アルゴリズムとデータ構造

- 第５回講義トピック：計算量
  - アルゴリズムの解析
    - コストと計算量の比較
  - アルゴリズムの実現
    - 開発過程の最適化

Algorithms and Data structures

- Topic of lecture note no.5: complexity
  - Complexity of an algorithm
  - Cost optimization for algorithm development

アルゴリズムのコスト

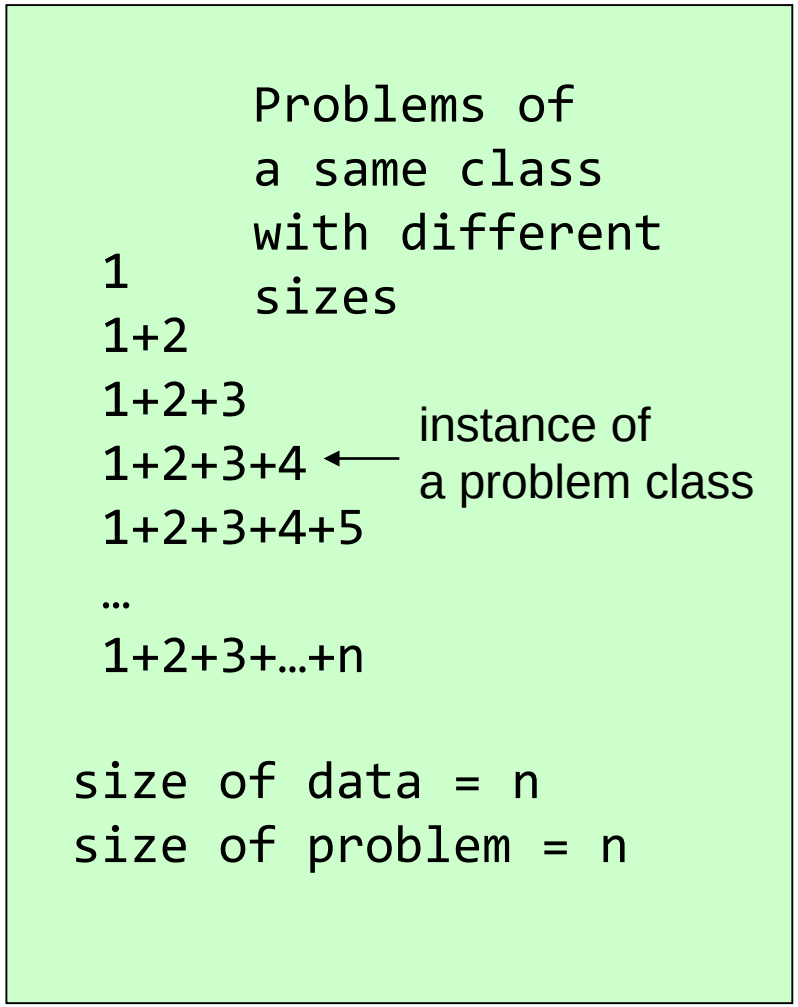
- 人間による開発コスト（ソフトウエアコスト）
  - 問題の理解
  - アルゴリズムの実装
- 機械による実行コスト（ハードウエアコスト）
  - 実行時間
  - メモリー空間
- 数回しか実行しないようなアルゴリズムはハードウエアコストよりもソフトコストの方を重視するべきである。
- これに対して、繰り返し使われるアルゴリズムはソフトウエアコストよりもハードウエアコストの方を重視するべきである。

Cost of algorithm development

- Software cost (human resource)
  - analysis of algorithms
  - implementation of algorithms
- Hardware cost (machine resource)
  - time cost
  - memory cost
- Priority
  - For some very important tasks with huge complexity, hardware cost will be the first priority.
  - For some algorithms that will be used many times repeatedly, the hardware cost is also important.
  - For some algorithms that will only be used for several times, then we should consider the software cost firstly.

