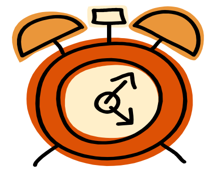
Analysis of algorithms (Outcome (1))

Q: how good a given algorithm is?

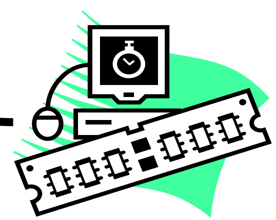
Recall

Two major concerns



Time

- Not measure actual time (why?)
- Can be used to predict the actual running time and for comparisons of algorithms



Space

- Also not measure actual memory used (why?)
- memory/harddisk space used by data structures and working space used by the algorithms

Other issues (not our concern):
e.g. is it difficult to implement?

Remarks:

(1)We concern time & space when the data volume is huge (why?)

(2)The actual running time/space used still important => second stage of comparison/evaluation

If we do not implement the algorithm, how we can measure the "running time" of it?

Answer:

We count # of steps (or operations) to be executed.

How?

- What are primitive operations?

- How to count, different inputs may require different # of operations?

- How to represent this "running time"?

Issue (1): What to count?

Issue (1a): What are primitive operations?

Primitive operations $\approx$ the basic operations provided by a programming language (e.g. +, -, x, $\div$, relational operators: >, <, $\geq$, $\leq$, logical operators: and, or, not).

Remark: Exact definition not important, we will see why later.

Issue (1b): Is it fair to compare $\times$ operation with + operation, they may take different time to complete?

Answer: YES

Consider 3 different program fragments for computing n^2.

Assuming that it takes:
c1 units to perform ×; c2 units for +; c3 units for increment (++) [c1 > c2 > c3]

Let c1 = 200; c2 = 10; c3 = 1

(1) Sum = n × n;	1 "×" operation	c1	200
(2) Sum = 0; For I = 1 to n Sum = Sum + n;	n "+" operations	$c2 \times n$	10n

(3)
Sum = 0;
For I = 1 to n
    For J = 1 to n
        Sum ++;
	n^2 "++" increment operations	$c3 \times n^2$	n^2

(1) 200 (2) $10n$ (3) n^2

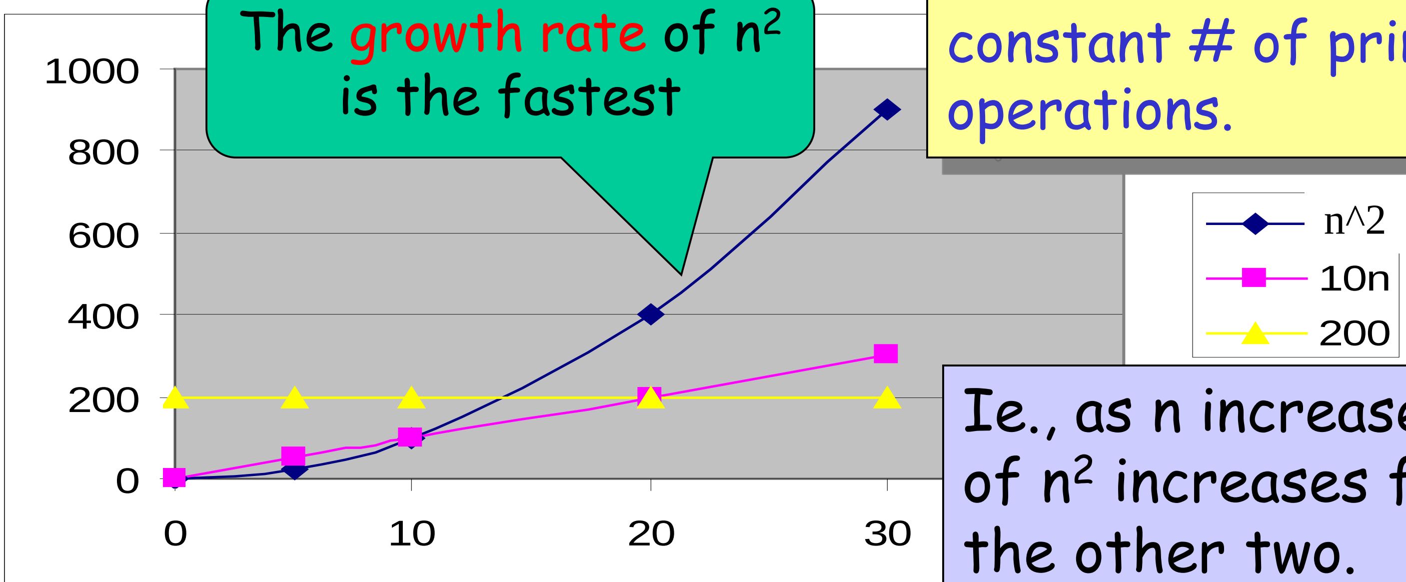
n	Fastest algorithm
1 – 10	(3) [n^2]
10 – 20	(2) [$10n$]
>= 20	(1) [200]

The constant (the coefficient) does not really matter when n is large.

