Lecture 2:
Algorithm analysis and Computational Complexity

Algorithm

Jeong-Hun Kim

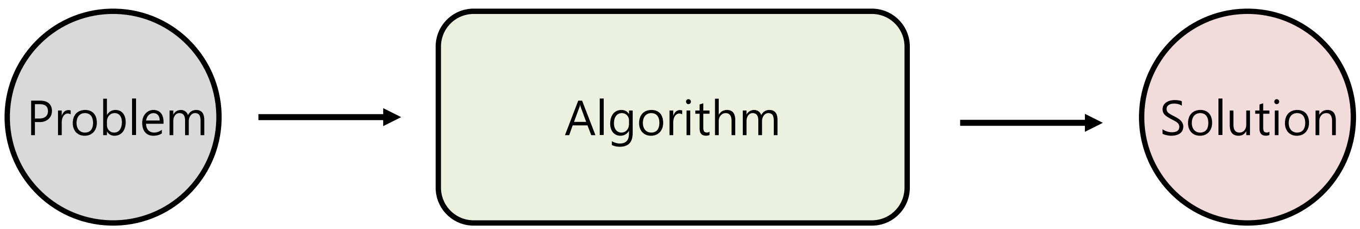
Table of Contents

Part 1

PRELIMINARIES

Preliminaries

❖ What is an algorithm?



Preliminaries

❖ What is a problem?

- A task that needs to be processed to achieve a goal

❖ Example 1) Sort the list S of n integers in monotone increasing order.

- Solution:

For $x, y \in \mathbb{R}$, if $x \leq y$, then when $f(x) \leq f(y)$, it is denoted an increasing function, and f is monotonically increasing.

❖ Example 2) Determine whether the integer x exists in the list S of n integers. If it exists, return its index, and if not, return -1.

- Solution:

Preliminaries

- What are parameters?
  - Unassigned variables mentioned in the problem

- Example 1) Sort the list S of n integers in monotone increasing order.
  - Parameters: S, n

- Example 2) Determine whether the integer x exists in the list S of n integers. If it exists, return its index, and if not, return -1.
  - Parameters: x, S, n

Preliminaries

- What is an instance?
  - Actual value assigned to the parameter

- Example 1) Sort the list S of n integers in monotone increasing order.
  - Instances: $n = 6$, $S = [10, 7, 11, 5, 13, 8]$

- Example 2) Determine whether the integer x exists in the list S of n integers. If it exists, return its index, and if not, return -1.
  - Instances: $n = 6$, $S = [10, 7, 11, 5, 13, 8]$, $x = 5$

Preliminaries

- Algorithmic problem
  - A problem rigorously defined mathematically
  - It necessarily has a solution (finiteness)

- How to solve the algorithmic problem?
  - In the case of human:
    - Using numbers and operators
  - In the case of computer:
    - Using the computer’s instruction set

Preliminaries

- Problem:
  - Sort the list S of n integers in monotone increasing order
- Inputs:
  - $n \in \mathbb{N}$, Unsorted list S
- Output:
  - Soted list S
- Algorithm:

```
void sorting_algorithm(int n, int S[]){
    int i, j;    // indexes
    for (i = 0; i ≤ n; i++)
        for (j = i+1; j ≤ n; j++)
            if (S[j] < S[i])
                Swap S[i] and S[j]
}
```

Pseudo-code

A brief description of the procedures of the algorithm

Preliminaries

- Problem:
  - Determine whether the integer x exists in the list S of n integers. If it exists, return its index, and if not, return -1
- Inputs:
  - $n \in \mathbb{N}$, List $S \subseteq \mathbb{Z}$, $x \in \mathbb{Z}$
- Output:
  - Answer
- Algorithm:

```
void searching_algorithm(int n, const int S[], int x){
    int i;   // index
    for (i = 0; i < n; i++)
        if (S[i] == x)
            break;
    if (i < n)
        return i;
    else
        return -1;
}
```

