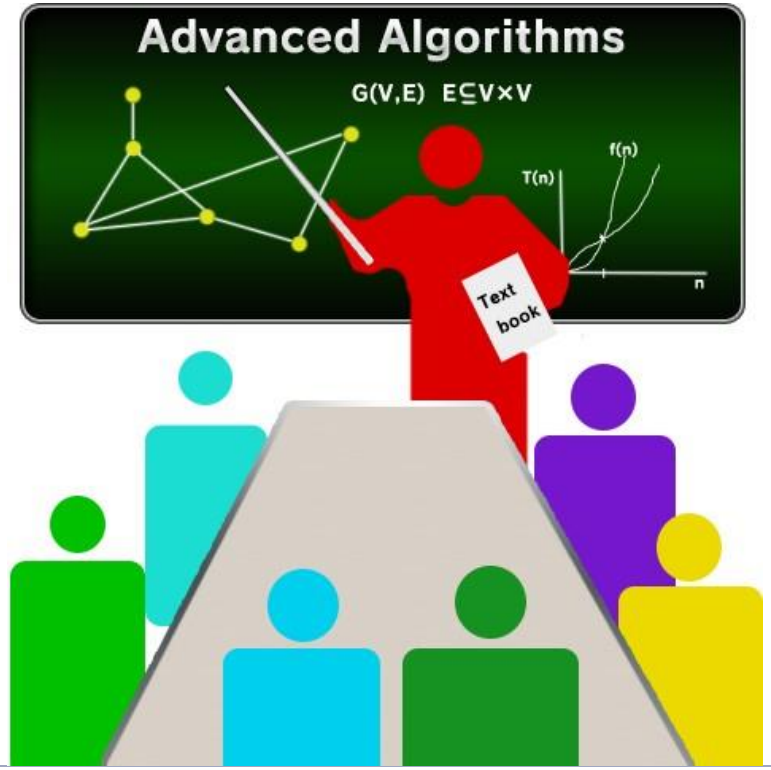


アルゴリズム特論 [AA201X]
Advanced Algorithms

Lecture01. Algorithms and their Complexity

Overall Contents of AA201X

前半

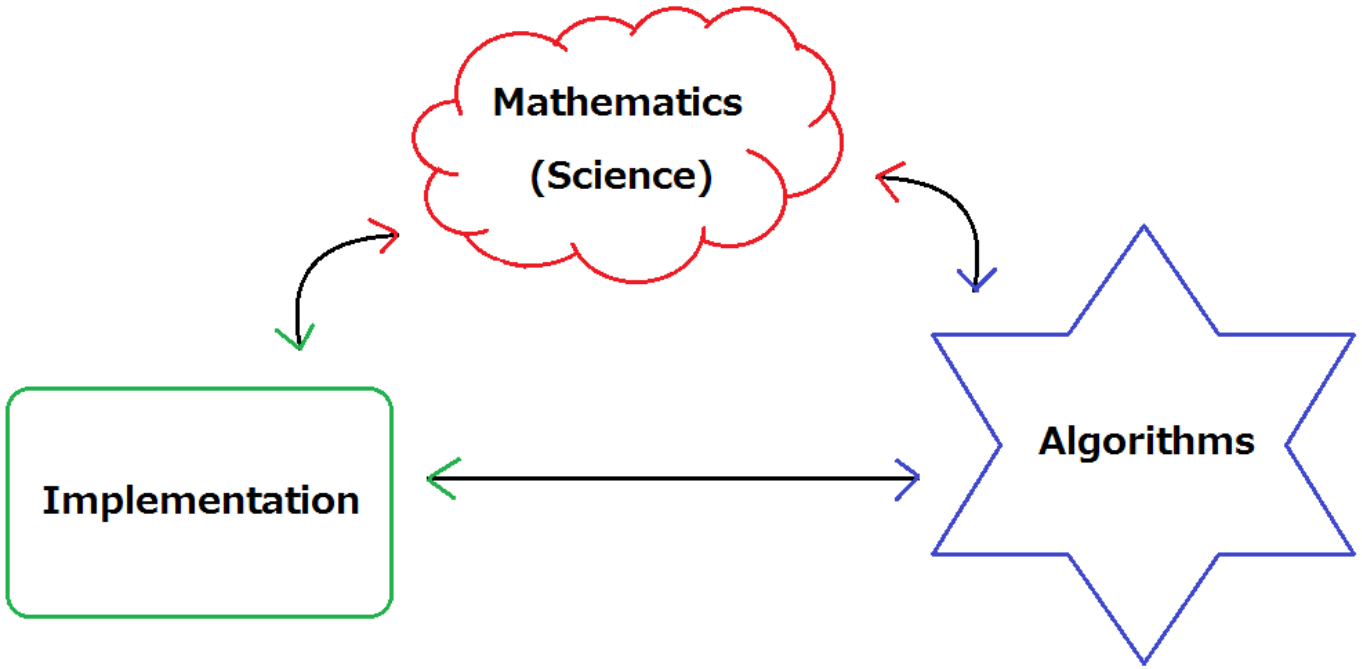
- **時間計算量とその解析**
- **ヒープ構造**
- **グラフ （～代数的経路アルゴリズムまで）**
- **文字列検索**

後半

- **アルゴリズム設計テクニック**
  ⇒ 貪欲法、分割統治法、動的計画法(DP)、バックトラック
- **乱数生成／乱択アルゴリズム**
- **並列処理**

Aim of AA2016

- 書いてある手続きの通りに操作できる
- 一連の手続きをプログラムとして実装できる
- アルゴリズムの構造に注目し、一般化できる



計算機で問題を解くにはアルゴリズムが必要

- アルゴリズム・・・与えられた問題に対する、

あらゆる入力に対して期待される解を得ることが可能な計算の手続き

- ある問題に対するアルゴリズムが、どれくらい効率的（efficiency）であるかを知っている必要がある
- あるアルゴリズムで与えられた問題を解くために、どれくらい時間（time）やメモリ領域（memory space）を要するのかに興味がある

