ALGORITHMS AND DATA STRUCTURES II

Lecture 1
Algorithms and their Complexity

Lecturer: K. Markov

markov@u-aizu.ac.jp

Course webpage:

https://elms.u-aizu.ac.jp

Course Overview

- **Schedule:**
  - 6/09 – L1,
  - 6/13 – L2,
  - 6/16 – L3,
  - 6/20 – L4,
  - 6/23 – L5
  - 6/27 – L6,
  - 6/30 – MidTerm,
  - 7/04 – L7,
  - 7/07 – L8,
  - 7/11 – L9,
  - 7/14 – L10,
  - 7/21 – L11,
  - 7/25 – L12,
  - 7/28 – L13,
  - 8/XX – Final Exam.

- **Exams:**
  - MidTerm – Lectures 1 to 6.
  - Final – Lectures 7 to 12.

Course Overview

- **Grading**
  - Exercises – 40%
  - MidTerm Exam– 30%
  - Final Exam – 30%

- **Exercises**
  - Text problems.
  - Programming tasks.

Course Management

- Using UoA **Moodle** system:
  - https://elms.u-aizu.ac.jp
- Course name:
  - [IT_CMV-SE-DE] FU09 Algorithms and Data Structures II
- Exercises downloaded from **Moodle.**
- Answers uploaded to **Moodle.**
- Grades, comments – from **Moodle.**

Today's Outline

- **Algorithms:**
  - Definition.
  - Basic concepts.
- **Function growth.**
  - Upper bound.
  - Lower bound.
  - Tight bound.
- **Algorithm complexity**.
- **Merge sort algorithm**.

ALGORITHMS

- To solve any problem by a computer, we need an algorithm.
- Given an algorithm for the problem, we want to know the efficiency the algorithm.
- We are most interested in how much time and how much memory ***space*** the algorithm takes to solve the problem.

ALGORITHMS

- What is an algorithm?

An algorithm is a well-defined computational **procedure** that transforms inputs into outputs, achieving the desired input-output relationship.

ALGORITHMS

- The **computation time** of an algorithm depends on the number of computational steps of the algorithm and the computer used.
- To evaluate the efficiency of algorithms, it is ideal to use an **unique computer** to measure their computation time.

ALGORITHMS

- The computation time of an algorithm for a problem **depends** on the size of the problem.
- Important! How the computation time of the algorithm grows when the size of the problem increases.

ALGORITHMS

- The size of a problem is denoted by an integer ***n***, which is a measure of the quantity of input data.
  - The size of a matrix multiplication problem might be the **largest dimension** of the matrices.
  - The size of a sorting problem might be the **number of data** to be sorted.
  - The size of a graph problem might be the **number of vertices** or edges.

Algorithms Complexity

- The computation time needed by an algorithm expressed as a function of the size of a problem is called **time complexity** of the algorithm.
- Analogous definition can be made for **space complexity.**

Algorithms Complexity

- Given an algorithm for a problem of size ***n***, it is important to find the time complexity and how the time complexity grows when ***n*** increases.

- It is the <u>growth rate</u> of the **time complexity** (space complexity) of an algorithm which ultimately determines the **size** of problems that can be solved by the algorithm.

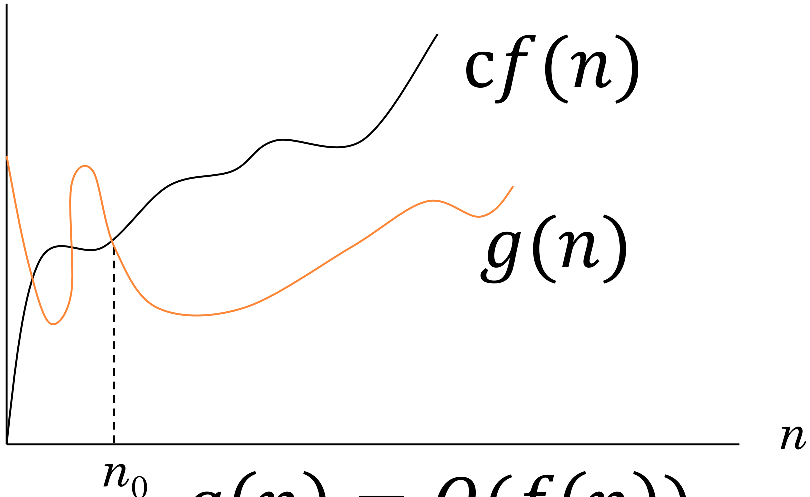
Growth of Functions

- **Upper bound.** $g(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $g(n) \leq cf(n)$.
- **Lower bound.** $g(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $g(n) \geq cf(n)$.
- **Tight bound.** $g(n) = \Theta(f(n))$ if $g(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

- Example: $g(n) = 32n^2 + 17n + 32$.
  - $g(n)$ is $O(n^2), O(n^3), \Omega(n^2), \Omega(n)$, and $\Theta(n^2)$.
  - $g(n)$ is not $O(n), \Omega(n^3), \Theta(n)$, or $\Theta(n^3)$.

