Data Structures

4. Algorithm Analysis

4. Algorithm Analysis

1. Performance Analysis 성능분석
2. Space Complexity
3. Time Complexity
4. Asymptotic Notation

Performance Analysis

- Ideal Criteria (이상적인) 소프트웨어 개발단계
  - Does a program meet the original requirement of the task?
  - Does it work properly?
  - Does it effectively use functions to perform a task?
- Realistic Criteria (현실적인)
  - the amount of memory space that a program needs to complete the execution (Space complexity) 디스크, 메모리, …
    낮을수록 좋은거
  - the amount of computational time for execution (Time complexity)

Space Complexity

- Fixed space requirements (S_c) 고정요구공간 : 명령어저장, 단순변수, ...
  - Memory space for instructions, simple variable, fixed-size structured variable, constants

- Variable space requirement (S_v) 가변요구공간
  - Memory space can be determined at run time because of array passing, or recursion

실행을해봐야앎: 배열, ..
예측불가, 주의

- Total Space Complexity

$$S = S_c + S_v$$

Space Complexity

- Simple variables

```
float abc (float a, float b, float c)
{
    return a + b + b * c + (a + b - c) / (a + b) + 4.00;
}
```

- input : three simple variables (a, b, c)
- ouput : a simple variable (float type)

- variable space requirements
  - $S_v(abc)$ = 0

  가변요구공간 X

Space Complexity

- Sum

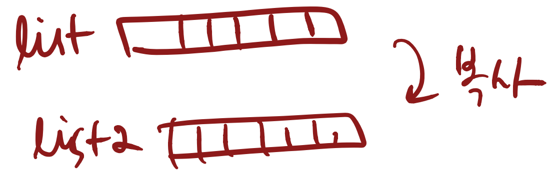
배열

```
float Sum (float list[], int n)
{
    int i;
    float total = 0;

    for (i = 0; i < n; i++)
        total = total + list[i];

    return total;
}
```

합계

- input : an array and an integer
- output : a simple float variable
    - Space complexity depends on array passing method

Space Complexity

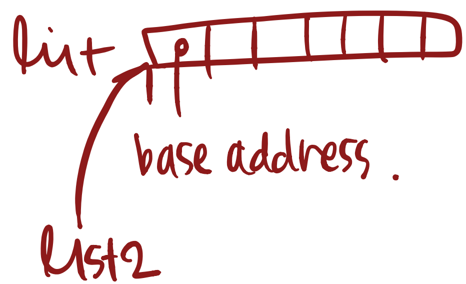


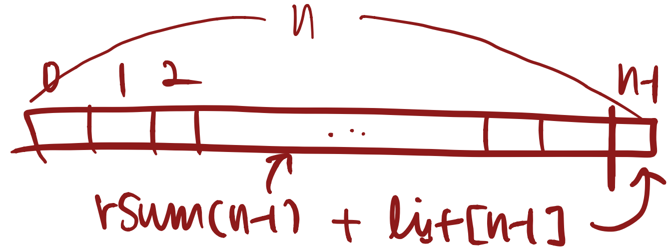
- Call by Value
  - Array elements are copied to function
  - Additional memory space is required in proportional to array size
    - $S_v(\text{sum})$ = n   // n is array size
  - ex) Pascal (옛날)

가변요구공간o

- Call by Reference
  - The base address of array is passed to function
  - No additional memory space required
    - $S_v(\text{sum})$ = 0
  - ex) C

가변요구공간x





Space Complexity

- Sum (recursive) context(정보)
  - At each function call, the followings must be saved
    - Local variables : list, n
    - Return address (the next instruction to resume)
    - n times function calls : rsum(n-1) …. rsum(0) → n번 불림
    - Space Complexity = $\boxed{12n}$ // 4 bytes each ? 외워야..

재귀적인 sum

```
float rsum (float list[], int n)
{
    if (n)
        return rsum (list, n-1 ) + list[n - 1];
    else
        return 0;
}
```

Time Complexity

- Time complexity (T)
  - amount of time taken by an algorithm to complete
  - T = Compile time time (T_c) + execution time (T_e)

    1번만 (가변성 x)　　실행시간

  - Execution time depends on computing environment
  - Instruction steps is more objective, and less accurate though

실행시간을 측정해보자

→ 모두 다름

물리적인 시간 : 객관성↓

명령어의 개수를 세는게 낫다.