When the input size is large,
(1) is the best algorithm

Implication: We do not need to classify primitive operations into different types (such as ×, +, / etc.).
=> We count # of statements if each statement only contains a constant # of primitive operations.

The growth rate of n^2 is the fastest



Ie., as n increases, the value of n^2 increases faster than the other two.

(1) 2^n operations (2) 1000000000n operations

1GHz computer: execute ~10^9 operations per second

If the input size n is 64,

then 1000000000n = 64000000000

the computer can finish the algorithm in 64sec

but then 2^n = 18,446,744,073,709,551,615

the computer will run ~18,446,744,073 sec(~600 years)

In fact, for some small values of n, 2^n < 1000000000n

But, the growth rate of 2^n is much much faster than 1000000000n

Note

So, for theoretical analysis, we are interested in the growth rate and do not care about the constant.

However, this constant does affect the practical performance of the program. We still need to care about this in some practical cases!

Issue (2): Best case, worst case, and average case

An algorithm can solve a problem for many different inputs; for some input, it may run faster, for some input, it has its worst case performance.

Consider a simple searching problem. Given an integer x and an array A[1..5] of integers, check if x is in the array or not.

```
find(A, x) {
   I = 1;
   while (x != A[I])
        if (I == 5) return "not found";
        else I++;
   return ("found");
}
```

A = [10, 45, 3, 25, 50]

Best => 1) If x = 10;

2) If x = 50;

3) If x = 3;

Worst => 4) If x = 60;

Usually we consider the worst case when analyzing an algorithm because we want to get an "upper bound" (or a guarantee) on the performance.

Note: sometimes, we also consider average case analysis

Let J be a possible input to algorithm A.
Let $f_A(J)$ be the running time of A with input J

Best case refers to the case that $f_A(J)$ is smallest over all possible J

Worst case refers to the case that $f_A(J)$ is largest over all possible J.

Average Case
Let prob(J) be the probability that J will occur.
Then, average case refers to the value of $\Sigma \text{prob}(J) \times f_A(J)$

```
find(A, x) {
    I = 1;
    while (x != A[I])
        if (I == 5) return "not found";
        else I++;
    return ("found");
}
```

A = [10, 45, 3, 25, 50]

Refer to the previous example, let $f_A(10) = 3$; $f_A(45) = 6$; $f_A(3) = 9$; $f_A(25) = 12$; $f_A(50) = 15$; $f_A(I) = 16$ for other I and the probability that x = 10, 20, 3, 25, or 50 is 1/10; the probability that x = other values is 1/2, then the average case of running time is:

$$1/10(3 + 6 + 9 + 12 + 15) + 1/2(16) = 12.5$$

Finding average case is not easy as the probability distribution of input is usually unknown and is harder to calculate.

Remark: This measure is a function of n, the input (data) size!

```
find(A, x) {
   I = 1;
   while (x != A[I])
         if (I == n) return "not found";
         else I++;
   return ("found");
}
```

Worst case:
~(3 x n + 1) operations

Issue 1(c): Do we need to count every statement?

Algorithm 1: n^3 statements

Algorithm 2: $100n^2+100n$ statements

(1) n^3 (2) $100n^2+100n$

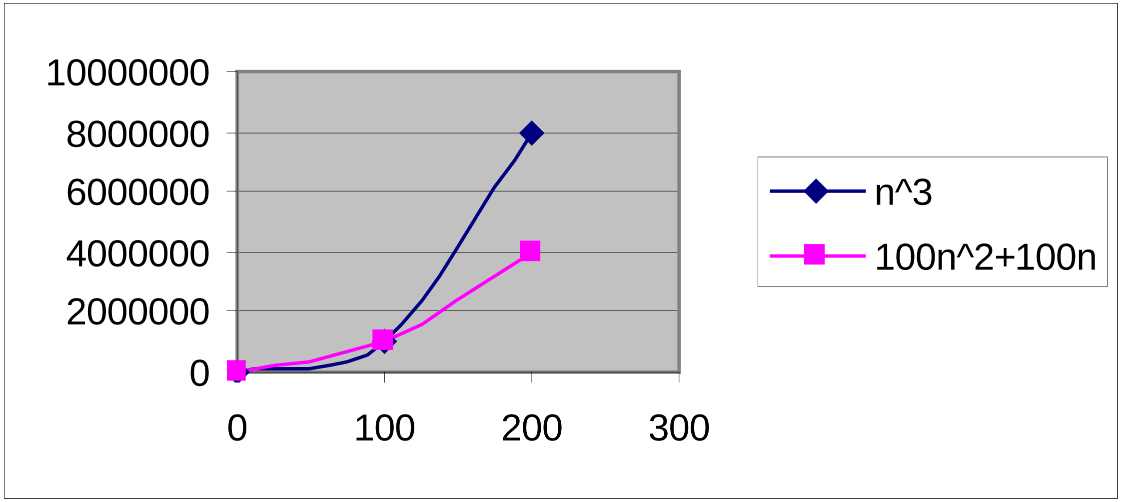
Which one is faster if n is large

n	Fastest algorithm
Small	(1)
Large	(2)

dominating part

Theoretical analysis:

- Drop "low-order" term
- Ignore leading constants
- Interested in "growth rate of total number of operations as a function of input size".