⇒ 計算機資源は有限。いかに近年の計算機の性能が進化してきているとはいえ、それに甘えて poor なコードを作っているようではダメ

計算機で問題を解くにはアルゴリズムが必要

- アルゴリズムの計算時間は、**演算回数に依存**する

- アルゴリズムの計算時間は、**問題サイズに依存**する

⇒ 特に、「問題サイズを変えた（増やしていった）ときに計算時間がどれくらい増大するか」が重要となる

複数のアルゴリズムの効率性を調査・比較するには、単一の計算機・実行環境下で行うのが望ましい

アルゴリズムの計算量を調べる

- 問題サイズは一般に $n \in Z$ を用いて表す。

⇒ アルゴリズムに入力されるデータ数の度合い

行列乗算：演算に使われる最も大きな行列の次元

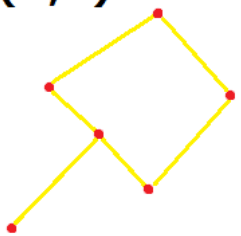
ソート　：ソートされるデータ列の長さ

グラフ　：頂点の集合Vや辺の集合Eの大きさ（要素数）

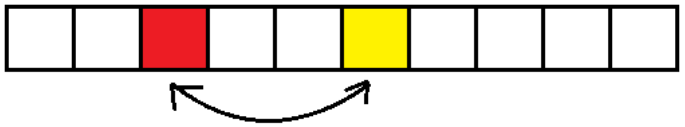
$M_{2,4} \times M_{4,3}$

$$\begin{pmatrix} 1 & 2 & 2 & 1 \\ 0 & 3 & -2 & 4 \end{pmatrix} \begin{pmatrix} 5 & 3 & 1 \\ -1 & 2 & 0 \\ 3 & 1 & 4 \\ 2 & -6 & 7 \end{pmatrix}$$

G(V,E)



Sorting of A[10]



アルゴリズムの計算量を調べる

- アルゴリズムの計算時間は、T(n) = … のように
  問題サイズの関数で表される

- 時間計算量（time complexity）
  ⇒ 計算時間はどれくらいかかるか？
- 空間計算量（space complexity）
  ⇒ 計算に必要なメモリ領域はどれくらいか？

