Analysis of Algorithms: time & space

Today..

- Why ‘Time Limit Exceeded’?
- How to know your code will work in time?
- Why ‘Memory Limit Exceeded’?
- How to know your code will not exceed memory?

Why ‘Time Limit Exceeded’?

- ..because your code was too slow than the intended solution.
- Several possible reasons:
  - Your code has a bug.
  - Your algorithm is too slow than the intended solution.
- We should find an sufficiently fast algorithm.

How to know your algorithm will work in time?

- Simple: implement your algorithm and run the code.
- However, it would be *better* if we can **estimate** the running time of the algorithm..
- Lots of things give influence on running time:
  - Hardware (CPU speed / # of cores / ..)
  - Compiler optimization

    1. We can’t control these..
    2. Differs by a factor

  - Input / Output size → Proportional to the input/output size
  - Your algorithm → **Depends on your algorithm!**
  - …

How to measure efficiency of an algorithm?

- From now, we assume the running time *only* depends on your algorithm!
- Of course, the running time of an algorithm depends on the ***input size*** $\boldsymbol{n}$.
- We cannot measure the exact running time, since we don't 'run' the algorithm.
- Instead, let's try to <u>count the number of steps </u>in the algorithm.

Counting the number of steps of an algorithm.

- Example problem:
  - Given an initival value a and common ratio r,
  - Calculate the sum of the first n terms of the geometric progression.
  - i.e. Calculate $a + ar + ar^2 + \cdots + ar^{n-1} = \sum_{i=1}^{n} ar^{i-1}$.
- For now, we are going to consider only algorithms, not on how to implement it.

Counting the number of steps of an algorithm. (cont.)

- Algorithm 1.

1. Make a variable sum with initial value 0, which stores the answer.
2. Consider all $i = 1,2,\cdots,n$:
   1. Make a variable x with initial value a.
   2. Repeat $i-1$ times:
      1. Multiply r to x.
   3. Add x to sum.

- So Algorithm 1 needs $\frac{(n+1)(n+2)}{2} + 1 = \frac{1}{2}n^2 + \frac{3}{2}n + 2$ steps.

# of steps

+1

$$\boxed{\left.\begin{array}{l} +1 \\ +(i-1) \\ +1 \end{array}\right\} +(i+1)}$$

$$\sum_{i=1}^{n}(i+1) = \frac{(n+1)(n+2)}{2}$$

Counting the number of steps of an algorithm. (cont.)

- Algorithm 2.

# of steps

1. Make a variable sum with initial value 0, which stores the answer. $+1$
2. Make a variable x with initial value a. $+1$
3. Consider all $i = 1,2,\cdots,n$:
   1. Add x to sum. $+1$
   2. Multiply x by r. $+1$

$$\boxed{\left.\begin{matrix}+1\\+1\end{matrix}\right\} +2} \quad \sum_{i=1}^{n} 2 = 2n$$

- So Algorithm 2 needs $1 + 1 + 2n = 2n + 2$ steps.

Counting the number of steps of an algorithm. (cont.)

- Algorithm 3:
- Use this formula:

$$a + ar^2 + \cdots + ar^{n-1} = \begin{cases} \dfrac{a(r^n - 1)}{r - 1}, & r \neq 1 \\ a \cdot n, & r = 1 \end{cases}$$

- ..because we can calculate r^n very fast. How?

Counting the number of steps of an algorithm. (cont.)

- Algorithm 3:
- How to calculate r^n much faster (for computers)

1. Calculate $r^1, r^2, r^4, \cdots, r^{2^k}$ first.
   - ..by $r^1 = r$, $r^2 = (r^1)^2$, $r^4 = (r^2)^2$, $\cdots$, $r^{2^k} = \left(r^{2^{k-1}}\right)^2$.
2. Write n by binary representation.
   - Ex) $19 = 2^4 + 2^1 + 2^0$
3. Multiply some precalculated values according to the binary representation.
   - Ex) $r^{19} = r^{2^4} \cdot r^{2^1} \cdot r^{2^0}$

Counting the number of steps of an algorithm. (cont.)

- Algorithm 3:
- How to calculate r^n much faster (for computers) (cont.)
  - If we were to calculate $r^{19} = r^{16+2+1}$,
  - The naïve method needs 18 multiplications:

$$r \times r = r^2, r^2 \times r = r^3, r^3 \times r = r^4, \cdots, r^{18} \times r = r^{19}$$

  - However, this algorithm requires only 6 multiplications.

$$r \times r = r^2, r^2 \times r^2 = r^4, r^4 \times r^4 = r^8, r^8 \times r^8 = r^{16}$$
$$r^1 \times r^2 = r^3, r^3 \times r^{16} = r^{19}$$

Counting the number of steps of an algorithm. (cont.)