Preliminaries

- Problem:
  - Calculate the sum of all elements in the list S composed of n integers
- Inputs:
  - $n \in \mathbb{N}$, List $S \subseteq \mathbb{Z}$
- Output:
  - sum
- Algorithm:

```
void sum_algorithm(int n, const int S[]){
    int i, sum = 0;   // index
    for (i = 0; i < n; i++)
        sum = sum+S[i];
    return sum;
}
```

Part 2

THEORETICAL ANALYSIS VS. EXPERIMENTAL ANALYSIS

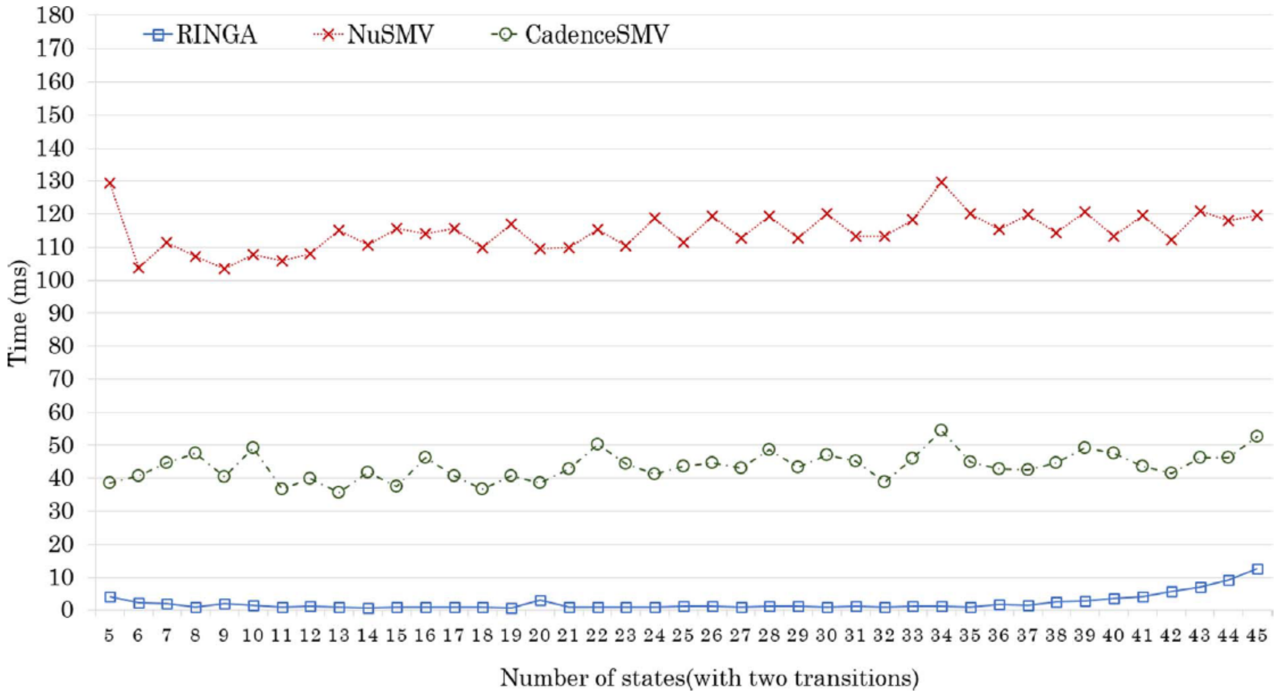
Theoretical vs. Experimental Analysis

- Algorithm analysis
  - Predicting the required resources
  - Resources:
    - Computational time, memory, communication bandwidth, hardware, etc.
  - Primarily measuring computational time

- Why should we analyze algorithms?
  - To identify the most efficient algorithm for a given problem

Theoretical vs. Experimental Analysis

- Experimental analysis
  - Implementing the given algorithm in source code
  - Run it in a real environment and measure the elapsed time
  - E.g., in the C programming language, the elapsed time of an algorithm is measured by using the '**clock()**' function



Theoretical vs. Experimental Analysis

- Limitations in experimental analysis
  - Implementing an algorithm requires additional time and efforts
  - There are various external factors that cannot be considered when measuring elapsed time:
    - Coding style
    - Hardware performance
    - Computer’s states at the execution
    - Etc.