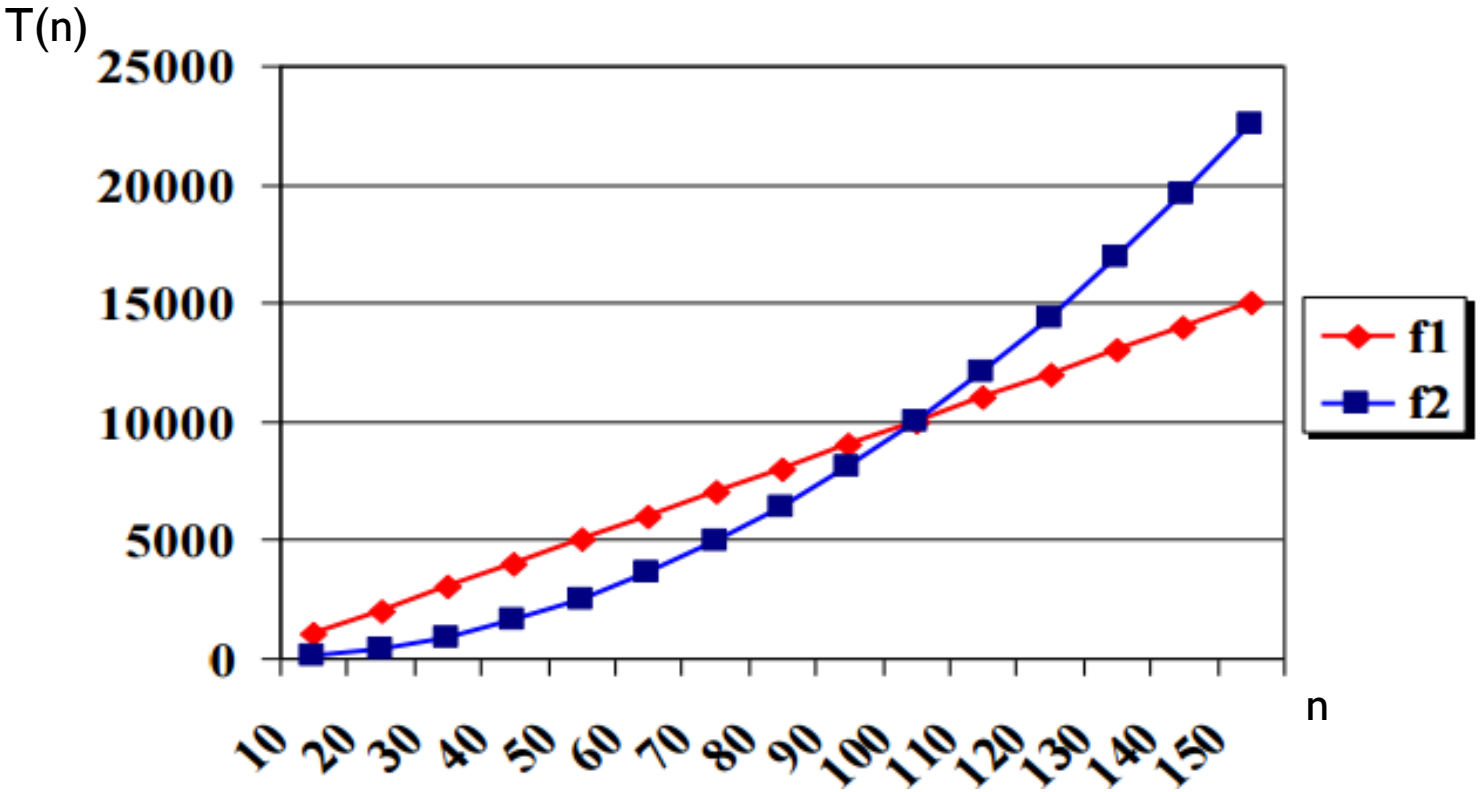
※この授業では、時間計算量を中心にして見ていくことにする

アルゴリズムの計算量を調べる

- 問題サイズ n が与えられたときのアルゴリズムの計算時間を、また n を増加させるとアルゴリズムの時間計算量がどう増加するかを知ることが重要

- 与えられたアルゴリズムの時間（空間）計算量の増加率は、究極的には解く問題の大きさで決まる

アルゴリズムの計算量を調べる



f1 = 100n , f2= n^2 とする。　　n<100までは f1 > f2,　n>100で f2 > f1 に。

f1,f2 をアルゴリズムのデータ数に対する計算回数を示す関数だと思えば、
データが100より大きくなると、f2の方が圧倒的に計算量が増加する

計算量のオーダー

- データ数に応じて、アルゴリズムの計算量がどのくらいの**増加率** (growth rate) になるかを見極める指標

- ランダウ記号

lim[n->∞] にしたときの関数の変化の割合に対して
大凡の評価を与えるための記法

$$O(n) \qquad \Omega(n) \qquad \Theta(n)$$

$$o(n) \qquad \omega(n)$$

これらを用いて、複数のアルゴリズムの**計算量の増加率を相対的に比較**する

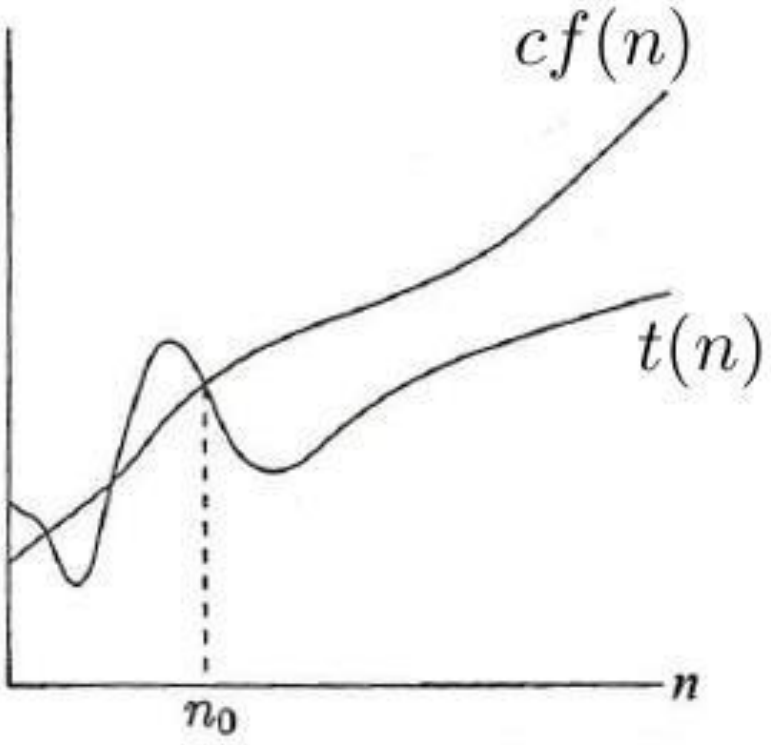
計算量のオーダー $t(n) = O(f(n))$

$$\exists c > 0, n_0 > 0 \text{ s.t. } \forall n \geqq n_0, t(n) \leqq cf(n)$$

t(n)とc倍のf(n)の間の大小関係が固定される
あるn0 以上の n に於いて、
t(n)はcf(n)に上から抑えられている

「t(n) の増加量は高々 f(n)程度」
⇒「 t(n) の増加量はf(n)の増加量と等しいか、
それよりも小さい」

f(n) は t(n) の上界 (upper bound) である



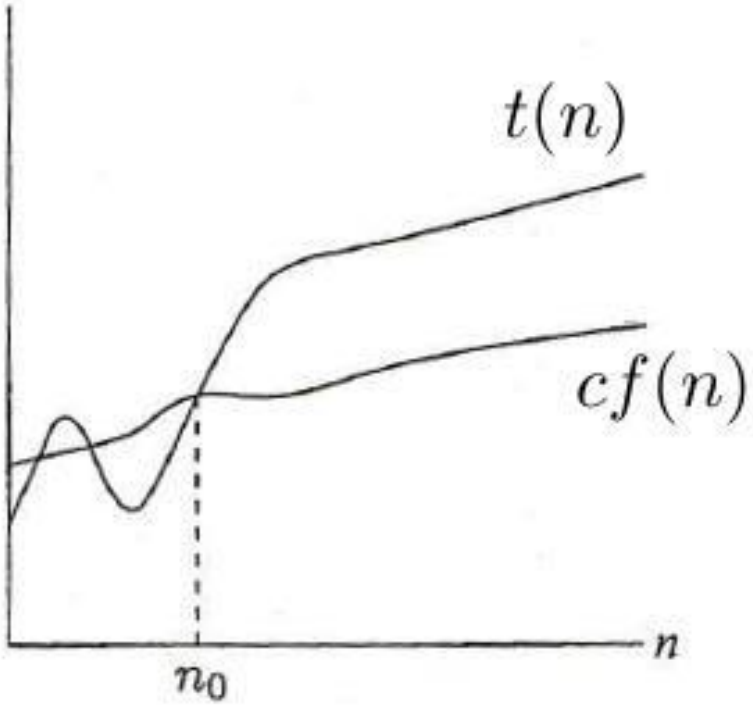
計算量のオーダー $t(n) = \Omega(f(n))$

$$\exists c > 0, n_0 > 0 \text{ s.t. } \forall n \geqq n_0, t(n) \geqq cf(n)$$

t(n)とc倍のf(n)の間の大小関係が固定される
あるn0 以上の n に於いて、
t(n)はcf(n)に下から抑えられている

「t(n) の増加量は少なくとも f(n)程度」
⇒「 t(n) の増加量はf(n)の増加量と等しいか、
それよりも大きい」

f(n) は t(n) の下界 (lower bound) である



計算量のオーダー $t(n) = o(f(n)),\ t(n) = \omega(f(n))$

strictly smaller (greater)