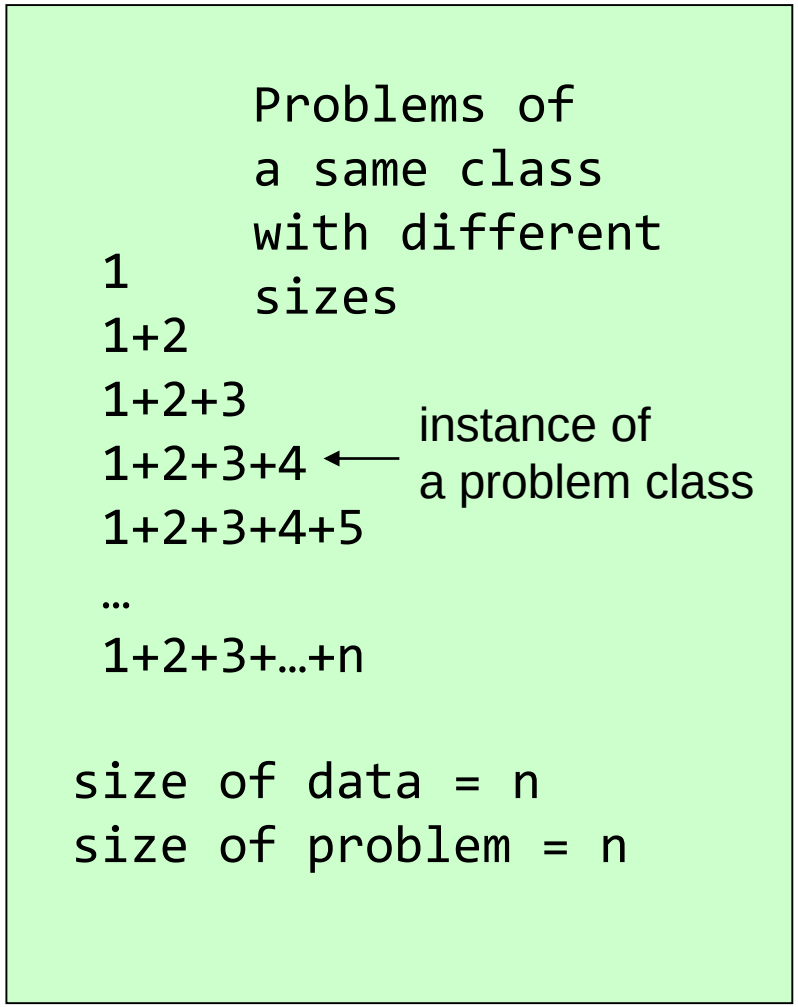
問題のサイズとクラス

- 問題のクラス
  同じタイプの問題の集まりを言う
- 問題のサイズ
  問題の大きさを表す量で、データのサイズと関係する
- 問題クラスのインスタンス
  ある問題クラスにおいて、与えたれた1つの具体的な問題



The size of a problem class

- Problem class:
  a set of problems with same type
- Size of a problem:
  the size of data that will be processed for the problem.
- An instance of a problem class:
  a practical problem of a problem class given with a specific data input.



アルゴリズムの計算量 (complexity)

- 基本操作
  - プログラムの実行時間はいろんな要素に影響される
    - 計算機の実行速度
    - プログラムの効率
    - コンパイラーの良し悪し
    - コードの最適化の具合
    - 命令の数など
  - アルゴリズムの計算量を分析する際はこういったアルゴリズム固有の要素以外のものについては考えない。アルゴリズムの中の基本操作だけを取り出して分析を行う。
  - また、データサイズが小さい時の詳細な振る舞いよりも、特に問題のサイズnが大きくなる時のアルゴリズム計算量の増加の傾向を解析する
  - 計算量、メモリ量などの資源のオーダーを調べる。

Complexity of an algorithm

- Running time of a program depends on many factors
  - efficiency of algorithm
  - computation power of a machine
  - efficiency of compiler
  - code optimization
  - parallel processing
- Abstract computer and base operation
  - Since algorithm if not machine dependent, to analyze an algorithm, we need to count some base operations that are proper for solving a problem set.
  - abstract computer: an abstract machine that executes the base operations to solve problems.
- Order of complexity
  - Analysis of the order of complexity for behavior of an algorithm when the data size is very large.

最大計算量と平均計算量

- 最大計算量:
  - 考えられるワーストケースのデータに対して必要な計算量
  - 普通、計算量といえば、最大計算時間を指す
- 平均計算量:
  - すべての可能性のある入力データに対する計算量の平均
  - 一般的に、最大計算量よりも求めることが難しい
  - しかし、最大計算量より平均計算量の方が大切な場合がある
    - 例えば、クイックソートの最大計算量はN^2であるが、平均計算量はNlogNのオーダーとなり、よいアルゴリズムとして知られる。

$$W(n) = \max_{\forall \text{ instance of size } n} T(I)$$

$$A(n) = \sum_{\forall \text{ instance of size } n} p(I)T(I)$$

Maximum complexity and average complexity

- Maximum complexity:
  - complexity for the data input of worst case
  - in most case, analysis of algorithm is for maximum complexity
- Average complexity:
  - The average complexity for all possible data inputs
  - It is more difficult to analyze the average complexity than the maximum complexity.
  - Cases that average complexity is needed:
    For example: the quick sort algorithm has maximum complexity in order N^2 same as simple basic sort algorithms, but the average complexity is in order of NlogN, faster than other basic sort algorithms.