Theoretical vs. Experimental Analysis

- Theoretical analysis
  - The method of theoretically describing the amount of resources required by an algorithm
  - Time and memory space are represented as the functions of the input size n
  - No implementation
  - No external factors
  - It is called computational complexity

Theoretical vs. Experimental Analysis

- Basic operations considered in theoretical analysis
  - Assigning (or substituting) a value to a variable
  - Function call
  - Returning the result value of a function
  - Arithmetic operations between variables
  - '**Get**' and '**Set**' operations of an array
  - Instructions defined in computer
  - Etc.

- Exact elapsed time for the above operations cannot be known
  - However, each operation is independent of the input size
  - Theoretically, these operations are considered to require constant time

Theoretical vs. Experimental Analysis

❖ Problem:

- Sort the list S of n integers in monotone increasing order

❖ Inputs:

- $n \in \mathbb{N}$, Unsorted list S

❖ Output:

- Soted list S

❖ Algorithm:

```
void sorting_algorithm(int n, int S[]){          ← 0
    int i, j;   // indexes                        ← n + 1
    for (i = 0; i ≤ n; i++)                       ← n − i + 1
        for (j = i+1; j ≤ n; j++)                 ← 1
            if (S[j] < S[i])                      ← 1
                Swap S[i] and S[j]
}
```

Theoretical vs. Experimental Analysis

- Problem:
  - Determine whether the integer x exists in the list S of n integers. If it exists, return its index, and if not, return -1
- Inputs:
  - $n \in \mathbb{N}$, List $S \subseteq \mathbb{Z}$, $x \in \mathbb{Z}$
- Output:
  - Answer
- Algorithm:

```
void searching_algorithm(int n, const int S[], int x){
    int i;    // index                         0
    for (i = 0; i < n; i++)                    n + 1
        if (S[i] = x)                          n
            break;
    if (i < n)                                 1
        return i;                              1
    else
        return -1;                             1
}
```

Theoretical vs. Experimental Analysis

- Problem:
  - Calculate the sum of all elements in the list S composed of n integers
- Inputs:
  - $n \in \mathbb{N}$, List $S \subseteq \mathbb{Z}$
- Output:
  - sum
- Algorithm:

```
void sum_algorithm(int n, const int S[]){
    int i, sum = 0;   // index              ← 1
    for (i = 0; i < n; i++)                 ← n + 1
        sum = sum+S[i];                     ← n
    return sum;                             ← 1
}
```

Part 3

COMPUTATIONAL COMPLEXITY

Computational Complexity

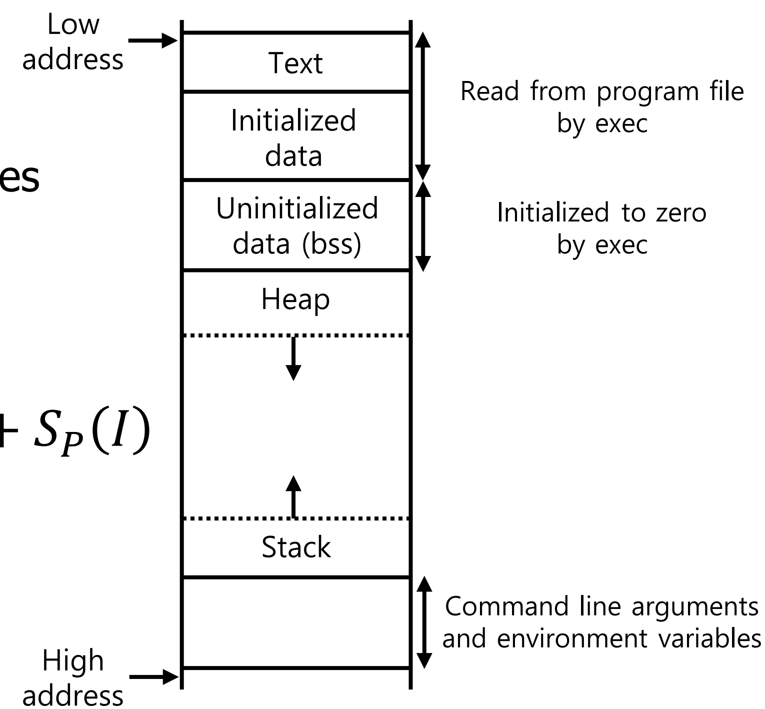
- Limitations in experimental analysis (Remind)
  - Implementing an algorithm requires additional time and efforts
  - There are various external factors that cannot be considered when measuring elapsed time:
    - Coding style
    - Hardware performance
    - Computer's states at the execution
    - Etc.