真に小さい（大きい）　・・・
(または「漸近的に」， asymptotically)

O記法,Ω記法の定義で、 = を許さないもの

つまるところ、極限で表すと…

$$t(n) = o(f(n))$$

$$\Leftrightarrow \lim_{n\to\infty} \frac{t(n)}{f(n)} = 0$$

$$t(n) = \omega(f(n))$$

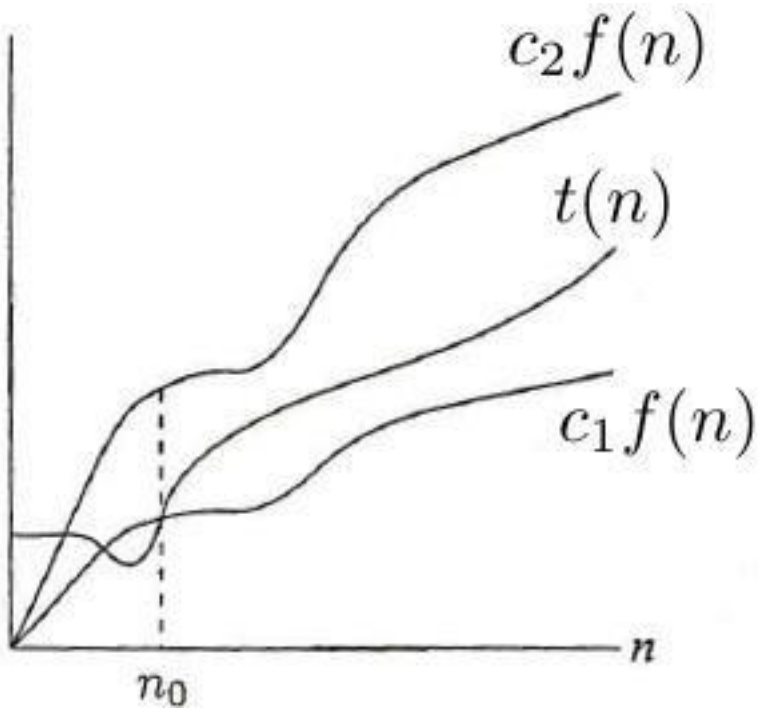
$$\Leftrightarrow \lim_{n\to\infty} \frac{t(n)}{f(n)} = \infty$$

計算量のオーダー $t(n) = \Theta(f(n))$

$$\exists c_1 > 0, c_2 > 0, n_0 > 0 \quad \text{s.t.} \quad \forall n \geqq n_0, \quad c_1 f(n) \leqq t(n) \leqq c_2 f(n)$$

t(n)とc1倍のf(n)、 t(n)とc2倍のf(n)の間の
大小関係が固定される
あるn0 以上の n に於いて、
t(n)は c1f(n) と c2f(n) に挟まれている