$$W(n) = \max_{\forall \text{ instance of size } n} T(I)$$

$$A(n) = \sum_{\forall \text{ instance of size } n} p(I)T(I)$$

解析の進め方

- 解析法を決める
  - 最大計算量か平均計算量を決める
- アルゴリズムから基本操作を取り出す
  - 詳細の実現法によらない基本的なアルゴリズムの操作
  - 一回だけ、あるいは低いオーダーの処理は無視する
    - 90%の時間が10%のコードで使われる
- 数学的な解析を行う
  - 基本操作の平均計算時間あるいは最大計算時間を数式で表す
  - 解析結果をオーダーで形で表表す
    - $N < N\log N < N^2 < N^3 < 2^N$の順で計算量が増える
      ここで、logNの底は定数の問題であるので、特に問題にしない
      特別示す必要がある場合はlgN, lnNなどで表す

Algorithm analysis

- Maximum complexity or average complexity
- Base operations
  Some common abstract operations to be repeated for solving the problems of a same class.
  (90% time is for 10% code)
- Analysis
  - To figure out the math equation for the maximum (or average) complexity with the base operations.
  - To represent the complexity with the description of order.
    We know the complexity increases with
    $1 < \log N < N < N\log N < N^2 < N^3 < 2^N < N!$
    Here we do not care about the base of logN, since they are only different with a constant factor.

異なるアルゴリズムの計算量の比較

- 絶対比較

  同じ問題を解くアルゴリズム1とアルゴリズム2を同じ計算機を実行し、アルゴリズム1が速く結果が得られれば、アルゴリズム1はアルゴリズム2よりもよいアルゴリズムであるといえる。
  ＊これは工学的な問題の考え方である。

- 増加傾向比較

  しかし、理論的には問題サイズnが小さい時よりも、nが非常に大きくなった時の増加の傾向が問題になる。例えば

  - アルゴリズム1： $W1(n) = 1000N^2$
  - アルゴリズム2： $W2(n) = 10n^3$
  - $n < 100$の時： $W1(n) > W2(n)$
  - $n = 100$の時： $W1(n) = W2(n)$
  - $n > 100$の時： $W1(n) < W2(n)$

  従って、増加の傾向で比較した場合はアルゴリズム1がよりよいアルゴリズムと言える。

Comparison of two algorithms

- In absolute running time
  to compare two algorithms using a same computer and same data set, the faster one is better (a engineering measure).
- In order of complexity
  to see how the complexity increase when the data size n increase.
- For algorithms $W1(n) = 1000N^2$, and $W2(n) = 10n^3$
  - When $n < 100$: $W1(n) > W2(n)$
  - When $n = 100$: $W1(n) = W2(n)$
  - When $n > 100$: $W1(n) < W2(n)$

  Algorithm W1 is better than W2 when n increases.

いくつかの関数のNによる変化の比較

logN	9	19
$\log^2 N$	81	361
N	1,000	1,000,000
NlogN	9,000	19,000,000
$N\log^2 N$	81,000	361,000,000
N^2	1,000,000	1兆

Comparison between some different functions

logN	9	19
$\log^2 N$	81	361
N	1,000	1,000,000
NlogN	9,000	19,000,000
$N\log^2 N$	81,000	361,000,000
N^2	1,000,000	1,000,000,000,000

計算量のO記法

- $g(n)$ is $O(f(n))$ (no worse than), if $^{\exists}c$ $(\infty > c \geq 0)$

$$\lim_{n\to\infty} g(n)/f(n) = c$$

  Or, $^{\exists}K > 0$ and $^{\exists}n_0$, $g(n) \leq Kf(n)$ for $^{\forall}n \geq n_0$

- $g(n)$ is $\Omega(f(n))$ (no better than), if $^{\exists}c$ $(\infty \geq c > 0)$

$$\lim_{n\to\infty} g(n)/f(n) = c$$

  Or, $^{\exists}K > 0$ and $^{\exists}n_0$, $g(n) \geq Kf(n)$ for $^{\forall}n \geq n_0$

- $g(n)$ is $\Theta(f(n))$ (the same efficiency as), if $^{\exists}c$ $(0 < c < \infty)$

$$\lim_{n\to\infty} g(n)/f(n) = c$$

  Or, $^{\exists}K_1$, $^{\exists}K_2 > 0$ and $^{\exists}n_0$, $K_1 f(n) \leq g(n) \leq K_2 f(n)$ for $^{\forall}n \geq n_0$

- If g is big $O(f)$ but f is not $O(g)$ then, $O(g)$ is better than $O(f)$.