Another example

	Cost	# of executions
for I = 1 to n	c1	n+1
for j = 1 to n	c2	(n+1)n
A[j] += 10;	c3	n^2
for k = 1 to n	c4	$(n+1)n^2$
B[k] += 5;	c5	n^3

Constant

$2n^3+3n^2+2n+1$

The dominating term

$$c1(n+1) + c2(n+1)n + c3(n^2) + c4(n+1)n^2 + c5n^3$$
$$= (c4+c5)n^3 + (c2+c3+c4)n^2 + c2\ n + c1$$

So, we can just focus on the statement where the # of executions is the most to get the dominating term.

A short summary

How to measure an algorithm's running time?

- Count the total no. of primitive operations that the algorithm executes in (1) worst case; (2) average case; (3) best case. (usually we are interested in worst case analysis).

- The result is a function in terms of n, the input problem size.

-What we care (From the theoretical analysis point of view): When n increases, whether the total number of operations to be executed increases dramatically – Growth rate of the function.

- It implies that:
    - No need to distinguish different types of primitive operations (just count # of statements)
    - We only need to compare the dominating term of the function: (1) constants/coefficients not important; (2) lower order terms not important.

How to capture this concept mathematically?

Review

```
for i = 1 to n
    for j = 1 to n
        A[j] += 10;
for k = 1 to n
        B[k] += 5;
```

Dominating statement
(# of execution: n^2)

Dominating statement
(# of execution: nm; don't care # of operations)

```
for i = 1 to n
    for j = 1 to m
        A[j] = A[j] * A[i-1] +10 – A[i*j]/5;
for k = 1 to n
        B[k] += 5;
```

```
if n is odd  // assume n is non-negative
    n = 0;
else
    while (n ≠ 0)
        n = n div 2      //get the quotient of n divided by 2
```

Dominating statement (worst case)
(# of execution: ??) $\approx \log_2 n$

How to capture this concept mathematically?

Preliminaries

1) Functions $f(n)$
   - Our scope: running time of algorithms, number of operations etc.
   - Domain: $n = 1, 2, 3, ...$
   - Only consider +ve functions, i.e., $f(n) > 0$.

2) Effects of multiplying $f(n)$ by a constant c
   - Example: $f(n) = n$; show the graph for $f(n) = 2n$; $f(n) = \frac{1}{2} n$. What you notice?

     No matter what n you choose, $f(n) = n$ is always in between $f(n) = 2n$ and $f(n) = \frac{1}{2} n$.

   - Try again with $f(n) = cn$ where c is a constant and $f(n) = n^2$, we can try $c = 2, 10, ...$, what you notice?

Preliminaries

3) Set
   - Let A = {all functions with coefficient of $n^2 = 4$}.
     - e.g. $4n^2 + \log n$ (Yes); $8n^3+2n^2+4$ (no); $10n^6 + 5$ (no).
   - Notations: $\exists$ (there exists); $\forall$ (for all).
     - e.g. Let B = {all functions $f(n)$ s.t. $\exists$ a constant $n_0 \geq 1$, $f(n) > 100\ \forall\ n \geq n_0$}.
       - e.g. $f(n) = n$ (yes); $f(n) = n/2$ (yes); $f(n) = 1/n$ (no) (proof?)

Asymptotic Notation

e.g. $10000n^2$, $n^2 + 10000n$, $0.5n^2 + 1000000n$

If you understand our discussion so far, you will agree that they have the same growth rate.

So, we have to develop a mathematical definition so that these functions should be treated as "the same".

Ignore the effect of Lower order terms and coefficients

Short review:

(i) Represent running time of an algorithm using a function of n, where n is the input size based on worst (or average) input case.

(ii) Only care about the dominating term (with the fastest growth rate) of the function.

```
m = 2        // assume n is a +ve integer //
while (m * m) <= n
        if n is divisible by m, stop.
        else m = m+1
Report "n is a prime number"
```

Running time= $\sim\sqrt{n}$

(iii) Asymptotic notation: Big-Theta (Θ)

Same growth rate or not?

(a) $0.00000001n^3$ vs $10000n^3$

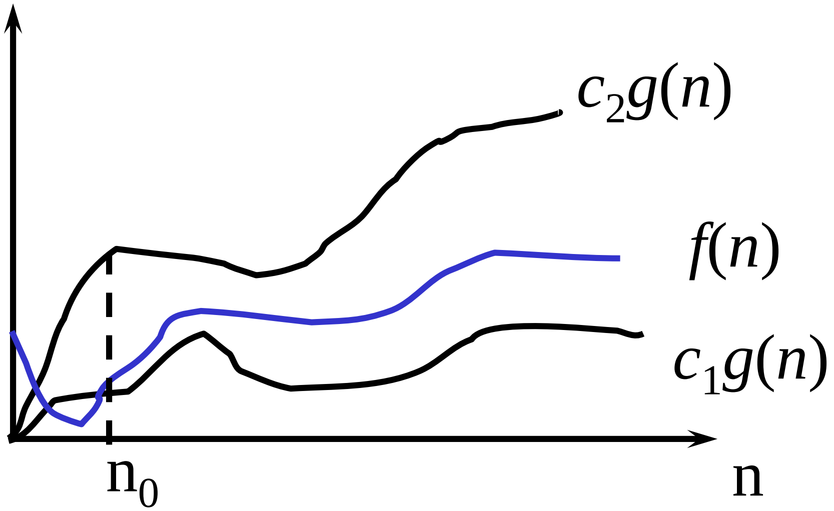
(b) $1000000n^2$ vs $0.00001n^3$

(c) n^3 vs $n^3 + 999999999n^2$

Q: Still remember how to capture this concept formally?