⇒「 t(n) の増加量はf(n)の増加量とだいたい
等しい」

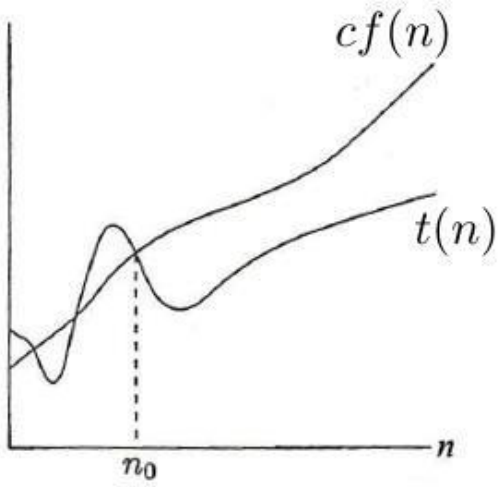


【 O や Ω よりも制限の強い定義 】

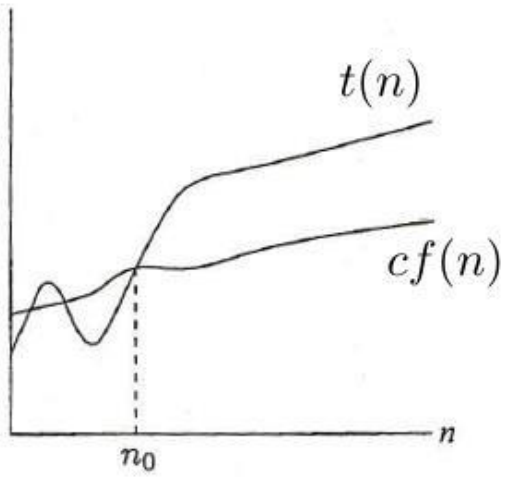
$$t(n) = \Theta(f(n))$$

$\Leftrightarrow$ $t(n) = O(f(n))$ かつ $t(n) = \Omega(f(n))$

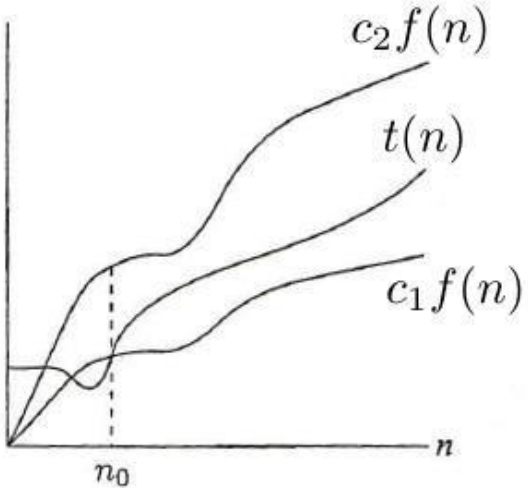
計算量のオーダー　O,Ω,Θを比べると…



(a)　$t(n) = O(f(n))$



(b)　$t(n) = \Omega(f(n))$



(c)　$t(n) = \Theta(f(n))$

※この授業では、最悪時間計算量の意味で主にO記法によって計算量を記述します

ランダウ記法のルール

- O記法は関数の増加率を表すわけだから、

・最高次数以外の項

・項の係数

は無視をして記述する。例えば以下は全てO(n)と書く

$t(n) = O(2n)$　　$t(n) = O(n+\log n)$　　$t(n) = O(n+\sqrt{n})$

- 本当にいいの？　⇒　定義に従って確認すればよい

- $2n = O(n)$ かどうか

$n \geqq n0$ に対して、 $2n \leqq cn$ となる　正の数 $c, n0$ があればよい

$n0 = 1, c = 2$ ととれば、どんな$n \geqq n0$に対しても、上の不等式は成り立つ

ランダウ記法のルール

- とはいえ、ランダウ記号は割と大ざっぱな見積もり

$n = O(n^2)$　は正しい

- $n \leqq c\, n^2$　（$n \geqq n_0$）を満たす c と n_0 を探してみる
- $c = 1, n_0 = 1$ にしてみる　$\Rightarrow$ $n \leqq n^2$
- 成り立った

- 確かに、定義は満たす（n^2の増加量よりはどう考えても小さい）
- でもそんな指標はアルゴリズムの評価としては役に立たない
- なるべく定義を満たすギリギリの関数で書く

ランダウ記法のルール

- ところで、 t(n) = O(f(n)) は気持ちが悪い？

- 数学で a ＝ b と書くと、＝ は a と b は全く同じもの同士であることを示す **同値関係**

《二項関係 ＝ が同値関係である》　⇔　以下の３つの性質を満たす

反射律　a ＝ a

対称律　a ＝ b　⇒　b = a

推移律　a ＝ b, b ＝ c　⇒　a ＝ c

- ランダウ記号に等号を使うと、これを満たさない
- O(n) = n = 2n = n + log n = n + 10000

- 本質的には、O(n) は 定義に従ってオーダーを取ると f(n) = n となる時間計算量を示す関数 t(n) たちの **集合** である。　n ∈ O(n),　n + log n ∈ O(n)

- しかし、記述やその意味がややこしくなるので、便宜上 ＝ を用いることが慣習となっている

ランダウ記法上の演算

- （便宜上の）加法と乗法

$$O(f(n)) + O(g(n)) = \max(O(f(n)), O(g(n)))$$

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

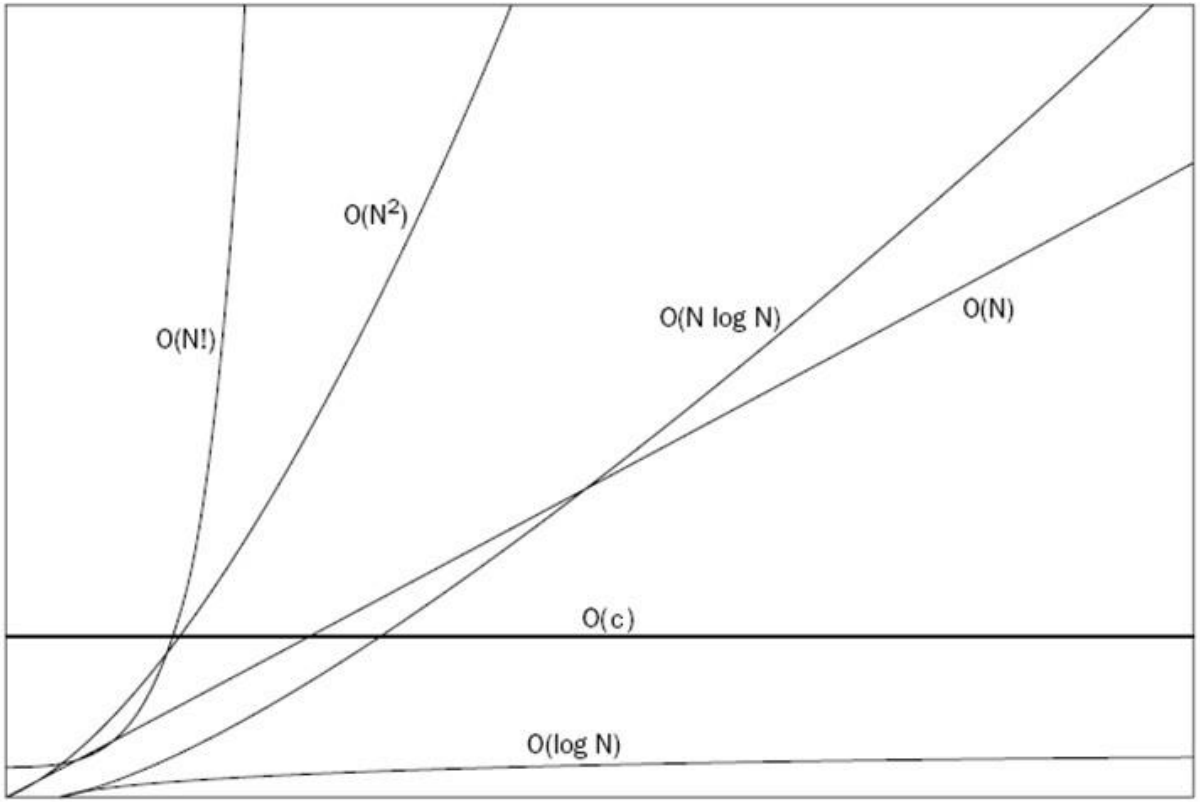
加法の場合： f(n) という量の計算をして、g(n)という量の計算をする
⇒ 全体で計算量を見るときは、オーダーの大きい方を選ぶ