Computational Complexity

- How should theoretical analysis be performed?
  - Space complexity
    - The amount of space needed until the program terminates
    - E.g., RAM memory space
  - Time complexity
    - The amount of time (or operation) needed until the program terminates
    - E.g., CPU time

Computational Complexity

- Space complexity
  - Fixed space requirement: $S_P(C)$
    - Space determined at compile stage
    - Space requirement unrelated to the input size and iterations
  - Variable space requirement: $S_P(I)$
    - Space requirement dependent on instances
    - Space that varies at runtime
- Total space requirement: $S(P) = S_P(C) + S_P(I)$



Computational Complexity

❖ Space complexity

- Example 1

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

  - $S_{abc}(C)$: a, b, c
  - $S_{abc}(I)$: nothing

Computational Complexity

- Space complexity
  - Example 2

```
float sum (float list[], int n){
      float tempsum = 0;
      int i;
      for (i = 0; i < n; i++)
            tempsum += list[i];
      return tempsum;
}
```

- $S_{sum}(C)$: $list, n, tempsum, i$
- $S_{sum}(I)$: nothing

Computational Complexity

❖ Space complexity

- Example 3

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

- Recursive function is unknown how many recursive calls are needed
  - Space required for one iteration
    - $S_{rsum}(C)$: nothing
    - $S_{rsum}(I)$: 12 bytes $*$ MAX_SIZE
    - Sizeof(n) = 4 bytes
    - Sizeof(address of $list$[]) = 4 bytes
    - Sizeof(return address) = 4 bytes

Computational Complexity

- Time complexity
  - $T(P) = \text{Compile time} + T_P(n)$
    - $T_P(n) = C_a Add(n) + C_s Sub(n) + C_l LDA(n) + C_{st} Sta(n)$
      - C_a, C_s, C_l, C_{st}: the constant time required to perform each operation
      - $Add, Sub, LDA, Sta(n)$: the number of executions of each operation

  - Program step
    - A unit of the program that has semantic independence → 1 step
    - E.g., $a = 2$

      $a = 2 * b + 3 * c/d - e + f/g/a/b/c$
    - Both expressions are 1 step
    - Time required to execute 1 step should be independent of target instance

Computational Complexity

❖ Time complexity

- Computation of the total number of steps in the algorithm
  - Using the global variable '**count**'

```
int count = 0;
float sum (float list[], int n){
    float tempsum = 0;                          ←......... 1
    count++;   // for assignment
    int i;
    for (i = 0; i < n; i++){
        count++;   // for the loop              ←......... n
        tempsum += list[i];
        count++;   // for assignment            ←......... n
    }
    count++;   // last condition check of for loop   ←......... 1
    count++;   // for return                         ←......... 1
    return tempsum;
}
```

$T_{sum}(n) = 2n + 3$

Computational Complexity

- Time complexity
  - Computation of the total number of steps in the algorithm (cont'd)
    - Using the tabular method
      - Number of steps for the command line: steps / execution (s/e)
      - Number of iterations of the command line: frequency
      - Total number of steps = s/e x frequency

Commands	s/e	Freq.	Steps
float sum(float $list$[], int n){	0	0	0
float $tempsum = 0$;	1	1	1
int i;	0	0	0
for($i = 0$; $i < n$; i++)	1	$n + 1$	$n + 1$
$tempsum$ += $list[i]$;	1	n	n
return $tempsum$;	1	1	1
}			
Total	$2n + 3$		

Part 4

ASYMPTOTIC NOTATION

Asymptotic notation

- Time complexity
  - It only represents how many steps are required
  - There is a difference from the actual elapsed time