Idea:
f(n) and g(n) have the same growth rate if we can find two constants c1, c2 > 0 such that:

Intuitive meaning



Recall: in our domain (algorithm analysis), we can assume that the functions are all positive.

Small n:
Don't care!

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

$\Theta(g(n))$ is defined as a set of functions with same growth rate as g(n)

One can also write $g(n) \in \Theta(f(n))$;

Asymptotic running time of an algorithm

Big Theta (Θ notation)

Roughly speaking, for any given function *g*(n), we denote the set of all functions that have the same growth rate as *g*(n) by $\Theta(g(n))$.

e.g. $n^2 + 10000n \in \Theta(n^2)$; $10000n^2 \in \Theta(n^2)$ etc.

Formal definition

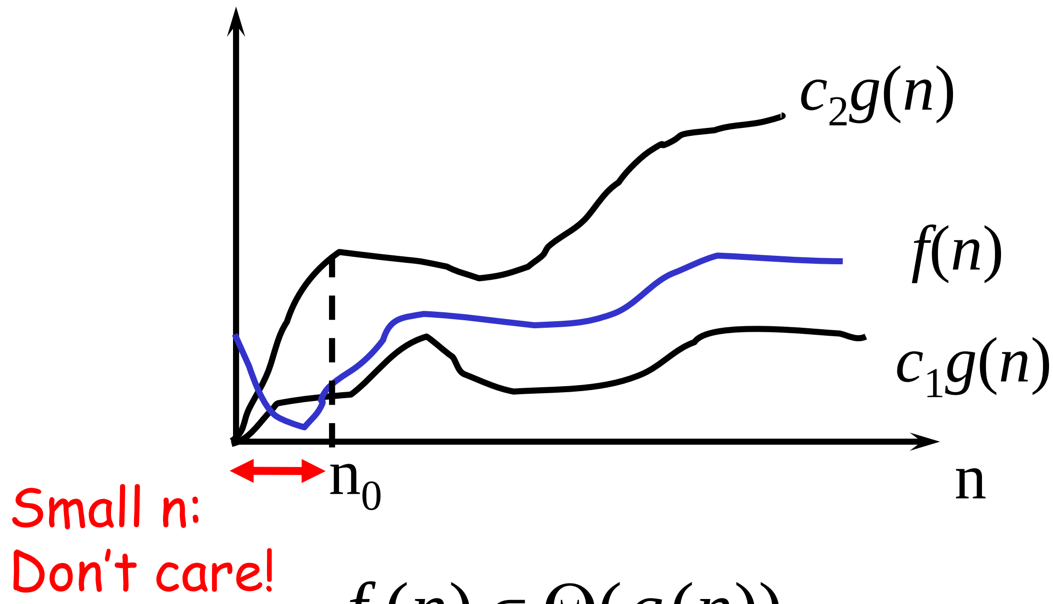
$$\Theta(g(n)) = \{ f(n) : \exists \text{ positive constants } c_1, c_2, n_0$$
$$\text{such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall n \ge n_0 \}$$

Domain of *n*: natural numbers

($\exists$: there exists; $\forall$: for all)

In English: a function f(n) belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n.

Intuitive meaning



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

Recall: in our domain (algorithm analysis), we can assume that the functions are all positive.

If $f(n) \in \Theta(g(n))$, we say that g(n) is an <u>asymptotically tight bound</u> for f(n) or f(n) behaves asymptotically as g(n).

~ f(n) and g(n) have the same growth rate

Example:

Show that $100n^2 + 20n + 5 \in \Theta(n^2)$.

Can we find constants c_1, c_2, and n_0 s.t. whenever $n \geq n_0$, we have:

$$c_1 n^2 \leq 100n^2+20n+5 \leq c_2 n^2$$

$$c_1 \leq 100+20/n+5/n^2 \leq c_2$$

Yes, let $c_1 = 100$; $c_2 = 125$, $n_0 = 1$

we can always have "$c_1 \leq 100+20/n+5/n^2 \leq c_2$" when $n \geq n_0$

So, $100n^2 + 20n + 5 \in \Theta(n^2)$

Example:

Show that $n^3 \notin \Theta(n^2)$.

Assume that there exist positive constants c_1, c_2, n_0 such that $c_1 n^2 \leq n^3 \leq c_2 n^2$ whenever $n \geq n_0$.

$\Rightarrow c_1 \leq n \leq c_2$ whenever $n \geq n_0$

Contradictions.

[Can you point out the part that has the contradictions?]