乗法の場合： f(n)という量の計算をする度に g(n) という量の計算を行う
⇒ 計算量の積をとってからオーダーを求める

※いわゆる 多重ループ処理 はこれに当てはまる

主な計算増加量の種類と比較

- 計算量の関数をグラフで表すと、ある程度増加量の大小関係が直感的になる
- ただし、n≧n0 なる n で比較すること



- c　定数(constant)
- log n　対数(Logarithmic)
- n　線形(Linear)
- nlog n
- n^2　Quadratic　(読み：n square)
- n^3　Cubic
- 2^n　指数 Exponential　(読み：two-to the power n)

主な計算増加量の種類と比較

- 直感的に分かりづらい関数の比較
- 今すぐにグラフを書けないとき　⇒　計算しよう

$f(n)$ と $g(n)$ では、どちらの増加量が大きいか？

$$\lim_{n\to\infty}\frac{g(n)}{f(n)}=\infty$$ $g(n)$ の方が大きい

$$\lim_{n\to\infty}\frac{g(n)}{f(n)}=0$$ $f(n)$ の方が大きい

もし極限をとった結果、0より大きく、無限大未満の定数になれば
どちらも同程度の増加量

主な計算増加量の種類と比較

$t_1(n) = \log n$ と $t_2(n) = n$ では、どちらの増加量が大きいか？

$\lim_{n\to\infty} \dfrac{n}{\log n}$ を調べればよい．

∞/∞ の不定形であるから，ロピタルの定理より

$$\lim_{n\to\infty} \frac{n}{\log n} = \lim_{n\to\infty} \frac{1}{\frac{1}{n}} = \lim_{n\to\infty} n = \infty$$

よって、$t_2(n) = n$ の増加量の方が大きい

ロピタルの定理

$\lim_{n\to a} \dfrac{g(x)}{f(x)}$ が $0/0$ や ∞/∞ の不定形であって，

$\lim_{n\to a} \dfrac{g'(x)}{f'(x)}$ が存在するならば，$\lim_{n\to a} \dfrac{g(x)}{f(x)} = \lim_{n\to a} \dfrac{g'(x)}{f'(x)}$

計算量によるアルゴリズム比較（スループット編）

- 1秒間、1分間、1時間以内で、
  ある時間計算量のアルゴリズムが解ける問題サイズの上限

Algorithm	Time complexity	Maximum problem size		
		1 sec	1 min	1 hour
A_1	n	1000	6×10^4	3.6×10^6
A_2	$n\log n$	140	4893	2.0×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

※ 使用する計算機は、１秒間に1000回計算できる性能のものとする
※ logの底は2

計算量によるアルゴリズム比較（スループット編）

- 同じ条件下で、もしコンピュータの性能が１０倍のものに
  なったらどう変わるか？

Algorithm	Time complexity	Maximum problem size	
		before speed-up	size after speed-up
A_1	n	s_1	$10s_1$
A_2	$n\log n$	s_2	$\sim 10s_2$ for large s_2
A_3	n^2	s_3	$3.16s_3$
A_4	n^3	s_4	$2.15s_4$
A_5	2^n	s_5	$s_5+3.3$

計算量によるアルゴリズム比較（実行時間編）

- 1000000 個のデータに対する演算数と時間の関係

Algorithm	Time complexity	# of Operations	Time at 10^6 op/s
A_1	$O(1)$	1	1 μs
A_2	$O(n)$	10^6	1 sec
A_3	$O(n^2)$	10^{12}	11.6 days
A_4	$O(n^3)$	10^{18}	32,000 yrs.
A_5	$O(2^n)$	$10^{301,030}$	$10^{301,006}$ times the age of the universe

プログラムの計算量（計算時間）の解析

- 繰り返し処理 (for / while) の扱い

```
for (i = 1;i < = m;i++)   S;
```

- ‘S’ という処理に $t_i(n)$ という時間がかかるとする
- for文により、 m回の繰り返し処理になるため
  全体では、 $\sum_{i=1}^{m} t_i(n)$ の時間がかかる。

- もし、 ループカウンタ i の値に関わらず、S の処理が一定の時間 t(n) でできるなら、全体の時間は mt(n)となる

プログラムの計算量（計算時間）の解析

- **連続する文の扱い**
- 2つの処理がそれぞれ $t_1(n), t_2(n)$ の時間がかかるものとする。

2つの処理が続けて行われる場合、全体の計算時間は

$$t_1(n) + t_2(n)$$

- **条件分岐** (if / else) **の扱い**

```
if( condition ) S1;
else S2;
```

- 分岐後の処理 S1,S2 がそれぞれ $t_1(n), t_2(n)$ の時間がかかるものとする。
  このときの条件分岐全体の計算時間は

$$\max(t_1(n), t_2(n))$$

⇒ 処理時間のかかる方を採用する（最悪の場合を考える）

Selection Sort

- n個のデータが1次元配列に格納されている
- 配列を前から順に走査し、最も小さい要素を配列の先頭の要素と交換する
- 同様に走査し、２番目に最も小さい要素を配列の２番目の要素と入れ替える
- 同様に繰り返し、i 番目に小さい要素を配列の i 番目の要素と入れ替える