- Asymptotic analysis for computational complexity
  - Use several notations
  - E.g., Ω (omega), Θ (theta), O (big-oh)

Asymptotic notation

- Break even point
  - Condition for approximating the time complexity of a given program

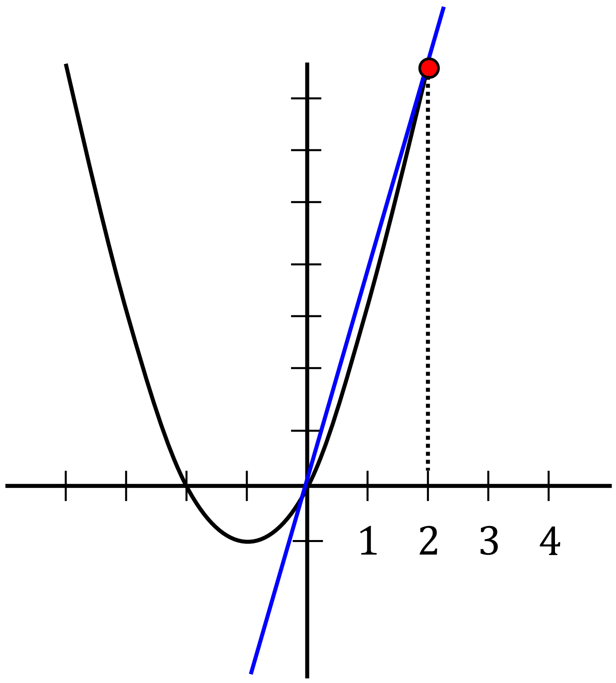
$f(n) = n^2 + 2n$

$g(n) = 4n$

$f(n) \leq g(n), \qquad 0 \leq n \leq 2$

$g(n) \leq f(n), \qquad 2 < n$

Break even point = 2



Asymptotic notation

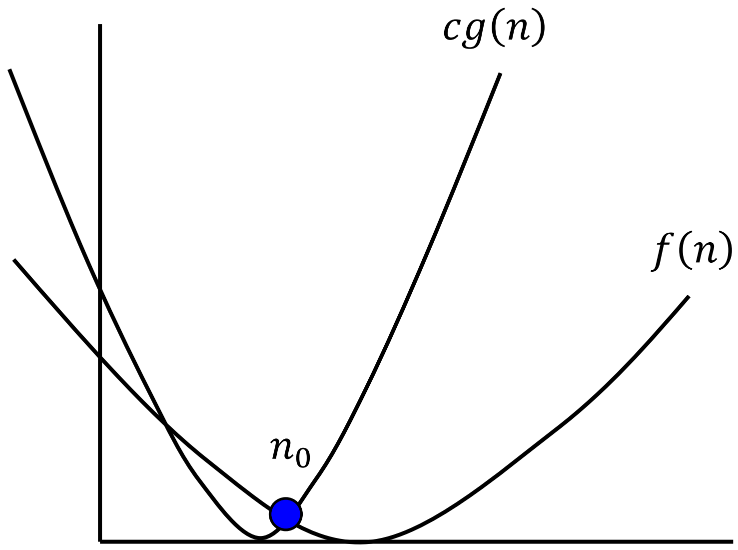
- Big-oh notation
  - For $\forall n \geq n_0$,

    if there exist positive constants c and n_0 such that $f(n) \leq cg(n)$, then

    $f(n) = \mathrm{O}\big(g(n)\big)$ (n_0 is break even point)

  $\mathrm{O}\big(g(n)\big) = \{f(n) \mid \exists c > 0, n_0 \geq 0, \quad s.t.\ \forall n \geq n_0, cg(n) \geq f(n)\}$

  - Here, $g(n)$ is the smallest function that satisfies the above conditions



Asymptotic notation

- Big-oh notation
  - The maximum elapsed time required for algorithm f for n input data
  - If the big O of $f(n)$ is $O(n^2)$, then $f(n)$ takes at most n^2 time in the worst case
  - Big-O notation indicates an asymptotic upper bound