└ 경우마다 다를수있지만 같다고 가정하고 (step count)

Time Complexity

- Step count table
  - $\boxed{\text{Steps}}$
    - Instructions per line
    - Function header, variable declaration : no count for steps
    - ex) a = 1; b = 3; // two steps in a line
  - Frequency
    - the number of execution times for each instruction
    - ex) for, while loops
  - Total steps
    - steps * frequency per line
    - Add up all total steps of each line

Time Complexity

- Step Count Table (Iterative Sum)

Statement	steps	frequency	total steps
`float sum (float list[], int n) {` 함수header	0 제외	0	0
`int i;` 변수선언	0 제외.	0	0
`float temp = 0;`	1	1	1
`for (i = 0; i < n; i++)` 반복문header 하나로	1	1 n+1	1 n+1
`temp += list[i];`	1	n	n
`return temp;`	1	1	1
`}`	0	0	0
Total step counts			2n+3

n: 배열의 원소의 수

Time Complexity

- Step Count Table (Recursive Sum)

Statement	steps	frequency	total steps
float rsum (float list[], int n) {	0	0	0
if (n) n~0 이므로 n+1	1	n+1	n+1
return rsum (list, n - 1) + list[n - 1];	1	n	n
return 0; → if가 거짓일때 1 ↳ if가 참일때 n	1	1	1
}	0	0	0
Total step counts			$\boxed{2n + 2}$

n이 커진다면 1차이는 별의미X
둘다 2n으로 생각

Time Complexity

- Step Count Table (Matrix Addition)

Statement	steps	frequency	total steps
void add () {	0	0	0
행 열 int i, j;	0	참일때만 0	0
for (i = 0; i < rows; i++)	1	rows + 1	rows + 1
for (j = 0; j < cols; j++)	1	rows * (cols+1)	rows * (cols +1)
c[i][j] = a[i][j] + b[i][j];	1	rows * cols	rows * cols
}	0	0	0
Total step counts			2rows * cols + 2rows + 1

Big O Notation

더 빨리 증가 (dominant한)
주도적인

n: input의 양

- Time complexity
  - $T = n^2 + 2n \Rightarrow O(\underline{\ n^2\ })$

점근적으로

  - To approximate a time complexity asymptotically where n is very large
  - implies how much time an algorithm takes to run if n increases
  - A standard for evaluating the algorithm performance

$n^2 + 2n$ vs. $100n$

꼭 그런건 X
입력이 99이하일 때는 $n^2+2n < 100n$.

- Assume that we have two different algorithms solving for a problem
- Which algorithm is better ?