- Algorithm 3:

1. Make a variable sum, which stores the answer.
2. If $r = 1$, the answer is $a \cdot n$. So store $\mathrm{a} \cdot n$ in sum.
3. Otherwise,
    1. Make an array $P[0..\lfloor \log n \rfloor]$, which will store the powers of r. ($P[i]$ stores r^{2^i})
    2. Store r in $P[0]$.
    3. Consider all $i = 1,2,\cdots,\lfloor \log_2 n \rfloor$:
        1. Store $P[i-1] \times P[i-1]$ in $P[i]$.

Counting the number of steps of an algorithm. (cont.)

- Algorithm 3: (cont.)
  4. Make a variable $prod$, which will store r^n.
  5. Consider all $i = 0,1,2,\cdots,\lfloor\log_2 n\rfloor$:
     1. If 2^i is in the binary representation, update $prod$ to $prod \cdot P[i]$.
  6. Store $\frac{a\cdot(prod-1)}{r-1}$ in sum.

Counting the number of steps of an algorithm. (cont.)

- Algorithm 3: **# of steps**

1. Make a variable sum, which stores the answer. $+1$
2. If $r = 1$, the answer is $a \cdot n$. So store a $\cdot\, n$ in sum. $+1$
3. Otherwise,
    1. Make an array $P[0..\lfloor\log_2 n\rfloor]$, which will store the powers of r. ($P[i]$ stores r^{2^i}) $+\lfloor\log_2 n\rfloor + 1$
    2. Store r in $P[0]$. $+1$
    3. Consider all $i = 1,2,\cdots,\lfloor\log_2 n\rfloor$:
        1. Store $P[i-1] \times P[i-1]$ in $P[i]$. $+\lfloor\log_2 n\rfloor$

Counting the number of steps of an algorithm. (cont.)

- Algorithm 3: (cont.)

# of steps

4. Make a variable $prod$, which will store r^n. — $+1$
5. Consider all $i = 0,1,2,\cdots,\lfloor \log_2 n \rfloor$:
    1. If 2^i is in the binary representation, update $prod$ to $prod \cdot P[i]$. — $+\lfloor \log_2 n \rfloor$
6. Store $\frac{a \cdot (prod-1)}{r-1}$ in sum. — $+1$

Counting the number of steps of an algorithm. (cont.)

- Algorithm 3: (cont.)

1. +1
2. If $r = 1$, …: +1
3. Otherwise,
   1. $+\lfloor \log_2 n \rfloor + 1$
   2. $+1$
   3. $+\lfloor \log_2 n \rfloor$
   4. $+1$
   5. $+\lfloor \log_2 n \rfloor$
   6. $+1$

So, Algorithm 3 needs

- 2 steps if $r = 1$,
- $6 + 3\lfloor \log_2 n \rfloor$ steps if $r \neq 1$.

