

情報構造 第四回

良いアルゴリズムとは

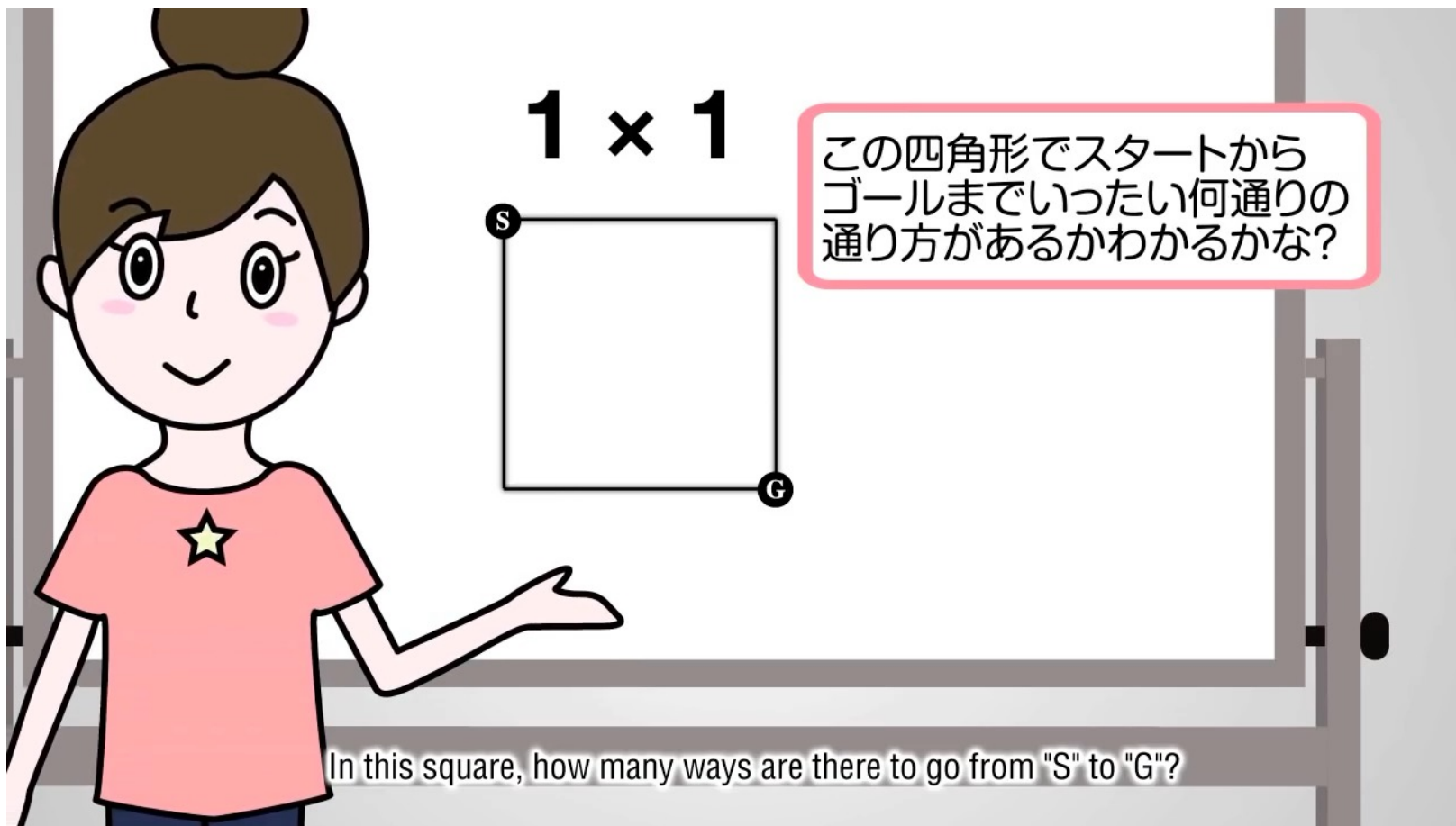
おまけ

JST ERATO 港離散構造処理系プロジェクト

YouTube MiraikanChannel

『フカシギの数え方』 おねえさんといっしょ！ みんなで数えてみよう！

<https://youtu.be/Q4gTV4r0zRs>



本日の内容：

- アルゴリズム
 - べき乗 a^n の計算
- 計算量
 - ランダムアクセス機械
 - 漸近的計算量（おおよその計算量）
 - n^n の計算量