Classroom exercises:

(1) Show that $n^2 \in \Theta(100n^2 + 20n + 5)$.

(2) $n^2 \notin \Theta(n^3)$.

(3) $n^3 \notin \Theta(n^4)$.

Abuse of equality

<u>In practice:</u>

If $f(n) \in \Theta(g(n))$, we usually write it as $f(n) = \Theta(g(n))$

We use $\Theta(g(n))$ informally to mean "some function that is in the set $\Theta(g(n))$

e.g. $20n^4 + 3n = \Theta(n^4)$

Note: $\Theta(1)$ refers to a constant or a constant function

e.g. $1{,}000{,}000 = \Theta(1)$? $\quad 0.0000001n = \Theta(1)$?

Big Theta (Θ notation)

Big O (O notation)

Sometimes, we just want to bound the growth rate of a given function f(n), saying that the growth rate of f(n) is at most the same as that of g(n).

$f(n) \in O(\mathrm{g(n)})$; $f(n) = O(g(n))$

Asymptotic upper bound

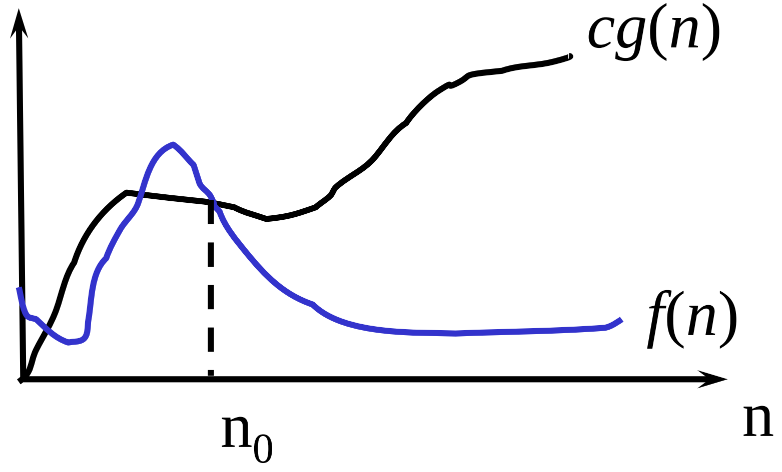
Only required to be upper bounded (within a constant factor)

Format definition

$$O(g(n)) = \{ f(n) : \exists\, c > 0, \mathrm{n}_0 > 0$$
$$\text{such that } \forall n \ge n_0,\ 0 \le f(n) \le cg(n) \}$$

$$\Theta(g(n)) = \{ f(n) : \exists \text{ positive constants } c_1, c_2, n_0$$
$$\text{such that } 0 \le \mathrm{c}_1 g(n) \le f(n) \le \mathrm{c}_2 g(n), \forall n \ge n_0 \}$$

Intuitive meaning



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

<u>Examples:</u>

$6n^3 + 10n^2 + 4 = O(n^2)$

$6n^3 + 10n^2 + 4 = O(n^3)$

$6n^3 + 10n^2 + 4 = O(n^4)$

$6n^3 + 10n^2 + 4 = \Theta(n^3)$

$6n^3 + 10n^2 + 4 = \Theta(n^2)$

$6n^3 + 10n^2 + 4 = \Theta(n^4)$

Big Theta (Θ notation) Asymptotic tight bound

Big O (O notation) Asymptotic upper bound

Big Omega (Ω notation) Asymptotic lower bound

$$\Omega(g(n)) = \{ f(n) : \exists c > 0, \mathrm{n}_0 > 0$$
$$\text{such that } \forall n \geq n_0, \ 0 \leq cg(n) \leq f(n)\}$$

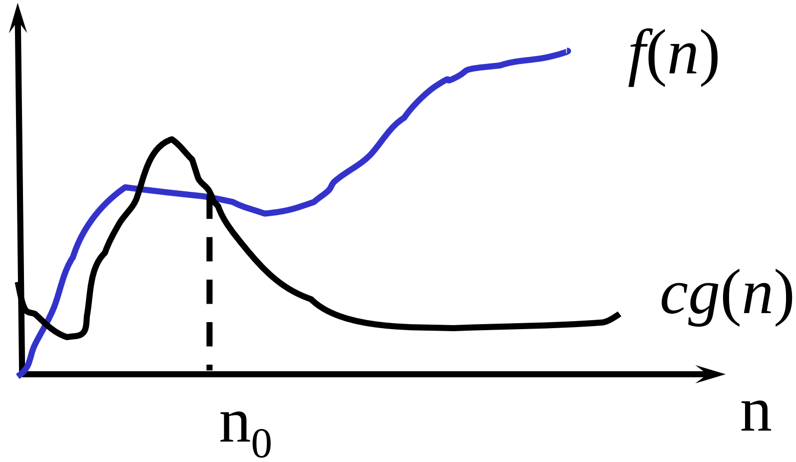
$$\Theta(g(n)) = \{ f(n) : \exists \text{ positive constants } c_1, c_2, n_0$$
$$\text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$$

$$O(g(n)) = \{ f(n) : \exists c > 0, \mathrm{n}_0 > 0$$
$$\text{such that } \forall n \geq n_0, \ 0 \leq f(n) \leq cg(n)\}$$