Asymptotic notation

- Example
  - $n_0 = 2$,

    $f(n) = 3n + 2$
  - What is the Big O of $f(n)$?
    - Assume $cg(n) = cn$
    - $f(2) = 8 \leq cg(2) = 2c$
    - $8 \leq 2c$
    - $4 \leq c$
    - $f(n) = O\big(g(n)\big)$
    - $\therefore f(n) \cong O(n)$

Asymptotic notation

- Practice
  - Obtain the big-O of the following $f(n)$
    - $n_0 = 3$, $f(n) = 5n + 1$
    - $n_0 = 4$, $f(n) = n^2 + 2n + 1$
    - $n_0 = 4$, $f(n) = 6 \cdot 2^n + n^2$
  - Obtain the big-O of the following $f(n)$
    - $f(n) = 3n + 1000$
    - Is $f(n) = O(3n + 1000)$ correct?

Asymptotic notation

- Practice
  - What if we don't know the break even point n_0?
    - $f(n) = 3n^2 - 4n + 2$
    - Assume $cg(n) = cn^2$, then $3n^2 - 4n + 2 \leq cn^2$
    - $(3 - c)n^2 - 4n + 2 \leq 0$
    - $(3 - c)n^2 \leq 4n - 2$
    - $\lim_{n\to\infty} \frac{(3-c)n^2}{n^2} \leq \frac{4n-2}{n^2}$
    - $3 - c \leq 0$
    - $c \geq 3$
    - $c = 3$, $3n^2 - 4n + 2 \leq 3n^2$
    - $2 \leq 4n$, $n \geq \frac{1}{2}$, $n_0 = 1$
    - $f(n) \cong O\big(g(n)\big) \cong O(n^2)$

Asymptotic notation

- Practice
  - Obtain the big-O of the following $f(n)$
    - $f(n) = 100n^3 + 10n \log n + 2$
    - $f(n) = \log n + 2 \log \log n - 3$
    - $f(n) = 2^{n+2}$
    - $f(n) = n!$

Asymptotic notation

- Big O of a constant function
  - Constrant function
    - A function that always has a constant value, regardless of the size of the input data n
    - e.g., $f(n) = 30 \cdot 1$

      $f(n) = O(1)$
    - Proof: $f(n) \le cg(n)$

      $30 \cdot 1 \le c \cdot 1$

      $c = 30$
  - All basic instructions of a computer are $O(1)$

Asymptotic notation

- Rules of Big O notation
  - The coefficient of each term in the function can be omitted
    - e.g., $f(n) = 13n^2 \cong O(n^2)$
  - When $a > b$, if terms n^a and n^b exist, only consider the n^a term
    - n^a dominates n^b
  - The exponential term (2^n) dominates the polynomial term (n^2)
  - The polynomial term (n^2) dominates the logarithm term ($\log n$)

Asymptotic notation

❖ Omega notation

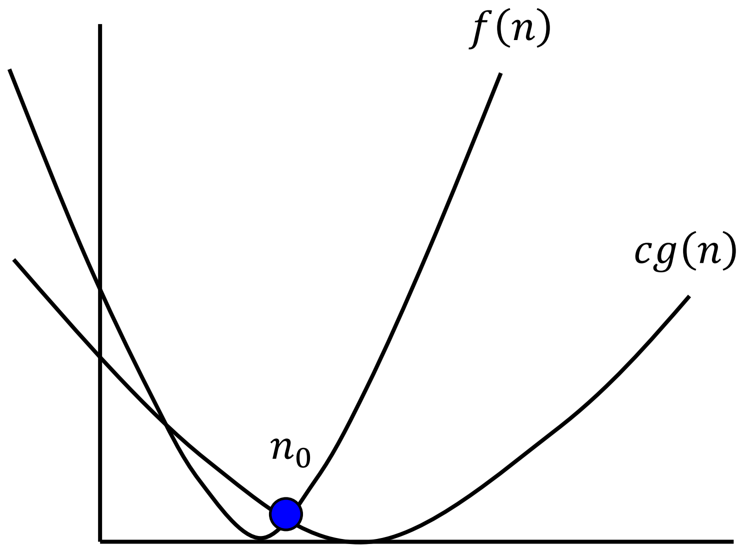
- For $\forall n \geq n_0$,

  if there exist positive constants c and n_0 such that $f(n) \geq cg(n)$, then

  $f(n) = \Omega\big(g(n)\big)$ (n_0 is break even point)

$$\Omega\big(g(n)\big) = \{f(n) \mid \exists c > 0, n_0 \geq 0, \qquad s.t.\ \forall n \geq n_0, f(n) \geq cg(n)\}$$

