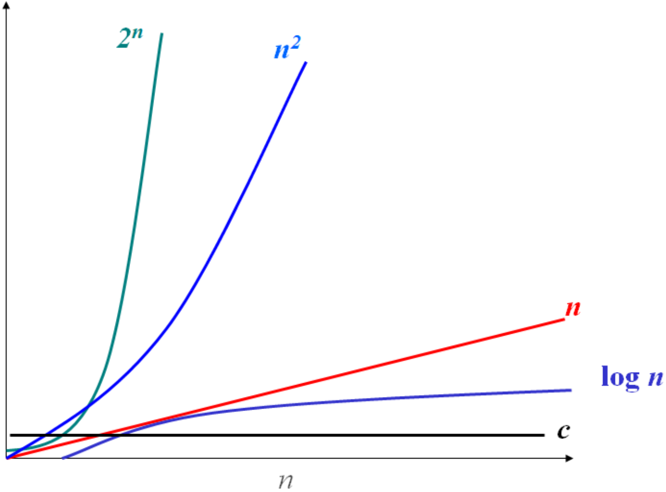
Among polynomial functions, those with same order of power are more comparable

- e.g., $f_3(n) = n^2 - 3n + 6$ and $f_4(n) = 2n^2$

Relative growth rate

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:

$n \times \log n$

$n \times n$

1	$\log n$	$n \; n^2 \; n^3 \cdots n^k \cdots$	2^n
$\Uparrow$	$\Uparrow$	$n \log n$ $\Uparrow$ $n^2 \log n$	$\Uparrow$
constant	logarithm	polynomial	exponential

- We can further refine the hierarchy by inserting
  - $\boldsymbol{n \log n}$ between $\boldsymbol{n}$ and $\boldsymbol{n^2}$
  - $\boldsymbol{n^2 \log n}$ between $\boldsymbol{n^2}$ and $\boldsymbol{n^3}$, and so on.

Hierarchy of functions (2)

$\log_2$

$2^{\log_2 n} \equiv n$

$3^{\log_3 n} \equiv n$

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

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

n^3 2^n

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

$\log n \quad \log^2 n \quad \log^3 n \ldots \log^k n \ldots \mid n$

Hierarchy of functions (3)

1	$\log n$	$n\ n^2\ n^3 \cdots n^k \cdots$	2^n
$\Uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow$
constant	logarithm	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.

$\Rightarrow$ Relative growth rates increase

Hierarchy of functions (4)

1	$\log n$	$n\ n^2\ n^3 \cdots n^k \cdots$	2^n
$\Uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow$
constant	logarithm	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.

$O(n^3)$

- This concept is captured by Big-O notation

Big-O notation

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

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

Big-O notation

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

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

When we have an algorithm, we can then measure its time complexity by

- counting number of operations in terms of the input size
- expressing it using big-O notation

Big-O notation

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

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

When we have an algorithm, we can then measure its time complexity by

- counting number of operations in terms of the input size
- expressing it using big-O notation

We can then compare the efficiency of two algorithms doing the same task by

- comparing their time complexities in terms of big-O notation

Summary: Measuring algorithm efficiency

Next: Exercises

For note taking