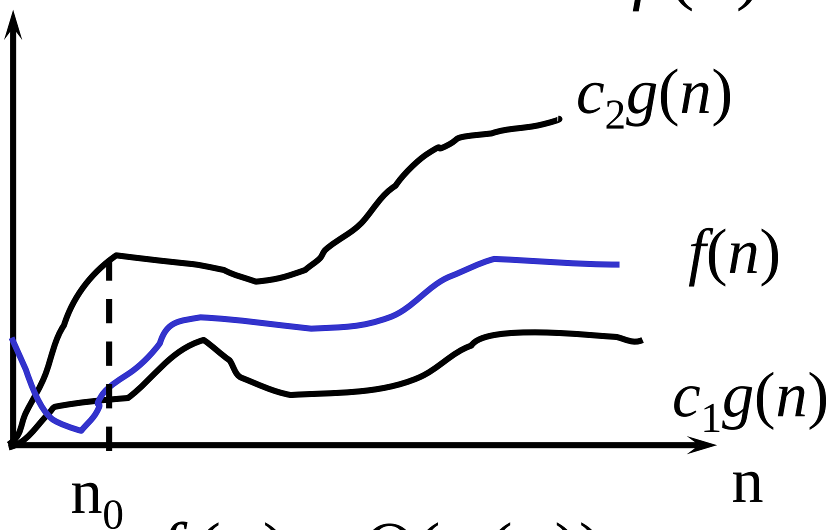
Examples:

$5n^3 + 8n^2 + 4 = \Omega(n^3)$; $5n^3 + 8n^2 + 4 = \Omega(n^2)$; $5n^3 + 8n^2 + 4 = \Omega(n^4)$

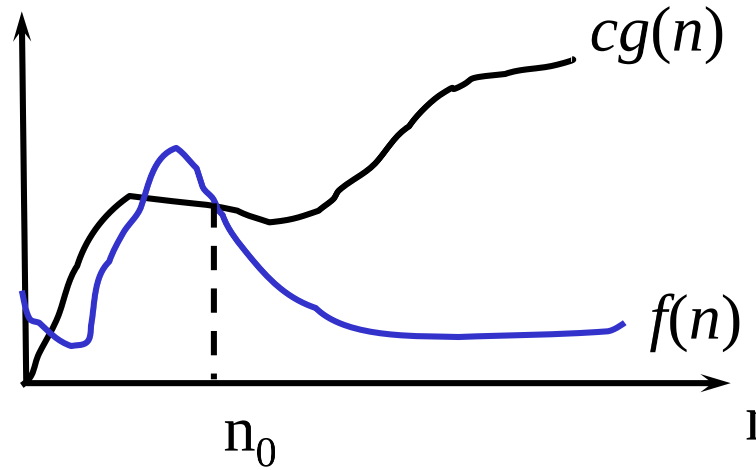
Intuitive meaning



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



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



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

We have $f(n) = \Theta(g(n))$ iff $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$

Asymptotic notation in equations (and inequalities)

Example 1

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

$\Theta(n)$: represents a function $f(n) \in \Theta(n)$

No interested in the exact details of the lower order terms.

Example 2

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

For any possible $f(n) \in \Theta(n)$, we can show that $2n^2+f(n)= \Theta(n^2)$

Intuitively, it is simple as for any function in $\Theta(n)$, it will be of lower order than $2n^2$.

Example 3

$O(n) + O(n^2) = O(n^2)$

For any possible $f(n) \in O(n)$, $g(n) \in O(n^2)$, we can show that $f(n) + g(n) \in O(n^2)$

Q: $O(1) + O(1) = O(1)$?

Some Simple rules:

$$
\begin{cases}
f(n) = O(f(n)) \\
O(f(n)) + O(g(n)) = O(f(n)+g(n)) \\
O(f(n))\, O(g(n)) = O(f(n)g(n)) \\
n^m = O(n^p) \quad \mathrm{m} \le p \\
c = O(1) \quad c \text{ is a constant}
\end{cases}
$$

Can you prove these rules?

Now, you are ready to apply this notation to analyze the running time of an algorithm.

Reminder:

When we measure the running time of an algorithm,
1) We count the no. of primitive operations (statements) to be executed in the algorithm (usually consider the worst case);
2) This count will be a function in terms of n, the input data size;

And finally we will use the asymptotic notation to represent this function

Usage (examples):

The algorithm runs in $O(n^2)$ time.
The time complexity of this algorithm is $O(n^3)$.
The running time of the algorithm is $O(n)$.
$O(n)$: read as "order n".