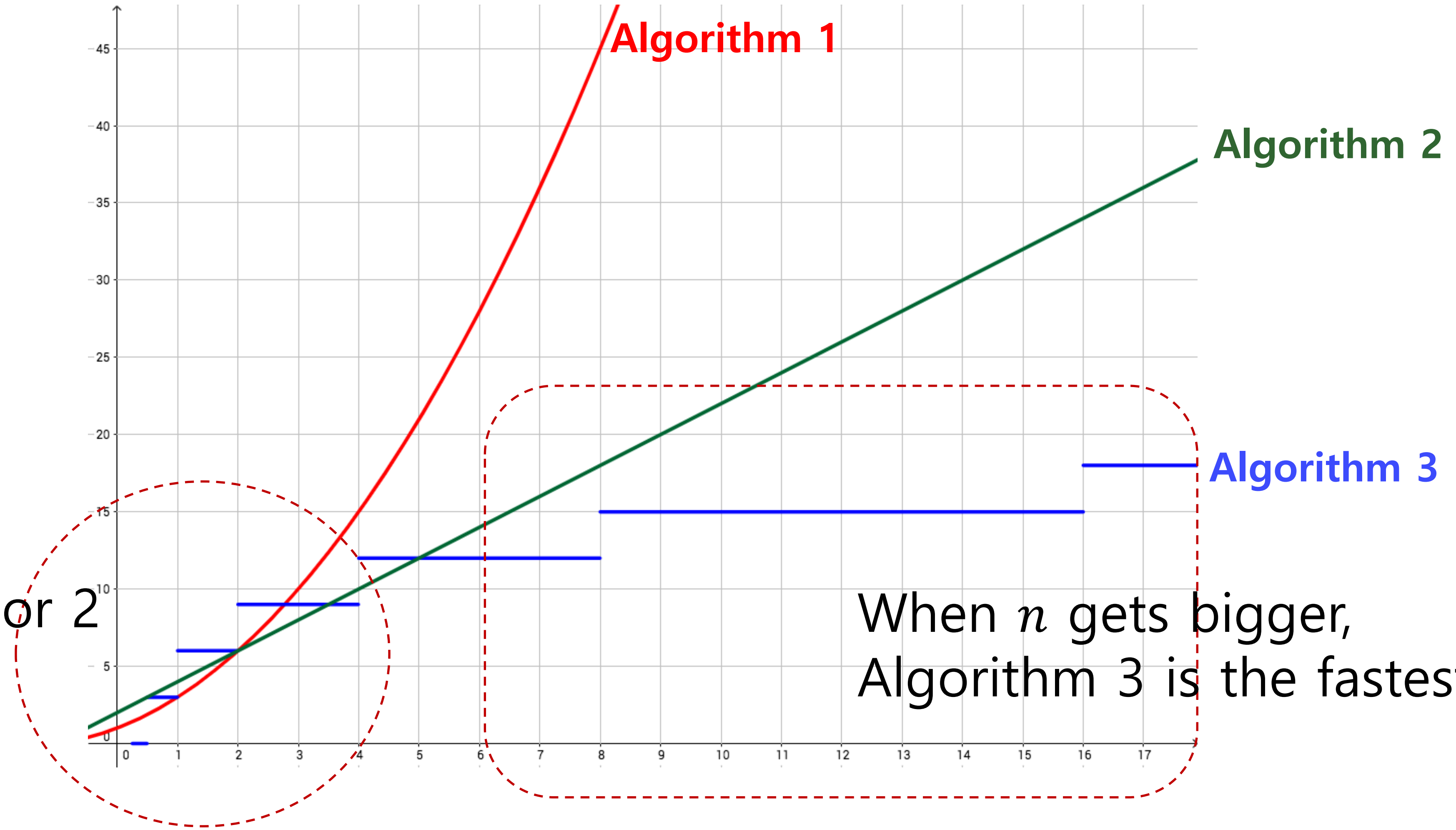
Counting the number of steps of an algorithm. (cont.)

- # of steps of three given algorithms:
  - Algorithm 1: $\frac{1}{2}n^2 + \frac{3}{2}n + 2$ steps
  - Algorithm 2: $2n + 2$ steps
  - Algorithm 3: 2 steps if $r = 1$, $6 + 3\lfloor \log_2 n \rfloor$ steps if $r \neq 1$.
  - Like this, sometimes the *number of steps* changes by the condition of the input.
  - We need to assure that the algorithm will run in time for ***every possible input***, so we usually take the ***worst*** case. (i.e. slower, with more steps)

Counting the number of steps of an algorithm. (cont.)

- # of steps of three given algorithms:
  - Algorithm 1: $\frac{1}{2}n^2 + \frac{3}{2}n + 2$ steps
  - Algorithm 2: $2n + 2$ steps
  - Algorithm 3: $6 + 3\lfloor \log_2 n \rfloor$ steps
- The # of steps depends on n, the input size.
- To compare, let's plot a graph.