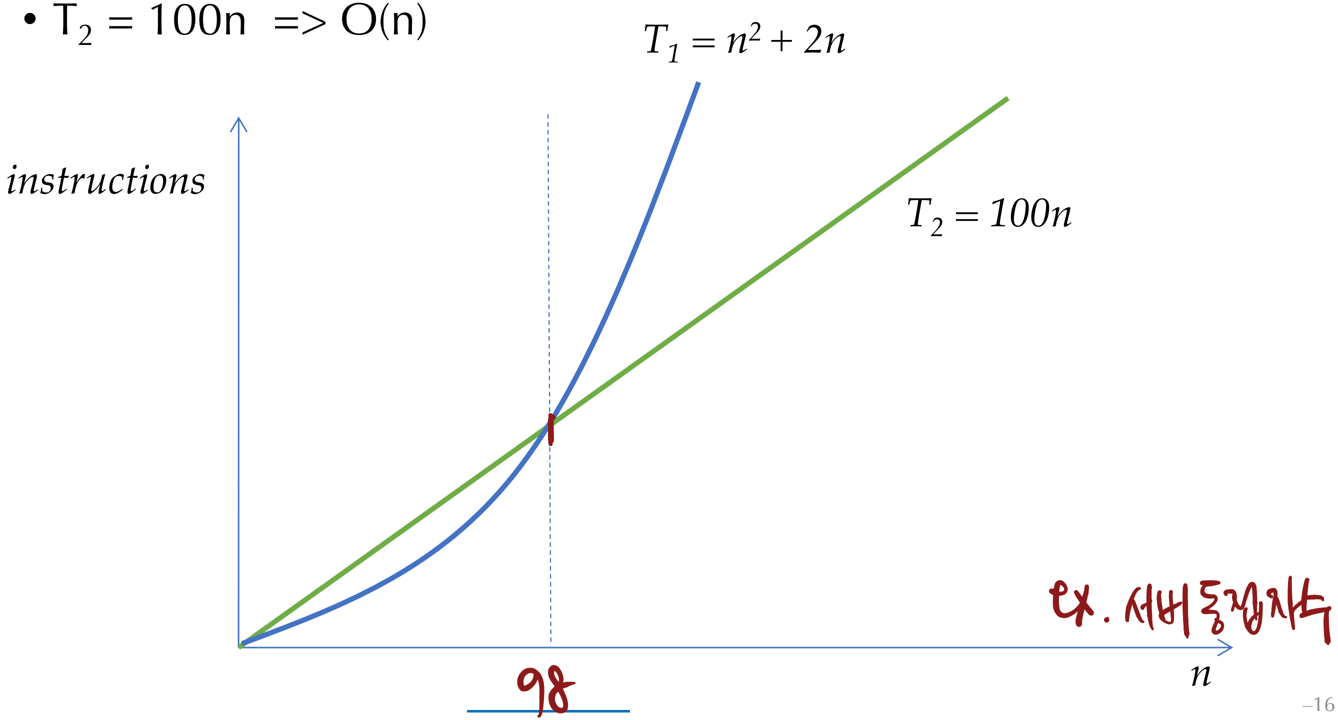
- $T_1 = n^2 + 2n$ and $T_2 = 100n$
  - If $n \leq 98$, then $n^2 + 2n \leq 100n$
  - If $n > 98$, then $n^2 + 2n > 100n$

  - if $n = 100{,}000$, then $100{,}000^2 + 100{,}000 > 100{,}000$

  - As n increases
    - T_1 takes much more time than T2
    - we should choose T_2
    - The smaller , the better

Asymptotic Notation

- Asymptotically notated in Big Oh
  - $T_1 = n^2 + 2n \Rightarrow O(n^2)$
  - $T_2 = 100n \Rightarrow O(n)$



Big-Oh Notation

찾으면됨

- $f(n) \in O(g(n))$ iff there exist positive constants (integer) c and n_0 that satisfy $f(n) \le c \cdot g(n)$ for all $n \ge n_0$