Some simple rules for calculating the time complexity of an algorithm:

Sequential statements:

S1;
S2;

Complexity(S1) + Complexity(S2)

Example:

I = I + 1;
J = J*3;

O(1) + O(1) = O(1)

Loops:

e.g. while (cond) do
S;

[Complexity(cond) + Complexity(S)]× no. of iterations

Example:

While (N > 0)
N--;

N(O(1) + O(1)) = O(N)

If-else statements:

```
If (cond) S1
Else S2
```

Let Complexity(S1)+Complexity(cond) = $O(f(n))$;
Complexity(S2)+Complexity(cond) = $O(g(n))$,
Complexity = $O(g(n))$ if $f(n) = O(g(n))$
Otherwise Complexity = $O(f(n))$

Example:

```
If (N <= 0)
   I ++;
Else
   for j= 1 to N
       k++;
```

If-path:
$O(1)+O(1) = O(1)$
Else-path:
$O(1) + N \times (O(1)+O(1)) = O(N)$

So, complexity = $O(N)$

Function Call:

```
F1(…);
```

Analyze complexity of F1 and then follow other rules

Example:

```
For j=1 to N { f1();}

void f1(){
  for I = 1 to N
      sum++;
}
```

Complexity of f1= $O(N)$;
Overall complexity = $O(N^2)$

Some more examples

```
for I = 1 to n
     for J = 1 to n
          sum += 10;
```

$O(n^2)$

Yes, but the analysis (bound) is not tight

```
for I = 1 to n
     sum += 10;
```

$O(n)$

1) Can we say that this code segment runs in $O(n^2)$ time?

2) Can we say that this code segment runs in $\Theta(n^2)$ time?

No.

3) Can we say that this code segment runs in $\Theta(n)$ time?

Yes.

Some typical complexity (growth rate)

Increasing order of growth rate

- $O(1)$, constant time
- $O(1\log n)$, logarithmic complexity
- $O(n)$, linear time
- $O(n \log n)$
- $O(n^2)$, quadratic time
- $O(n^3)$, cubic time
- $O(n^c)$ c: const, polynomial time

- $O(2^n)$, exponential time

"Fast" solution: a problem has a fast solution if we have an algorithm to solve it and runs in polynomial time in the worst case

In practice, we hope to have $O(n^3)$ algorithms!

Many others: e.g. $O(n^{0.5})$, $O(n/\log n)$ etc.

Hard problem: there are many problems which people are still not able to come up with a fast solution. Existing algorithms for solving these problems run in exponential time.

Some ideas of actual running time

Input size	n	$n\log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
10	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
30	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
50	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
100	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
1,000	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
10,000	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
100,000	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
1,000,000	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Remarks: (1) very long = over 10^{25} years.
(2) It won't help even if CPU is 1000 times faster

Slide from Prof. Lam's 0250 course

More examples (Give, the worst case running times of the following code segment using the "big O" notation)

1) for i = 1 to 100,000,000,000 do
        A[i] = A[i] + 10;

$O(1)$

2) for i = 1 to n do
        for j = 1 to i do
                A[j] = A[j] + 10;
    for k = 1 to n do
        B[k] = B[k] + 5;

$1 + 2 + 3 + \ldots + n = n(n+1)/2 = O(n^2)$

Overall time complexity = $O(n^2)$

$O(n)$

Reminder: it is correct to say that it runs in $O(n^3)$.

3) for i = 1 to n do
        for j = 1 to i do
                A[j] = A[j] + 10;
    for k = 1 to m do
        B[k] = B[k] + 5;

$O(n^2 + m)$

```
4) for j = 2 to n do        // n is the size of the array A[1..n] of integers
        temp = A[j]
        i = j – 1
        while (i > 0) and (A[i] > temp) do {
                A[i+1] = A[i]
                i = i – 1}
        A[i+1] = temp
```

a) What does this algorithm do?
b) What is the worst case time complexity?
c) What is the best case time complexity?

a) It sorts the entries in the array A in increasing order.
   Proof of correctness
   Loop invariant: At the start of each iteration of for-loop, elements in A[1..j-1] is always in sorted order.
   [How to show that it is correct? p.18-20 of MIT, not needed in the exam]

4) for j = 2 to n do

```
    temp = A[j]                                   n – 1
    i = j – 1                                     n – 1
    while (i > 0) and (A[i] > temp) do
        A[i+1] = A[i]
        i = i – 1
    A[i+1] = temp                                 n – 1
```

Can you think of a worst case input for this while loop?

e.g. n, n-1, ..., 3, 2, 1

1 + 2 + ... + n-1

a) What does this algorithm do?
b) What is the worst case time complexity?
c) What is the best case time complexity?

b) The worst case time complexity is $O(n^2)$

Q: Does it imply that the running time is always $O(n^2)$ for all possible input?

Q: Can we say that the algorithm runs in $\Theta(n^2)$ time in the worst case? Yes.

Q: Does it imply that the algorithm runs in $\Theta(n^2)$ time for all cases? No.

Can you give a counter example? e.g. 1, 2, 3, ..., n-1, n

c) For the best case, the algorithm runs in $\Theta(n)$.

Q: Can we say that the algorithm runs in $\Omega(n)$ time for the best case? Yes.

Q: Does it imply that the algorithm runs in $\Omega(n)$ time for all cases? Yes.

Remark: This measurement is for theoretical analysis.
e.g. Algorithm A that executes n^2 statements may run faster than Algorithm B that executes 10000000000n statements.