- Here, $g(n)$ is the largest function that satisfies the above conditions



Asymptotic notation

- Omega notation
  - The minimum elapsed time required for algorithm f for n input data
  - If the Omega of $f(n)$ is $\Omega(n^2)$, then $f(n)$ takes at most n^2 time in the best case
  - Omega notation indicates an asymptotic lower bound

Asymptotic notation

- Practice
  - Obtain the Omega of the following $f(n)$
    - $f(n) = 3n + 1$
    - $f(n) = 100n + 6$
    - $f(n) = 10n^2 + 4n + 2$
    - $f(n) = 6 \cdot 2^n + n^2$

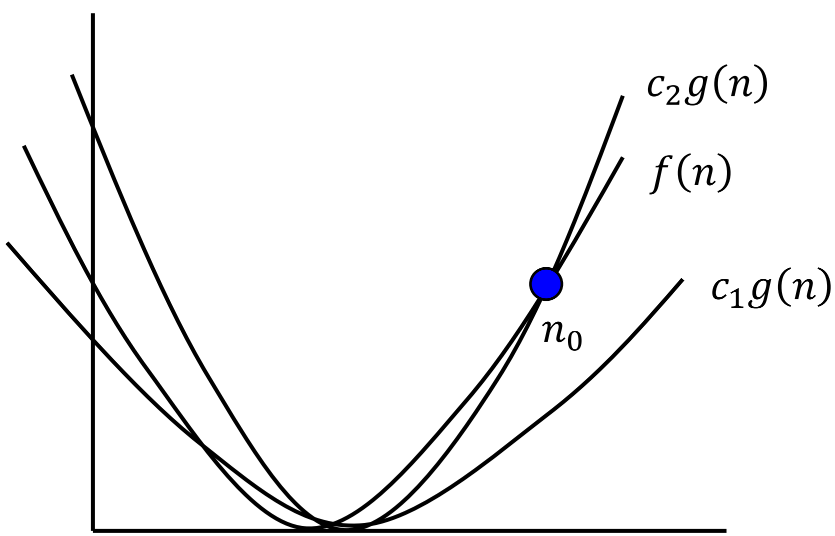
Asymptotic notation

- Theta notation
  - For $\forall n \geq n_0$,

    if there exist positive constants c_1, c_2, and n_0 such that

    $c_1 g(n) \leq f(n) \leq c_2 g(n)$, then $f(n) = \Theta\big(g(n)\big)$ (n_0 is break even point)

$\Theta\big(g(n)\big) = \{f(n) \mid \exists c_1, c_2 > 0, n_0 \geq 0, s.t.\ \forall n \geq n_0, c_2 g(n) \geq f(n) \geq c_1 g(n)\}$



Asymptotic notation

❖ Theta notation

- The average elapsed time required for algorithm f for n input data
- If the Theta of $f(n)$ is $\Theta(n^2)$, then $f(n)$ takes at most n^2 time in the average case

Asymptotic notation

- Practice
  - Obtain the Theta of the following $f(n)$
    - $f(n) = 3n + 3$
    - $f(n) = 10\log n + 4$
    - $f(n) = 10n^2 + 4n + 2$
    - $f(n) = 6 \cdot 2^n + n^2$

Asymptotic notation

- Algorithm analysis using asymptotic notation
  - The addition of two matrices

Commands	Asymptotic complexity
void mat_add(int $a[n][m]$, int $b[n][m]$, int $c[n][m]$){	0
int i, j;	0
for($i = 0$; $i < n$; i++)	$O(n)$
for($j = 0$; $j < m$; j++)	$O(n \cdot m)$
$c[i][j] = a[i][j] + b[i][j]$;	$O(n \cdot m)$
}	0
Total	$O(n \cdot m)$