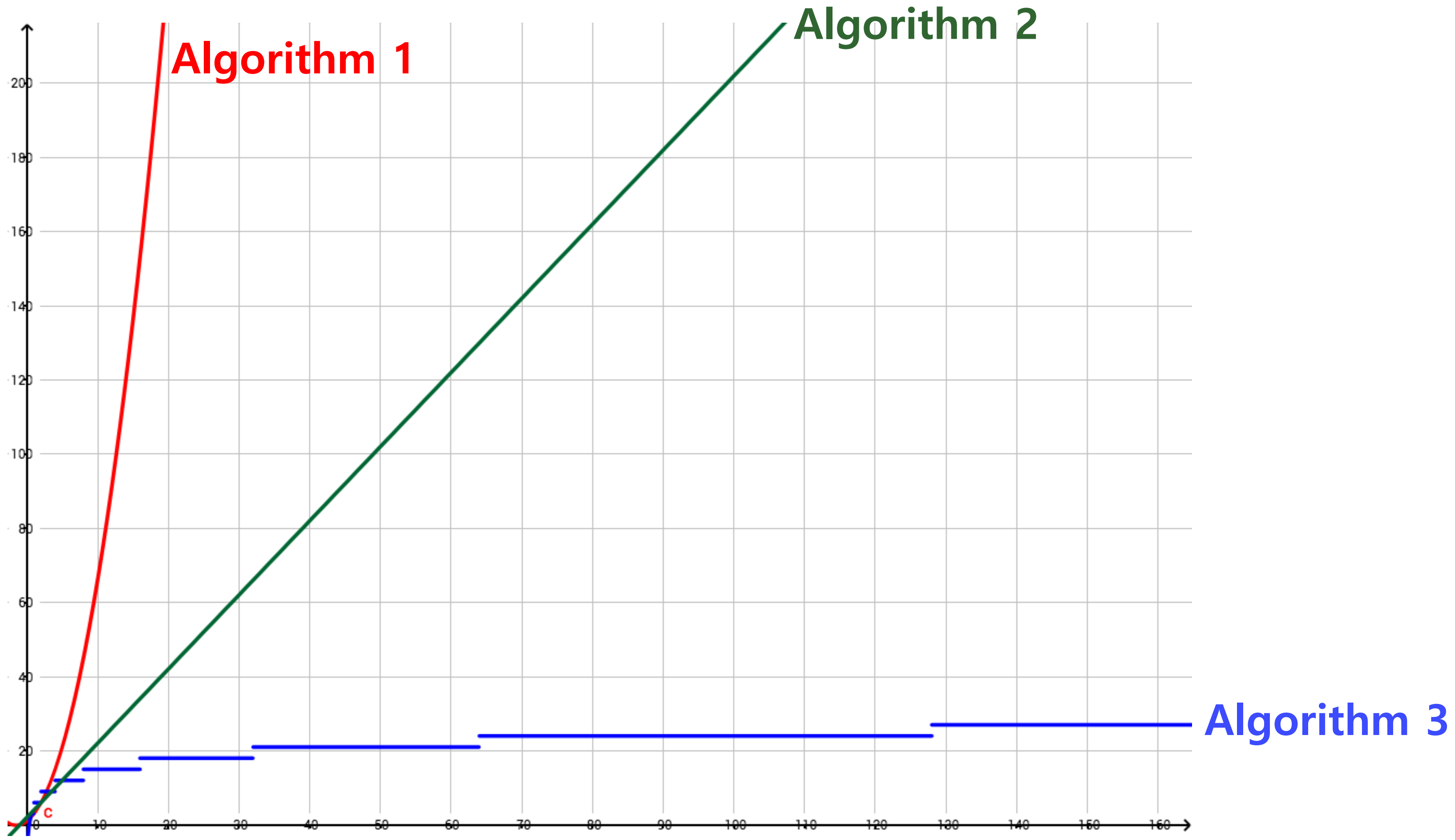
Counting the number of steps of an algorithm. (cont.)



For small n,
Algorithm 1 or 2
is faster.

When n gets bigger,
Algorithm 3 is the fastest.

Counting the number of steps of an algorithm. (cont.)



Is 'Step' analysis good enough?

- The 'step' analysis I've given to you have flaws:

1. We didn't define what a "step" is.
2. We didn't count some operations like "Consider all $i = 1,2,\cdots,n$", because we don't know what "step" means.
3. We assumed all 'steps' take the same time.
   - Obviously, calculating $\frac{a\cdot(prod-1)}{r-1}$ takes more time than calculating $x \times r$.

Is 'Step' analysis good enough? (cont.)

- However, we could use this analysis because..

1. This is not a 'rule' for exact running time.
2. This can only *estimate* the running time *asymptotically*.
   - In other words, **we can be certain** that, for **large $\boldsymbol{n}$**, Algorithm 3 is the fastest among given algorithms.
3. So we don't need to *define* the meaning of 'step', but we can introduce some criteria:
   - Good: Basic C++ operations like addition, multiplication, division, substitution, comparison, …
   - Bad: Comparing two strings, computing the sum a sequence of size n, Sorting a sequence, …

Step analysis is too difficult

- However, still calculating the # of steps is ambiguous, boring and tired.
- So we are going to simplify by..