Big O notation

- $g(n)$ is $O(f(n))$ (no worse than), if $^{\exists}c$ $(\infty > c \geq 0)$

$$\lim_{n \to \infty} g(n)/f(n) = c$$

  Or, $^{\exists}K > 0$ and $^{\exists}n_0$, $g(n) \leq Kf(n)$ for $^{\forall}n \geq n_0$

- $g(n)$ is $\Omega(f(n))$ (no better than), if $^{\exists}c$ $(\infty \geq c > 0)$

$$\lim_{n \to \infty} g(n)/f(n) = c$$

  Or, $^{\exists}K > 0$ and $^{\exists}n_0$, $g(n) \geq Kf(n)$ for $^{\forall}n \geq n_0$

- $g(n)$ is $\Theta(f(n))$ (the same efficiency as), if $^{\exists}c$ $(0 < c < \infty)$

$$\lim_{n \to \infty} g(n)/f(n) = c$$

  Or, $^{\exists}K_1$, $^{\exists}K_2 > 0$ and $^{\exists}n_0$, $K_1 f(n) \leq g(n) \leq K_2 f(n)$ for $^{\forall}n \geq n_0$

- If g is big $O(f)$ but f is not $O(g)$ then, $O(g)$ is better than $O(f)$.

計算量の式のオーダーの演算

- 定数はオーダーに影響しない
  - O(c f(n)) = O(f(n))
  - O(f(n)+a) = O(f(n))
- 計算量の加算:
  - O(f(n)+g(n)) = O(max(f(n), g(n)))
- 計算量の乗算:
  O(f(n))の計算量をもつループをO(g(n))回繰り返し実行する場合の計算量
  - O(f(n)･g(n)) = O(f(n))･O(g(n))

Arithmetic of Big-O notation

- Ignore constants
  $O(c\ f(n)) = O(f(n))$
  $O(f(n)+a) = O(f(n))$
- Sum
  $O(f(n)+g(n)) = O(\max(f(n), g(n)))$
- Multiplication
  e.g., repeat an algorithm with order $O(f(n))$ for order $O(g(n))$ times
  - $O(f(n)\cdot g(n)) = O(f(n))\cdot O(g(n))$

例：線形探索法の計算量

```
struct {
   int key;
   int data;
} table[100];
int n;
int search(int key) {
(1) int i=0;
(2) while (i<n) {
(3)   if(table[i].key == key)
(4)     return(table[i].data);
(5)   i++;
    }
(6) return -1;
}
```

- プログラムの命令の実行回数

(1)	1	O(1)
(2)	1*n/2	O(n)
(3)	1*n/2	O(n)
(4)	1	O(1)
(5)	1*n/2	O(n)
(6)	1	O(1)

- whileループ(WL)全体
  - O(n/2+n/2+n/2)=O(n)
  - あるいは O((1+1+1)･n/2)= O(n)
- 関数全体: (1),WL,(4),(6)
  - O(1+n+1+1)=O(n)

Example of linean search

```
struct {
   int key;
   int data;
} table[100];
int n;
int search(int key) {
(1) int i=0;
(2) while (i<n) {
(3)   if(table[i].key == key)
(4)     return(table[i].data);
(5)   i++;
    }
(6) return -1;
}
```

- Number of operation

(1)	1	O(1)
(2)	1*n/2	O(n)
(3)	1*n/2	O(n)
(4)	1	O(1)
(5)	1*n/2	O(n)
(6)	1	O(1)

- For the whole while loop
  O(n/2+n/2+n/2)=O(n)
  or
  O((1+1+1)·n/2)= O(n)
- For the whole function
  O(1+n+1+1)=O(n)

アルゴリズムの分類

- 計算量をデータサイズ**Nの関数**とし、
  - **計算量：1**　　どんな大きなデータに対しても有限回の操作で解決できる。実行時間が一定値以下→「究極のアルゴリズム」！
  - **計算量：logN**　　計算量が対数関数的に増える。Nが大きくなってもほんの少ししか増加しない。
  - **計算量：N**　　計算量が線形関数で、入力データに比例して増加する。
  - **計算量：NlogN**　　問題を分割して解き、そして合わせるタイプのアルゴリズムで現れる。
  - **計算量：N^2**　　２重ループのような場合で現れる。Nが小さい時しか実用的でない。
  - **計算量：N^3**　　ほんの小さいNでしか実用的でない。
  - **計算量：2^N**　　力づくで問題を解く解法で現れる。一般的に実用的でない。