Asymptotic notation

- Algorithm analysis using asymptotic notation
  - Binary search (complicated case)

```
int binsearch(int list[], int snum, int left, int right){
    int mid;
    while(left ≤ right){
        mid = (left + right)/2;
        switch(comp(list[mid], snum)){
            case -1: left = mid + 1   // mid < snum
                     break;
            case 0: return mid;   // mid = snum
            case 1: right = mid − 1;}   // mid > snum
    }
    return -1;
}
```

Asymptotic notation

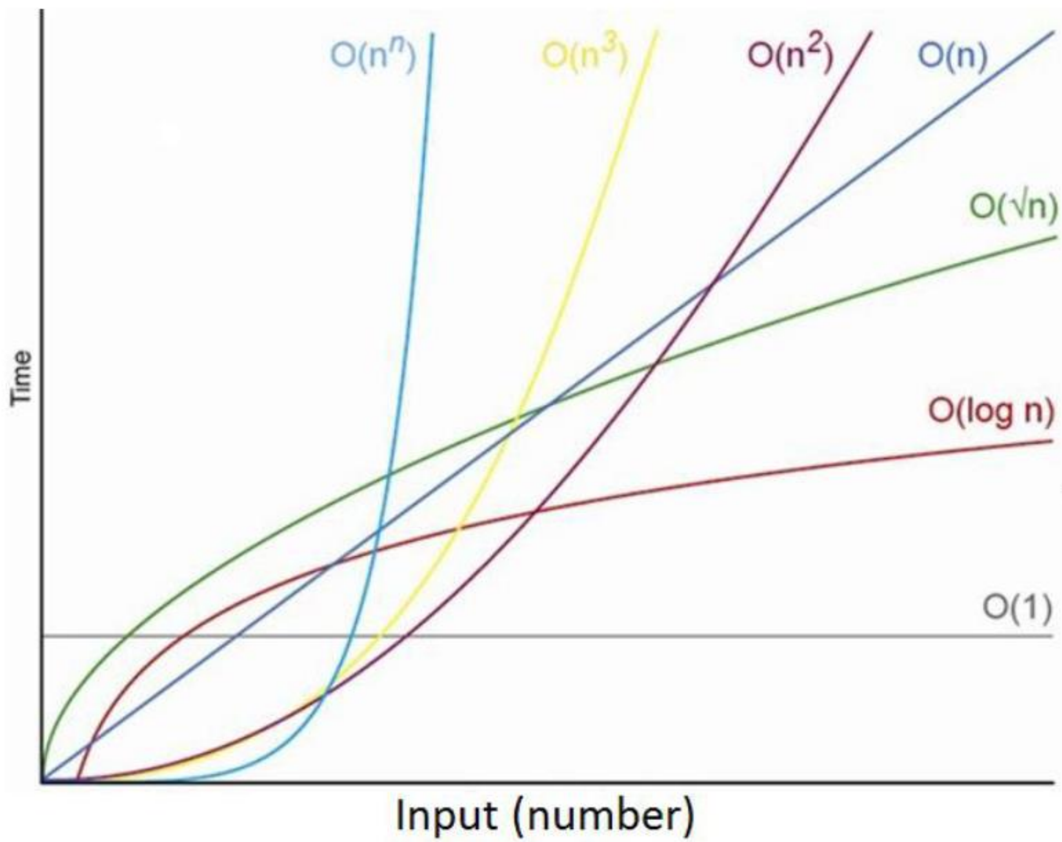
- Practical complexities
  - The value of general complexities
    - Assume the computer performs ten commands per second
    - If $n = 32$, an algorithm with $O(2^n)$ complexity would require 13.6 years to complete

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

Asymptotic notation

❖ Practical complexities

- The value of general complexities



BETTER

- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time

WORSE

Summary

- What is an algorithm?
  - Procedure for finding the optimal solution to a finite problem

- Algorithm analysis
  - Predicting the required resources
  - Experimental vs. theoretical analysis

- Computational complexity
  - Time and space complexities

- Asymptotic notations
  - Big O, Omega, Theta

Questions?

SEE YOU NEXT TIME!