1. Only considering the terms ***which increases fastest***.
2. Removing all coefficients

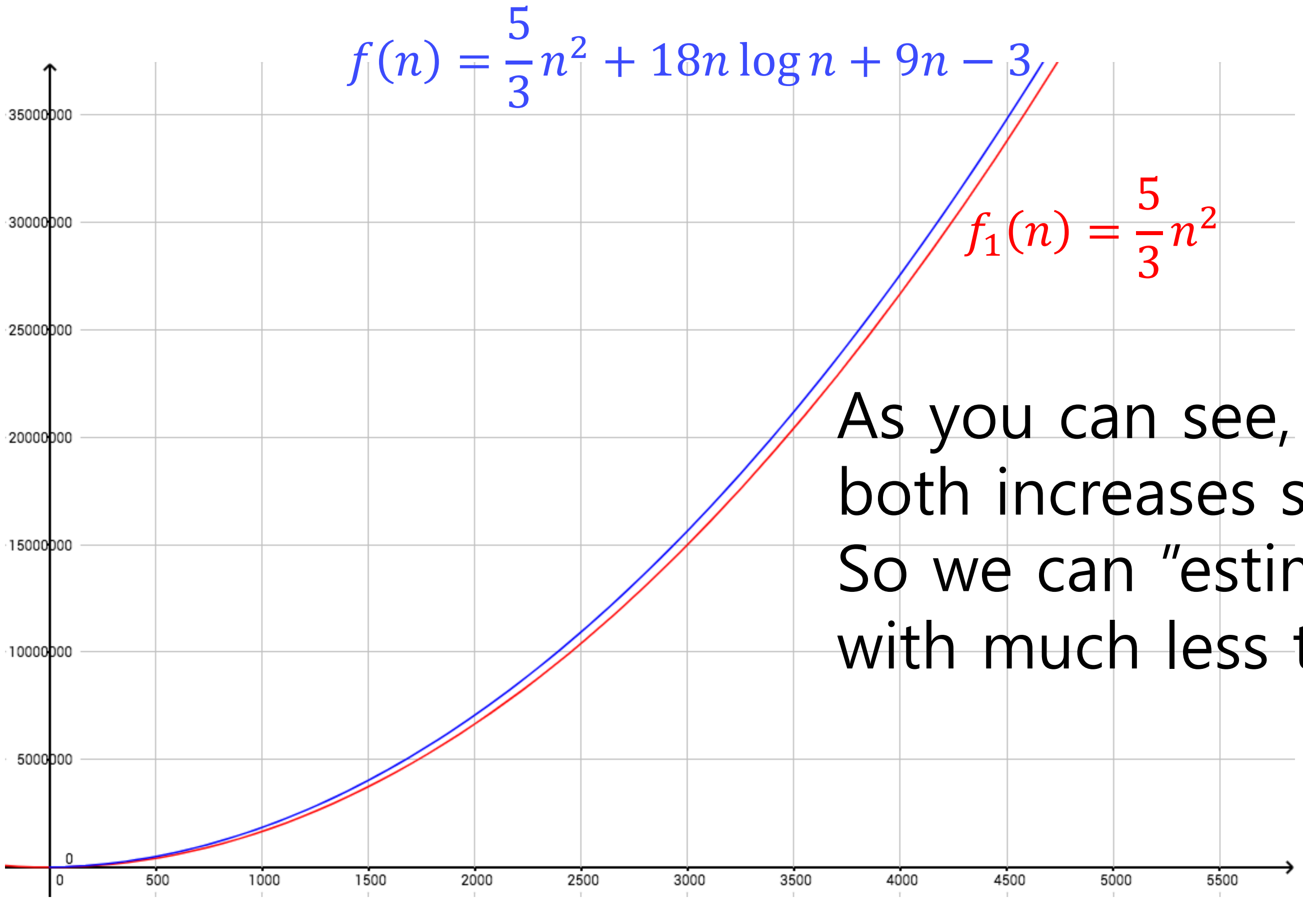
Simplifying 'step' analysis

1. Only considering the terms ***which increases fastest***.

- Example: $f(n) = \frac{5}{3}n^2 + 18n\log n + 9n - 3$
- By inspection we know $\frac{5}{3}n^2$ increases most rapidly. So we just write $f_1(n) = \frac{5}{3}n^2$.
- We can do this because for ***large*** $\boldsymbol{n}$, even without smaller terms, $f(n) \approx f_1(n)$.
  - Of course the difference gets bigger as n increases, but..

Simplifying 'step' analysis (cont.)

1. Only considering the terms ***which increases fastest***.



As you can see,
both increases similarly,
So we can "estimate" f
with much less terms.