Selection Sort

```
selection(int A[], int n){
  int i, j, min, t;
  for (i=1;i<n;i++){ //外のループ（ n 回）
    min=i;
    for (j=i+1; j<=n; j++) //先頭が一番小さいと仮定し、
      if (A[j]< A[min]) min=j;その先の要素にそれを
      t=A[min];                上回る最小の値が
      A[min]=A[i];              無いか調べる
      A[i]=t;                内側のループ（　？　回）
  }
}
```

１要素のアクセス、代入操作の時間計算量は O(1) 、つまり定数時間とみなせる。
全体の計算回数は（　？　）回になるので、$O(n^2)$ となる。

Merge Sort

- 2つのソート済み配列 A[1],A[2],…,A[m] と B[1], …, B[n] があるとする
- 配列AとBを結合してソートされた配列を作るには、

第３の配列 C[1],C[2],…,C[m+n] を用意する

- AまたはBの配列から最小の値を取り出しC[1]に格納
- AまたはBの配列から２番目に最小の値を取り出しC[2]に格納
- 同様にして、AまたはBの配列から i 番目に最小の値を取り出しC[i]に格納
- AとBの配列の要素が全てCに移されるまで繰り返す

Merge Sort

- この通りに実装すると

```
i=1;j=1;
A[m+1]=max; B[n+1]=max;
for (k=1; k<=m+n; k++)
  if (A[i] < B[j]) C[k]=A[i++];
   else C[k]=B[j++];
```

- m-by-n merging と呼ばれる
- 計算量は O(m+n)

Merge Sort (Recursive ver.)

- 再帰を用いて実装
- n/2の2つの配列に再帰的に分割して、求めた部分解の結合を繰り返すことで、全体の解を得る (Divide and Conquer)

```
mergesort(int A[],int left,int right){
    int i,j,k,mid;
    if (right>left) {
    mid=(right+left)/2;
    mergesort(A,left,mid);
    mergesort(A,mid+1,right);
    for (i=left;i<=mid;i++) B[i]=A[i];
  for (i=mid+1,j=right;i<=right;i++,j--)B[i]=A[j];
  i=left;j=right;
  for (k = left; k <= right; k++)
  if (B[i]<B[j]) A[k]=B[i++]; else A[k]=B[j--];
   }
  }
```

Merge Sort (Recursive ver.)

- 再帰を使う場合、１回の関数呼び出しで２回の再帰呼び出しが発生する
- 再帰のたびに、ソートする配列をn/2の大きさに切る
- 再帰呼び出しの後の処理の計算量は O(n)

- 時間計算量を考えると

  t(n) = 2t (n/2) + cn　　(c : 定数)

  n = 2のとき、 t(2) = O(1) である。

これを解くと、オーダーは　t(n) = O(n log n)

Master Theorem

再帰関数の時間計算量を示す漸化式からオーダーを求める最強の公式！

$T(n)$ は、以下を満たす単調増加関数であるとする。（ただし，$a \geq 1, b \geq 2, c > 0$）

$$T(n) = aT(\tfrac{n}{b}) + f(n)$$
$$T(1) = c$$

ここで、$d \geq 0$ に対して $f(n) \in \Theta(n^d)$ であるならば，
このアルゴリズムの時間計算量は以下で表される。

$$\begin{aligned}
&\text{Case 1: } a < b^d \text{ のとき} \quad &\Rightarrow \quad &T(n) = \Theta(n^d) \\
&\text{Case 2: } a = b^d \text{ のとき} \quad &\Rightarrow \quad &T(n) = \Theta(n^d \log n) \\
&\text{Case 3: } a > b^d \text{ のとき} \quad &\Rightarrow \quad &T(n) = \Theta(n^{\log_b a})
\end{aligned}$$