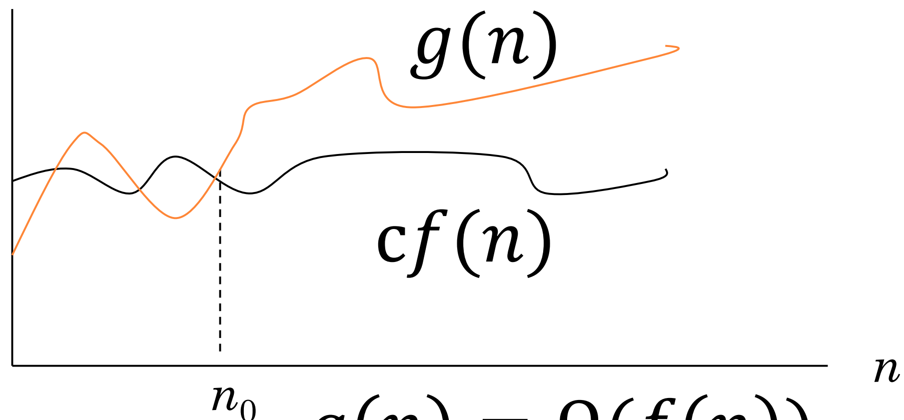
Growth of Functions

Upper bound



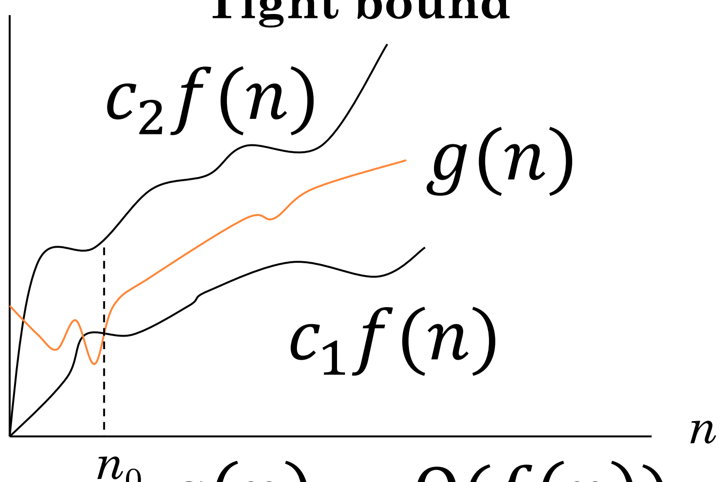
$$g(n) = O(f(n))$$

Lower bound



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

Tight bound



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

Growth of Functions

- **Upper bound** says that if constant factors are ignored $f(n)$ is at least as large as $g(n)$.
- $g(n) = O(f(n))$ means that the growth rate of $g(n)$ is *smaller than or equal* to the growth rate of $f(n)$.
- $O(\ldots)$ is read ``order ..." or ``Big-Oh ...."

GROWTH OF FUNCTIONS

- **Lower bound** $g(n) = \Omega(f(n))$ (read "omega") means that the growth rate of $g(n)$ is *greater than or equal to* the growth rate of $f(n)$.
- **Tight bound** $g(n) = \Theta(f(n))$ means that for all n right of n_0, the value of $g(n)$ lies at or above $c_1 f(n)$ and at or below $c_2 f(n)$.

Growth of Functions

- Prove $g(n) = an^2 + bn + c = \Theta(n^2)$
  - a, b, c are constants and $a > 0$.
  - Find c_1, and c_2 (and n_0) such that
    $c_1 n^2 \leq g(n) \leq c_2 n^2$ for all $n \geq n_0$.
  - It turns out: $c_1 = a/4$, $c_2 = 7a/4$ and
    $n_0 = 2\max(|b|/a, \sqrt{|c|/a})$
  - Here we also can see that lower terms and constant coefficient can be ignored.
  - How about $g(n) = an^3 + bn^2 + cn + d$?

Growth of Functions

- **Properties:**

If $g_1(n)$ is $O(f_1(n))$, $g_2(n)$ is $O(f_2(n))$ then

- $g_1(n) + g_2(n)$ is $O(\max(f_1(n), f_2(n)))$
- $g_1(n)g_2(n)$ is $O(f_1(n)f_2(n))$
- $ag_1(n)$ is $O(f_1(n))$ for any constant a.

Growth of Functions

- Bounds for some functions.
  - Polynomials.