Big Theta (Θ notation) "=" Little o (o notation) "<"

Big O (O notation) "$\leq$" Little ω (ω notation) ">"

Big Omega (Ω notation) "$\geq$"

o notation

We use o notation to denote an upper bound that is not asymptotically tight (c.f. Big O)

$$o(g(n)) = \{ f(n) : \text{for any } c > 0, \exists\, \mathrm{n}_0 > 0 \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) < cg(n) \}$$

e.g. $2n = o(n^2)$, but $2n^2 \neq o(n^2)$ [Do you know how to prove it?]

To show $2n = o(n^2)$:
For any $c > 0$, take $n_0 = \lceil 2/c \rceil + 1$.
Then for all $n \geq n_0$, consider n^2. $n^2 = n \times n > 2n/c$
$\Rightarrow 2n < cn^2$.

So, $2n = o(n^2)$,

To show $f(n) \neq o(g(n))$, you need to find a constant $c > 0$, for any n_0, $\exists n \geq n_0$ such that $f(n) \geq cg(n)$.

On the other hand, to show that $2n^2 \neq o(n^2)$

Let $c = 1$.
$2n^2 > n^2$ for all n.
So, $2n^2 \neq o(n^2)$

ω notation

We use ω notation to denote an lower bound that is **not** asymptotically tight (c.f. Big Omega)

$$\boxed{\begin{aligned}\omega(g(n)) = \{ f(n) : &\text{for any } c > 0, \exists\, \mathrm{n}_0 > 0 \\ &\text{such that } \forall n \geq n_0,\ 0 \leq cg(n) < f(n)\}\end{aligned}}$$

e.g. $2n^2 = \omega(n)$, but $2n \neq \omega(n)$ [Do you know how to prove it?]

Alternative definitions

Recall: Limit

$$\lim_{n\to\infty} f(n)$$

When n tends to be very large, what is the value f(n) tends to?

e.g. $$\lim_{n\to\infty}(n^2+5)=\infty;$$

$$\lim_{n\to\infty}\left(\frac{2n^2+5}{n^2}\right)=2;$$

$$\lim_{n\to\infty}\left(\frac{1000n+20\sqrt{n}}{n^2}\right)=0;$$

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

When n tends to be very large, what is the ratio of the values of f(n) and g(n) tends to?

e.g. $f(n)=\frac{4n+3}{n+2}, g(n)=\frac{3n+8}{n};\qquad \lim_{n\to\infty}\frac{f(n)}{g(n)}=\frac{4}{3}$

Q: if $\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty,$ what do you think about the growth rates of f(n) and g(n)? f(n) = ω(g(n))

Q: if $\lim_{n\to\infty}\frac{f(n)}{g(n)}=0,$ what do you think about the growth rates of f(n) and g(n)? f(n) = o(g(n))

Q: if $\lim_{n\to\infty}\frac{f(n)}{g(n)}=c,$ what do you think about the growth rates of f(n) and g(n) where c is a constant? f(n) = Θ(g(n))

Section 3.2 of MIT book

$$\lim_{n\to\infty}\frac{n^b}{a^n}=0$$ where a and b are real positive constants and $a > 1$. $\Rightarrow n^b = o(a^n)$

e.g. $$\lim_{n\to\infty}\frac{n^{10000}}{1.01^n}=0$$

$$\lim_{n\to\infty}\frac{\log n}{n^b}=0$$ where b is a real positive constant.

e.g. $$\lim_{n\to\infty}\frac{\log n}{n^{0.5}}=0; \quad \lim_{n\to\infty}\frac{\log n}{n^{0.001}}=0$$

In fact, $$\lim_{n\to\infty}\frac{\log^k n}{n^b}=0$$ for any k.

$$\lim_{n\to\infty}\frac{c}{\log n}=0; \quad \lim_{n\to\infty}\frac{c}{\log\log n}=0; \quad \lim_{n\to\infty}\frac{c}{\log\log\log n}=0$$

where c is any constant.

Q: Is it true that $(1.66666)^n = \Theta(2^n)$?

No, $(1.66666)^n$ has a slower growth rate.

Q: Is it true that $d^{1.000000001n} = \Theta(d^n)$ where d is a constant > 1?

No, d^n has a slower growth rate.

Q: Is it true that $2^{n+1} = \Theta(2^n)$?

Yes.

Q: Is it true that $2^{n^2+1} = \Theta(2^{n^2})$?

Yes.

Q: Is it true that $2^{n^2+n} = \Theta(2^{n^2})$?

No, the left hand side has a faster growth rate.

Q: $\dfrac{n}{\log\log n} = o(n).$

Yes, n has a faster growth rate.

Q: $n\log\log n = o(n^2).$

Yes, n^2 has a faster growth rate.

Some more about log (high school)

$$\log_a n = \frac{\log_b n}{\log_b a}$$ for any constants $a, b > 0$.

$$\log n^k = k \log n$$

$a^{\log b} = b^{\log a}$ Do you know how to prove it?

<u>Exercises:</u>

$\log_3 n = \omega(\log_{10000} n)$?

$2^{\log n} = \Theta(10n)$?

$8^{\log n} = \Theta(200n)$?