 - バブルソート，階乗計算の計算量

アルゴリズム

- ある目的のために，どのような手続きを表現するかを記述した手順書
- 問題を解くための機械的操作からなる，有限の手続き
 - 機械的操作：四則演算やジャンプなど有限個の命令の集まり
 - 有限の手続き：有限の時間で必ず停止する
- 数列の極限 $\lim_{n \rightarrow \infty} a_n$ は停止しないので命令ではない

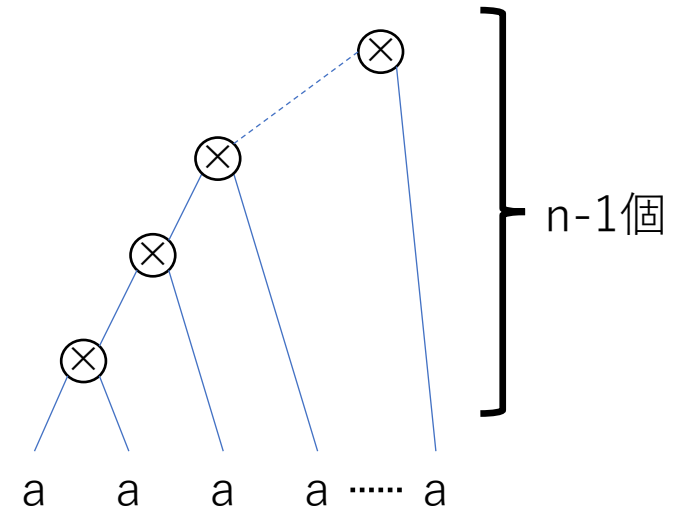
アルゴリズムの例：べき乗計算

- べき乗計算 (Ocamlの一回目の授業の演習で解いた?)