a：1 回の再帰呼び出しで分割された部分問題の個数

n/b：1 回の再帰呼び出しで分割された部分問題が扱うデータの個数

$f(n)$：再帰呼び出しにおける、部分問題の結果を結合するために必要な時間
（再帰処理に関わるオーバーヘッド）

Master Theorem

- $t(n) = 2t\,(n/2) + cn$

MasterTheorem と係数比較すると

$a = 2,\ b = 2,\ f(n) = cn$

また、 $\Theta(f(n)) = \Theta(n)$ より $d = 1$

$2 = 2^1$ より、 Case 2を適用すると

$T(n) = \Theta(n \log n)$

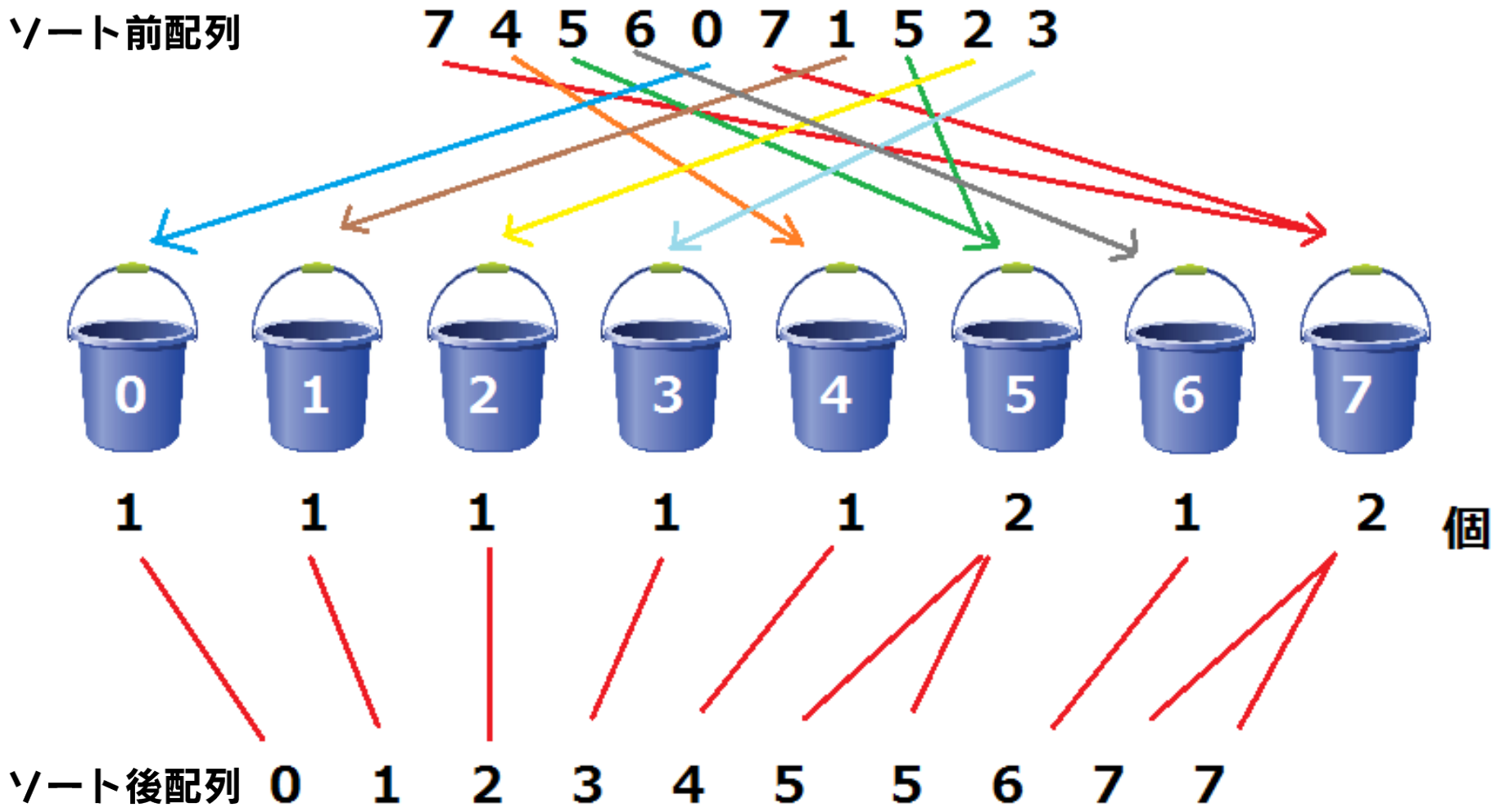
Bucket Sort

- 0からmの範囲の値をとる n 個のデータがあるとする
- m個のバケツ（入れ物）を用意する
- バケツは、順に 0 から m までの番号を付けておく

- n個のデータを順に1個ずつ取り出して、そのデータの値が a であれば、 a 番目のバケツにそれを入れる

- 番号の小さいバケツから順に、バケツに入っているデータを全て取り出して並べる

Bucket Sort



Bucket Sort

- 直感的には、線形的な走査しかしていない
- O(n)　ものすごく早いソート？

- ただし、制約が強い

- そもそも、値が int 型である必要がある
  ⇒配列（バケツ）の添字に小数は使えないため

- データの値の範囲分だけバケツ（配列）が必要
  ⇒メモリを大量に消費（空間計算量は肥大化）
  ⇒値が実数全体となると、2147483647 × 2 ≒ 43億 個
  ⇒そして大抵、未使用のバケツがたくさん出る

ランダウ記号による評価はあくまで指標

- オーダーが低い程、理論的には優れたアルゴリズムであるが、**実装時に必ず良い性能が得られるとは限らない**

⇒ オーダーを取るときは定数を完全に無視しているが、実際には処理時間に影響してくることも

⇒ 入力データの内容によって処理時間が変動する可能性あり

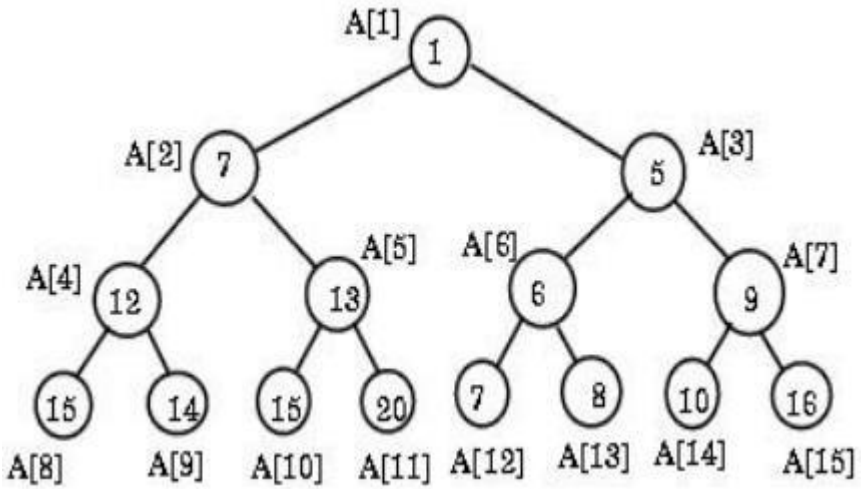
- **アルゴリズムのオーダーの高低、実用性だけに拘らない**

⇒ 当時は画期的アイデア、後の改良版や、続く新しいアルゴリズムの**基礎的な考え方として影響を与えたもの**

⇒ 扱うデータの型（**集合**）と**演算**をうまく定義することで、同型なコードだが**様々な問題に適用できる普遍な書き方、考え方**

次回予告 (Lecture. 2: Priority Queues)

- ソートに関する補足
- ヒープ構造



《キーワード》

- 二分木
- 順序関係

毎年、なぜか大量脱落する単元
⇒　本質的でない部分に捉われすぎている
⇒　大事なことは「ヒープ構造とは何か？」（どういう実装の道具か？）
⇒　あとはそれをどう使うかのみ