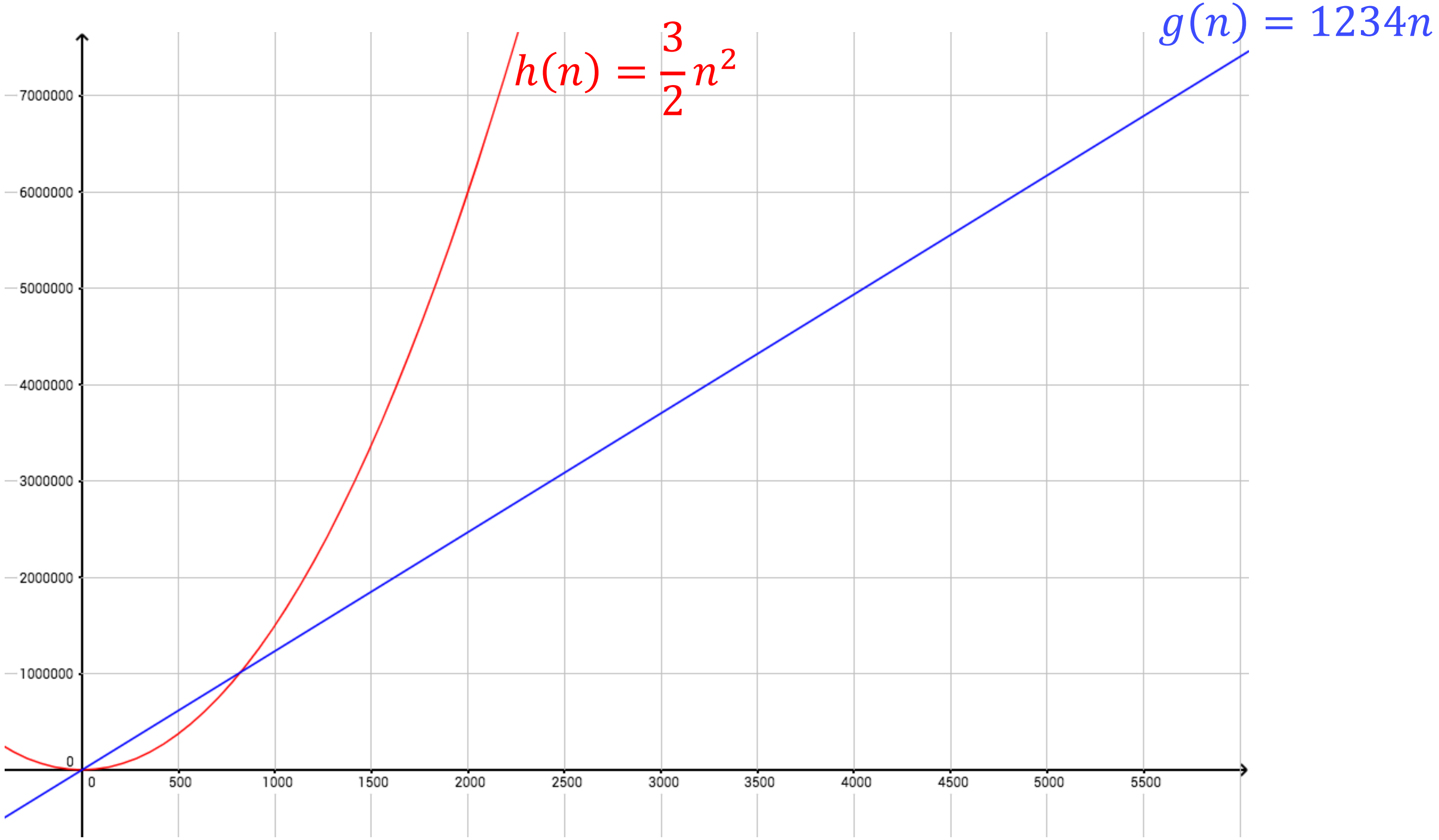
Simplifying 'step' analysis (cont.)

2. Removing all coefficients

- We don't know how much time each step takes.
- Even if we know that, coefficients are not *that* important..
- Example: $g(n) = 1234n$, $h(n) = \frac{3}{2}n^2$.
  - Because of big coefficient, g seems quite good. But..

Simplifying 'step' analysis (cont.)

2. Removing all coefficients



Simplifying 'step' analysis (cont.)