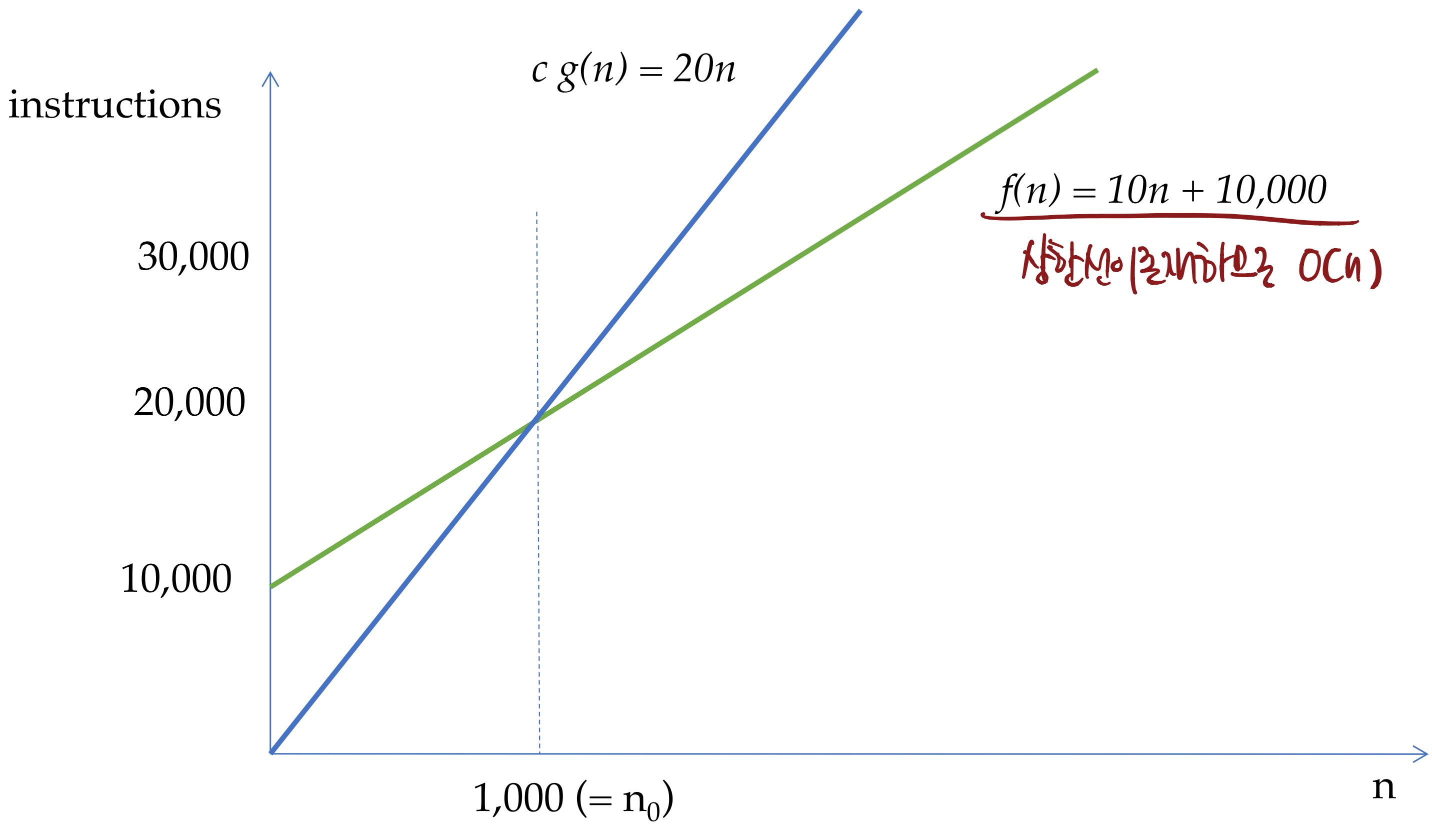
- $c \cdot g(n)$ is upperbound of $f(n)$ (상한선)
- $f(n)$ takes less time than $c \cdot g(n)$

- ex) prove $\boxed{f(n) = 10n + 10{,}000 \in O(n)}$
  - choose c and n_0 with 20 and 1,000
  - $f(n) \in O(n)$ as $f(n) \le$ 20 n for all $n \ge$ 1000