  $a_0 + a_1 n + \cdots + a_d n^d$ is $O(n^d)$ if $a_d > 0$.
  - Logarithms.

  $O(\log_a n) = O(\log_b n) = O(n)$ for $a, b > 0$.
  - Exponentials.

  For every $r > 1$ and every $d > 0, n^d = O(r^n)$.
  - Constant.

  $O(c) = O(1)$ for any c.

Growth of Functions

- Typical growth functions.

Function	Name
c	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Loglinear
n^2	Quadratic
n^3	Cubic
2^n	Exponential

ALGORITHMS

- Running time examples

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

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

Algorithms

- Statement complexity.
  - ***for/while*** loop:

for _i_ = 1 to m:

S

if the computation time of ***S*** is $t_i(n)$ for each i then the computation time of the for statement is $\sum_{i=1}^{m} t_i(n)$.

If $t_i(n) = t(n)$ for all i then the computation time of the loop is $mt(n)$.

ALGORITHMS

- Statement complexity.
  - ***if/else*** statement:

if (condition):

$\boldsymbol{S_1}$

else:

$\boldsymbol{S_2}$,

let $t_1(n)$ and $t_2(n)$ be the computation times of $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$, respectively. The computation time of the **if** statement is $\max\{t_1(n), t_2(n)\}$.

ALGORITHMS

- Statement complexity.
  - **Consecutive** statements:

…

S₁

S₂

…

Let $t_1(n)$ and $t_2(n)$ be the computation times of two consecutive statements, respectively. The total computation time of the two statements is $t_1(n) + t_2(n)$.

ALGORITHMS

- Linear Time: $O(n)$
  - Running time is at most a constant factor times the size of the input.

```
max = a1
for i = 2 to n {
   if (ai > max)
      max = ai
}
```

  - Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

ALGORITHMS

- **Quadratic Time:** $O(n^2)$
  - Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.
  - $O(n^2)$ solution. Try all pairs of points.

```
min = (x₁ - x₂)² + (y₁ - y₂)²
for i = 1 to n {
   for j = i+1 to n {
      d = (xᵢ - xⱼ)² + (yᵢ - yⱼ)²
      if (d < min)
         min = d
   }
}
```

ALGORITHMS

Polynomial Time: $O(n^k)$

- Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?
- $O(n^k)$ solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

  - Checking whether S is an independent set is $O(k^2)$.
  - Number of k element subsets is $\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots(2)(1)} \leq \frac{n^k}{k!}$
  - Total complexity is $O\left(\frac{k^2 n^k}{k!}\right) = O(n^k)$.

ALGORITHMS

- Exponential Time
  - Given a graph, which is the largest independent set?
  - $O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* = ϕ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* = S
   }
}
```

  - Checking whether S is an independent set is $O(k^2)$.
  - Number of all subsets of S is $O(2^n)$.
  - Total complexity is $O(n^k 2^n)$.

Merge-Sort Algorithm

- **Step 1:** divide the *n*-element sequence into two sub-problems of *n*/2 elements each.
- **Step 2**: sort the two subsequences recursively using merge sort. If the length of a sequence is *1*, do nothing since it is already in order.
- **Step 3**: merge the two sorted subsequences to produce the sorted answer.

Merge-Sort Algorithm

- Pseudo-code

```
def merge_sort(A):
    middle = len(A) / 2
    left = merge_sort (A[1:middle])
    right = merge_sort (A[middle+1:end])
    return merge (left, right)
```

Merge-Sort Algorithm

- Pseudo-code

def *merge* (***A*, *B***):
    $result\ = < empty >$
        while len$(A)\ >\ 0$ **or len**$(B)\ >\ 0$:
          if len$(A)\ >\ 0$ **and len**$(B)\ >\ 0$:
            if $A[1]\ <=\ B[1]$:
                append $result$ **with** $A[1]$, **delete** $A[1]$
            else:
              append $result$ **with** $B[1]$, **delete** $B[1]$
          else if len$(A)\ >\ 0$:
            append $result$ **with** $A[1]$, **delete** $A[1]$
          else if len$(B)\ >\ 0$:
            append $result$ **with** $B[1]$, **delete** $B[1]$
  return $result$

Merge-Sort Algorithm

- Animated example

6 5 3 1 8 7 2 4

Merge-Sort Algorithm

- **Time complexity**

- There are two recursive calls, each of them sorts a sequence of $n/2$, and the statements after the two recursive calls take $O(n)$ time.
- Let $t(n)$ be the time complexity of the algorithm, then

$$\boldsymbol{t(n) = 2t(n/2) + cn}$$

and $t(2) = O(1)$, where c is a constant. Solving the equation, $\boldsymbol{t(n) = O(n \log n)}$.

That's All for Today!