2. Removing all coefficients

- Eventually $1234n$ is faster than $\frac{3}{2}n^2$.
- So why don't we just think n and n^2 instead? Now we know which is faster by inspection.
- ***HOWEVER***, this example implies that for some small inputs, $\frac{3}{2}n^2$ ***can be faster*** than $1234n$. So we should be more careful when we handle *small inputs*.

Big-oh notation

- This simplification can be written mathematically:

DEFINITION Let $f(x)$ and $g(x)$ be positive for x sufficiently large. Then f is **of at most the order of** g as $x \to \infty$ if there is a positive integer M for which

$$\frac{f(x)}{g(x)} \leq M,$$

for x sufficiently large. We indicate this by writing $\boldsymbol{f = O(g)}$ ("f is big-oh of g").

Big-oh notation (cont.)

- Example: $\frac{5}{3}n^2 + 18n\log n + 9n - 3 = O(n^2)$
- ..because: the limit

$$\lim_{n\to\infty} \frac{\frac{5}{3}n^2 + 18n\log n + 9n - 3}{n^2} = \frac{5}{3}$$

- exists, we can let $M = 2$.
- From this “limit” approach, we can only consider the term that has maximum degree.

Big-oh notation (cont.)

- This gives the estimation of the ***upper bound*** of the running time.
- Example: $\frac{7}{3}n + \log n - 5 = O(n^2)$. n^2 is also a possible *upper bound*.

DEFINITION Let $f(x)$ and $g(x)$ be positive for x sufficiently large. Then f is **of at most the order of** g as $x \to \infty$ if there is a positive integer M for which

$$\frac{f(x)}{g(x)} \leq M,$$

for x sufficiently large. We indicate this by writing $\boldsymbol{f = O(g)}$ ("f is big-oh of g").

Big-oh notation (cont.)

- *Upper bound* should be as **low** as possible.
- So even though $\frac{7}{3}n + \log n - 5 = O(n^2)$, it is better to denote

$$\frac{7}{3}n + \log n - 5 = O(n)$$

- .. Since this is also true.

Time complexity

- By these methods, we can write the *time complexity* of an algorithm using the *Big-Oh notation*.

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

- We can compare the efficiency of the algorithm by comparing $g(n)$.

Comparing time complexities

- These are time complexities you will frequently encounter:
- $O(1) < O(\log n) \ll O(\sqrt{n}) < O(n) < O(n \log n) < O(n\sqrt{n}) < O(n^2) < O(n^3) < O(n^4) \ll O(2^n) < O(n \cdot 2^n) \lll O(n!) \ll O(n^n) < \cdots$
- $\underline{\boldsymbol{O(1)}}$: Running time is always the same regardless of input.
- $\underline{\boldsymbol{O(2^n)}}$: Running time is exponential to $\underline{n}$.
- We don't write the base of the logarithm, since we can convert the base by multiplying a constant: $\log_a x = \frac{1}{\log a} \log x$.

Comparing time complexities (cont.)

- Go back to the geometric progression:

1. Algorithm 1: $\frac{1}{2}n^2 + \frac{3}{2}n + 2 = O(n^2)$
2. Algorithm 2: $2n + 2 = O(n)$
3. Algorithm 3: $6 + 3\lfloor \log_2 n \rfloor = O(\log n)$

- So we can conclude Algorithm 3 is the best.

How to know the time complexity?

- Only the *fastest increasing term* is important.
- As you've seen in examples, ***loops*** gives biggest influence on running time.
- If there are some ***nested loops***, the deepest loop gives biggest influence. If there are many nested loops, consider the *most deepest* one.
- Try to focus on these "loops" on your algorithm, and find the time complexity *approximately*.

How to know the time complexity? (cont.)

- Example: Algorithm 1 again

# of steps

1. Make a variable sum with initial value 0, which stores the answer. $+1$
2. Consider all $i = 1,2,\cdots,n$:
   1. Make a variable x with initial value a. $+1$
   2. Repeat $i-1$ times: $+(i-1)$
      1. Multiply r to x.
   3. Add x to sum. $+1$

$$\left.\begin{matrix} +1 \\ +(i-1) \\ +1 \end{matrix}\right\} +(i+1)$$

$$\sum_{i=1}^{n}(i+1) = \frac{(n+1)(n+2)}{2}$$

- So Algorithm 1 needs $\frac{(n+1)(n+2)}{2} + 1 = \frac{1}{2}n^2 + \frac{3}{2}n + 2$ steps.

How to know the time complexity? (cont.)