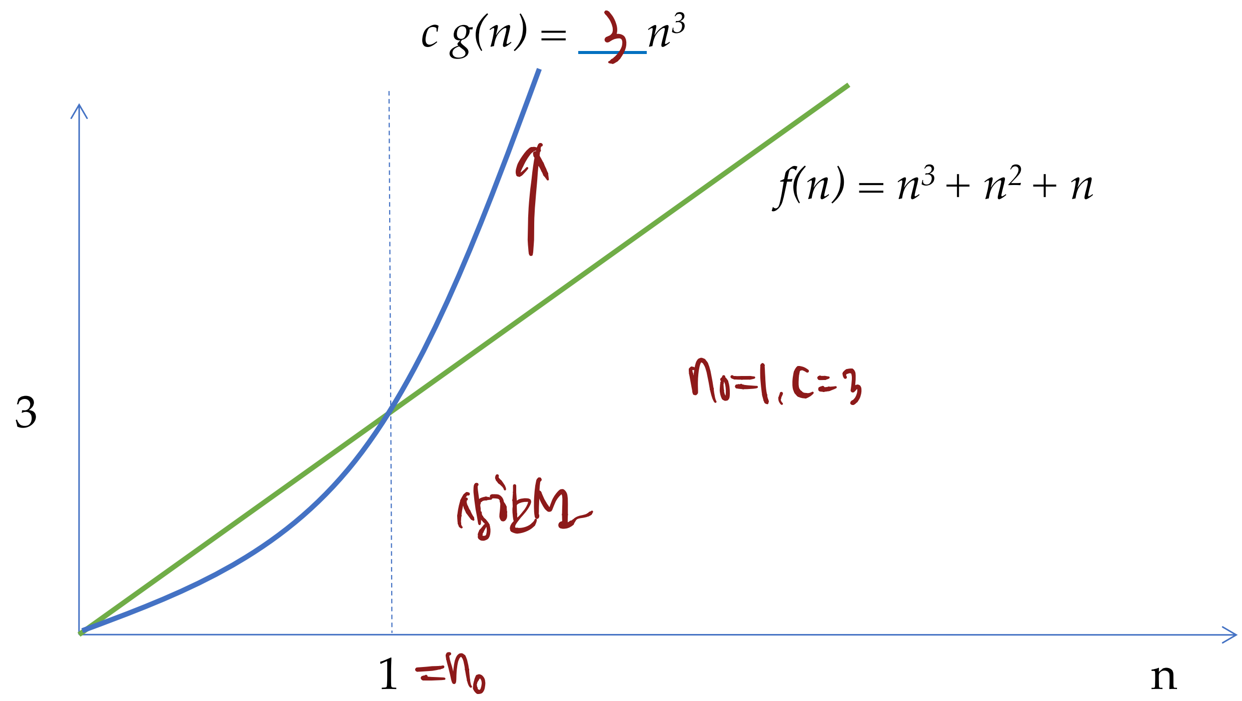
찾는방법은특별히없음
경험적으로

- ex) prove $\boxed{f(n) = n^3 + n^2 + n \in O(n^3)}$
  - choose c and n_0 with 3 and 1
  - $f(n) \in O(n^3)$ as $f(n) \le$ 3 n^3 for all $n \ge$ 1

Asymptotic Notation



Big-Oh Notation



Why f(n) <= cg(n) C를구하는적절한방법.

- Because g(n) grows faster than each term of f(n), c times g(n) increases always faster than f(n)
- Choose c by considering the coefficient of a dominant term and number of terms of f(n)

- ex)
  - $f(n) = n^2 + n \le n^2 + \underline{\;n^2\;} = 2n^2 \le 2g(n) = O(n^2)$
  - $f(n) = 2n^2 + n \le 2n^2 + \underline{\;n^2\;} = 3n^2 \le 3g(n) = O(n^2)$

이걸최고차항으로대체

Big-Oh Notation

- $3n + 2 \in O(n)$ as $3n + 2 \leq \underline{4n}$ for *all* $n \geq 2$
- $10n^2 + 4n + 2 \in O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for $n \geq 5$
- $6 \cdot 2^n + n^2 \in O(\underline{\;2^n\;})$ as $6 \cdot 2^n + n^2 \leq 7 \cdot 2^n$ for $n \geq 4$
- $3n + 3 \in O(n^2)$ as $3n + 3 \leq \underline{3n^2}$ for $n \geq 2$
- $\log n + n + 3 \in O(\underline{\;n\;})$
- $n \log n + n \in O(\underline{\;n\log n\;})$

어떻게 더 정확한가? 더 가까운거 O( )가 바람직

- $3n + 2 \notin \boxed{O(1)}$, $10n^2 + 4n + 2 \notin O(n)$
  - Because there is no such c and n_0 satisfying Big-Oh definition