Some typical algorithms

- Complexity for data size N,
  - 1: constant, e.g., lookup table
  - loglogN: double logarithmic, e.g., interpolation search
  - logN: logarithmic, e.g., binary search
  - N: linear, e.g., linear search
  - NlogN: linearithmic, e.g., merge sort
  - N^2: quadratic, typically for algorithms with double nested loops, e.g., selection sort
  - N^3: cubic, typically for algorithms with triply nested loops, e.g., partial correlation
  - 2^N: exponential, e.g., DP (dynamic programming)
  - N!: factorial, e.g., travelling salesman problem

漸化式タイプと計算量オーダー

- N回の操作で入力サイズを1つ減らすことを再帰
  $C_N = C_{N-1} + N$ for $(N>=2)$, $C_1 = 1$
  計算量: $C_N \sim N^2/2$、$(O(N^2))$
  e.g., selection sort
- 1回の操作で入力サイズを半分に減らす再帰
  $C_N = C_{N/2} + 1$ for $(N>=2)$, $C_1 = 0$
  計算量: $C_N \sim \log_2 N$、$(O(\log N))$
  e.g., binary search
- N回の操作で入力サイズを半分に減らす再帰
  $C_N = C_{N/2} + N$ for $(N>=2)$, $C_1 = 0$
  計算量: $C_N \sim 2N$、$(O(N))$
  e.g.,
- N回の操作で入力を2分割する再帰
  $C_N = 2C_{N/2} + N$ for $(N>=2)$, $C_1 = 0$
  計算量: $C_N \sim N\log_2 N$ 、$(O(N\log_2 N))$
  e.g., merge sort
- 1回の操作で入力を2分割する再帰
  $C_N = 2C_{N/2} + 1$ for $(N>=2)$, $C_1 = 0$
  計算量: $C_N \sim 2N$ 、$(N))$
  e.g., scaling a ruler

Algorithms with different recurrence relations

- N operations for deceasing of 1 data size.
  $C_N = C_{N-1} + N$ for $(N>=2)$, $C_1 = 1$
  complexity: $C_N \sim N^2/2$, $(O(N^2))$
  e.g., selection sort
- one operation to reduce the data size to half.
  $C_N = C_{N/2} + 1$ for $(N>=2)$, $C_1 = 0$
  complexity: $C_N \sim \log_2 N$, $(O(\log N))$
  e.g., binary search
- N operations to reduce the data size to half.
  $C_N = C_{N/2} + N$ for $(N>=2)$, $C_1 = 0$
  complexity: $C_N \sim 2N$, $(O(N))$
  e.g.,
- N operations to separate into two sub-problems.
  $C_N = 2C_{N/2} + N$ for $(N>=2)$, $C_1 = 0$
  complexity: $C_N \sim N\log_2 N$, $(O(N\log_2 N))$
  e.g., merge sort
- one operation to separate into two sub-problems.
  $C_N = 2C_{N/2} + 1$ for $(N>=2)$, $C_1 = 0$
  complexity: $C_N \sim 2N$, $(N))$
  e.g., scaling a ruler

アルゴリズムの選択と最適化

- いろんなタイプのアルゴリズム
  - 単純で遅いアルゴリズム
  - 少し複雑で速いアルゴリズム
  - 凝っているが、速さは大して変わらないアルゴリズム
- アルゴリズムの最適化が必要な場合
  - プログラムが繰り返し何度も使われる
  - 入力データのサイズが大きい
  - 払う努力に見合うだけ性能の改良ができる場合
- アルゴリズム開発過程を考える
  - 単純なアルゴリズムで実装し、プログラムのチェックを行い、そして、改良していく。開発コストと実行速度のバランスを考える。
  - 少ない入力で数回しか使われないようなプログラムは単純で遅くてよい。性能にこだわりすぎる必要はない。
  - プログラムの一部がシステムのボトルネックにならないように注意が必要である。

Selection of algorithm and program optimization

- Different types of algorithms
  - simple but slow algorithms
  - complex but fast algorithms
  - very complex but not much faster algorithms
- Cases need optimization
  - Algorithms that will be used many times repeatedly.
  - Algorithms for very large data size.
  - There will be returns match the payed effort.
- Algorithm developing process
  - Firstly implement a simple algorithm, and then check the program and improve. Considering the balance of cost and running time.
  - No need to improve programs for small data size and running for only few times.
  - Care about the bottle-neck programs.