- Example: Algorithm 1 again

1. Make a variable sum with initial value 0, which stores the answer.
2. Consider all $i = 1,2,\cdots,n$:
    1. Make a variable x with initial value a.
    2. Repeat $i-1$ times:
        1. Multiply r to x.
    3. Add x to sum.

Nested loop!
We know $\sum_{i=1}^{n}(i-1) = O(n^2)$.

The whole algorithm runs in $O(n^2)$.

- So Algorithm 1 needs $\frac{(n+1)(n+2)}{2} + 1 = \frac{1}{2}n^2 + \frac{3}{2}n + 2$ steps.

Then, how to estimate whether my program will run in time?

- Suppose you've got the time complexity in Big-Oh notation.
- **1e8 Rule** (not formal, made about 10 years ago):
  - Suppose 1 sec $\approx 10^8$(100 million) in Big-Oh notation.
- $O(1)$ algorithm will always work regardless of input size.
- $O(n)$ algorithm will work in 1 sec if $n \leq 10^8$.
- $O(n \log n)$ algorithm will work in 1 sec if $n \leq 4523071$.
- $O(n^2)$ algorithm will work in 1 sec if $n \leq 10^4$.

Then, how to estimate whether my program will run in time? (cont.)

- Substitute the maximum n into the time complexity, and check whether it exceeds 10^8.
  - There is a problem with constraint $n \leq 5{,}000$.
  - You've come up with an $O(n^2)$ algorithm.
  - $5{,}000^2 = 25{,}000{,}000 \ll 10^8$.
  - So you are confident that your algorithm will work in time.

Then, how to estimate whether my program will run in time? (cont.)

Time complexity	Max n to work in 1 sec.
$O(1)$	No condition
$O(\log n)$	2^{10^8}
$O(\sqrt{n})$	10^{16}
$O(n)$	10^8
$O(n \log n)$	4523071
$O(n^2)$	$10{,}000$
$O(n^3)$	464
$O(n^4)$	100
$O(2^n)$	26
$O(n!)$	11
$O(n^n)$	8

Then, how to estimate whether my program will run in time? (cont.)

- **CAUTION**: The 1e8 rule is not a "rule of thumb". You should not underrate or overrate your algorithm.
  - Example: There is a problem with constraint $n \leq 20{,}000$.
  - You've come up with a $O(n^2)$ solution.
  - $20{,}000^2 = 4 \times 10^8 > 10^8$, so you think it might won't work in 1 sec.
  - However, it turned out that your algorithm uses only simple operations, and it actually worked!
  - This is because the "1e8 rule" doesn't consider the cost of operations, CPU, …

Then, how to estimate whether my program will run in time? (cont.)

- You should use this rule like this:
  - Example: There is a problem with constraint $n \leq 1{,}000{,}000$.
  - You've come up with a $O(n^3)$ solution.
  - $1{,}000{,}000^3 = 10^{18} \ggg 10^6$, so this algorithm won't work.
  - We can be sure the algorithm won't work because factor like 10^{12} is very hard to reach by optimizations.

Why ‘Memory Limit Exceeded’?

- ..because your code needed much more memory than the given limit (256MiB).
- Several possible reasons:
  - You made lots of arrays.
  - Your code has a bug. (Allocated memory in an infinite loop?)
- However, generally, we can calculate the memory used.

How to know the memory usage?

- If you implemented your algorithm into a C++ code,
- ..there is an operator called **`sizeof`**!
- **`sizeof`**`(A)` gives the number of bytes used by `A`.

```
int a[5050][1203];
int b[5050];
…
printf("%lu\n", sizeof(a) + sizeof(b));
// Result: 24320800
```

- You can add all sizes of arrays that you used, and see whether it exceeds 268435456 (256×1024×1024) bytes.

How to know the memory usage? (cont.)

- However, you cannot use this method if:
  - You didn’t code
  - Your algorithm uses pointers.
- Then, you need to calculate your memory. Actually it is easy, and sometimes you can *explicitly* (not *approximately*!) “calculate” the memory.

How to know the memory usage? (cont.)

- Example: Suppose you need two $n \times n$ `int`-type arrays. How much bytes are needed?
  - Since one `int` takes 4 bytes, an $n \times n$ `int`-type array takes $4n^2$ bytes.
  - We need $8n^2$ bytes for two $n \times n$ `int`-type arrays.
  - If $n = 1000$, we need 8,000,000 bytes.
- If you didn't know one `int` takes 4 bytes, you can print `sizeof(int)` and check. You can do the same for other types.

```
printf("%lu\n", sizeof(int));
```