constant

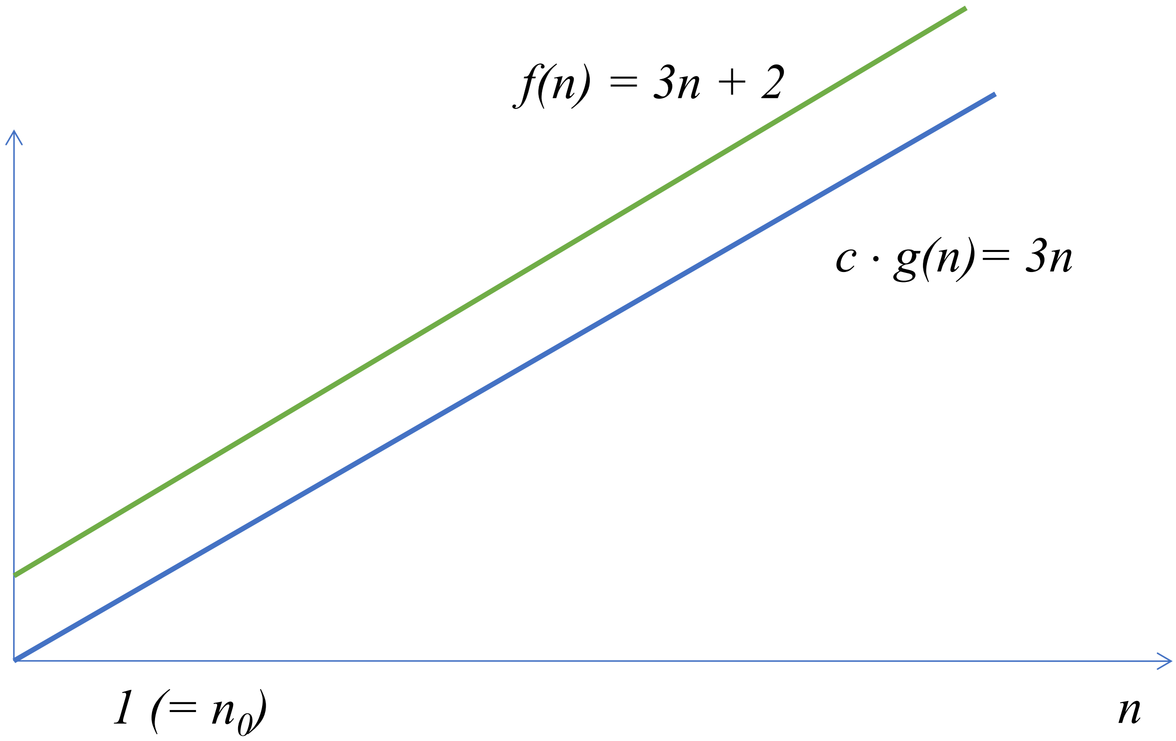
(k개의 명령어만 수행하면 되므로

Big-Omega Notation

- $f(n) \in \Omega(g(n))$
  - iff there exist positive constants c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$
  - $c \cdot g(n)$ is the lower bound of f(n)
  - F(n) takes more time than $c \cdot g(n)$

- ex)
  - $f(n) = 3n + 2 \in \Omega(n)$ as $f(n) \geq 3n$ for $n \geq 1$
  - $f(n) = 10n^2 + 4n + 2 \in \Omega(n^2)$ as $f(n) \geq n^2$ for $n \geq 1$

Big-Omega Notation

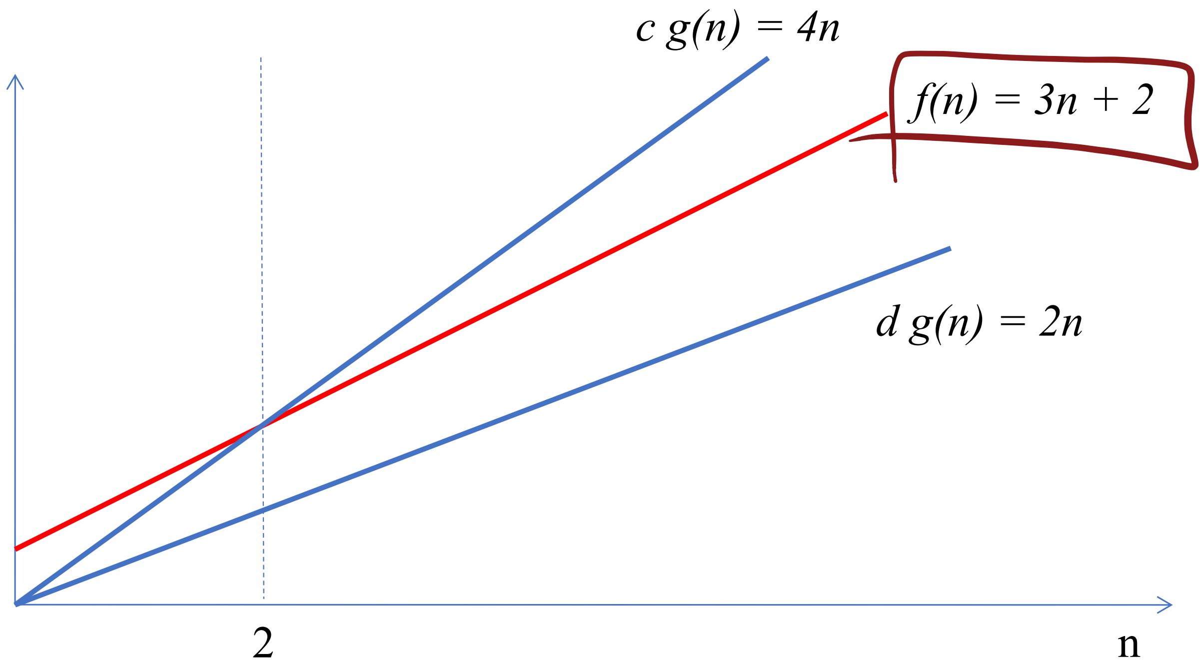


Big-Theta Notation

- $f(n) \in \Theta(g(n))$ iff $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
  - iff there exist positive constants c, d, and n_0 , satisfying $d \cdot g(n) \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$
  - f(n) takes the time _more_ than lower bound, and _less_ than upper bound

- $f(n) = 3n + 2 \in \Theta(n)$ as $3n \le f(n)$ and $f(n) \le 4n$ for all $n \ge 2$
- $10n^2 + 4n + 2 \in \Theta(n^2)$

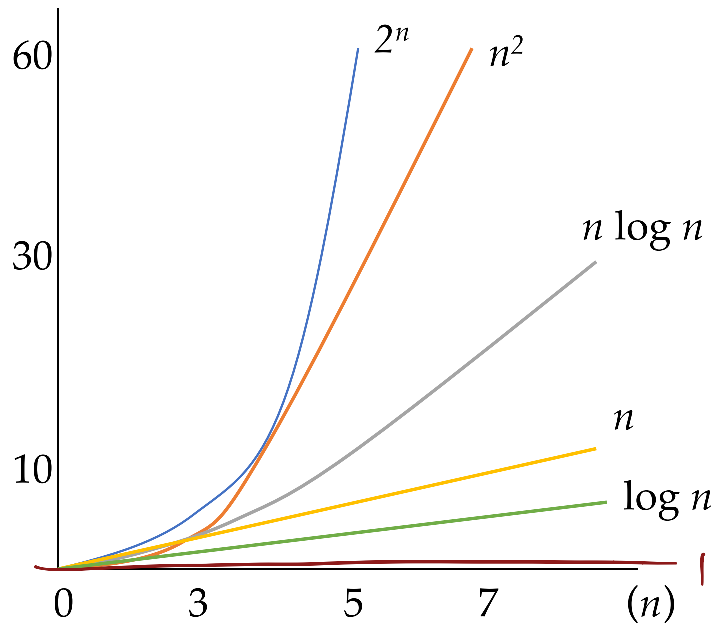
Big-Theta Notation



Time complexity Class

- $O(1)$ = constant (fast)
- $O(\log n)$ = logarithm
- $O(n)$ = linear
- $O(n \log n)$ = log linear
- $O(n^2)$ = quadratic
- $O(n^3)$ = cubic
- $O(2^n)$ = exponential
- $O(n!)$ = factorial (slow)

하노이탑



Practical Complexity

- $O(1) \sim O(n^2)$
  => tractable; useful when n is large
- $O(2^n) \sim O(n!)$
  => intractable, useful when n is very small

Instance characteristic *n*							
time	name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
log n	Logarithm	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
n log n	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
n!	Factorial	1	2	24	40326	20922789888000	$26313*10^{33}$

Practical Complexity

- Ex) 1 bps computer (= 1 billion inst./sec (10^9 /sec))

- If a program runs an algorithm that needs 2^n steps for execution
    - n = 40 → # of steps = $(2^{10})^4 = (1.024 * 10^3)^4 = 1100 * 10^9$
        - Total execution time is 1,100 sec/60 = 18.3 min
    - n = 50 → 13 days
    - n = 60 → 310.56 years
    - n =100 → 4×10^{13} years

- If an algorithm needs n^{10} steps
    - n = 10 → 10 sec
    - n = 100 → 3,171 years