- 入力：整数値 a と n
- 出力： a^n

- $a^n = ((\dots (((a \times a) \times a) \times a) \times \dots) \times a)$

- 一つずつかけていく
- $n - 1$ 回のかけ算



アルゴリズムの例：べき乗計算（改良）

- n が偶数の時

- $a^n = a^{2(\frac{n}{2})} = (a^2)^{n/2}$

- n が奇数の時

- $a^n = a \times a^{2(\frac{n-1}{2})} = a \times (a^2)^{\frac{n-1}{2}}$

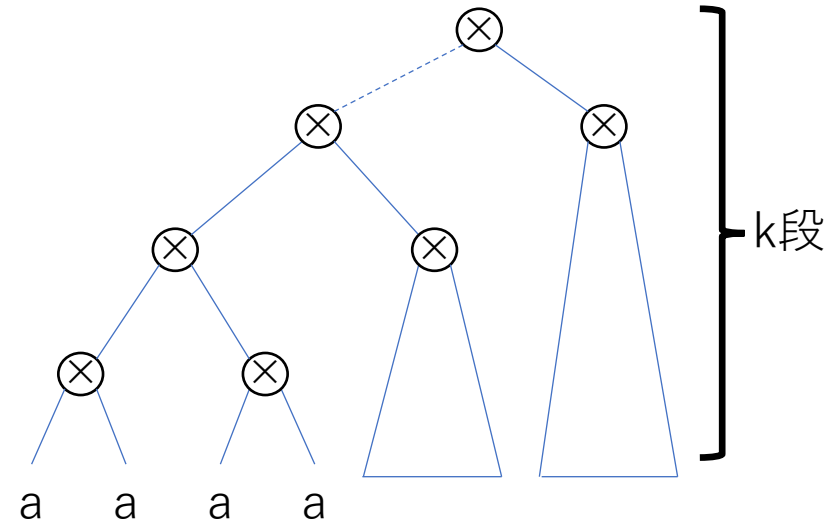
- さらに

- $n/2$ や $\frac{n-1}{2}$ が偶数または奇数でさらに式変形ができる

- $a^n = a^{2(\frac{n}{2})} = (a^2)^{n/2} = ((a^2)^2)^{n/4} = (((a^2)^2)^2)^{n/8} = \dots$

- 基本的に $n = 2^k$ に依存して計算回数変動する

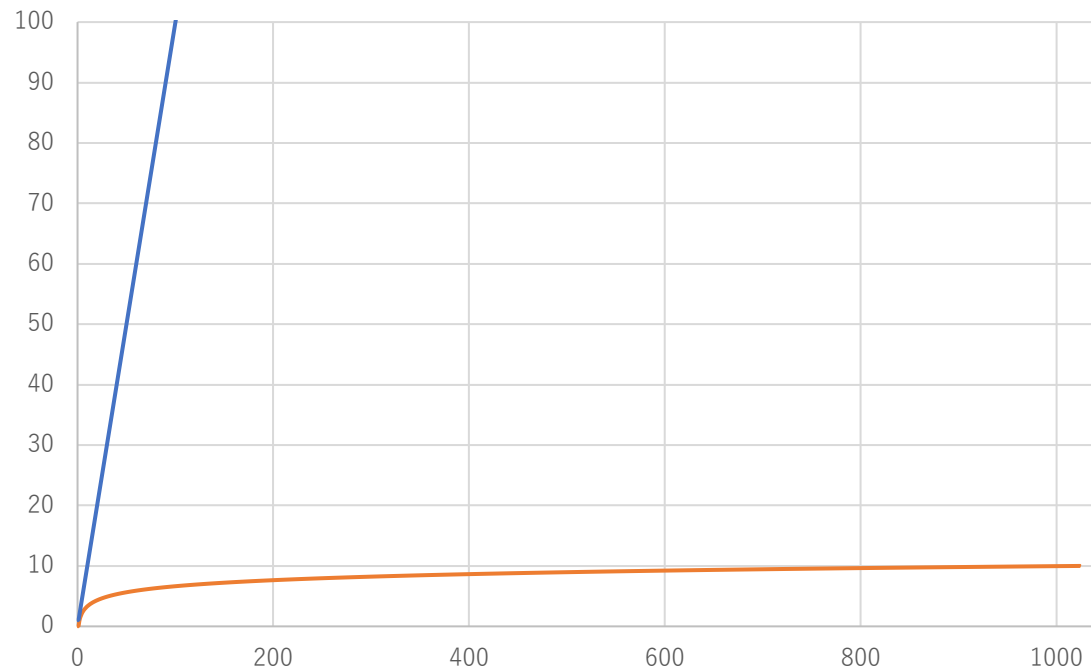
- つまりおおよそ $k = \log_2 n$ 回で計算可能になる



アルゴリズムの重要性

- $n = 1024$ のときのかけ算の回数

- 一つずつかける 1023回 (およそ n 回)
- 改良 10回 (およそ $\log n$ 回)



発散の仕方が
まったく異なる

アルゴリズムの選択は
重要！！

アルゴリズムの効率：計算量

- アルゴリズムの効率を決める尺度
 - 時間計算量 (time complexity) : 計算に要する時間
 - 領域計算量 (space complexity) : 使われるデータの大きさ
- 入力サイズに依存
 - 値の大きさ：整数のような原子データ型
 - 例えば, 階乗計算
 - 要素数：配列のような要素をもつデータ構造
 - 例えば：並び替え (ソート)

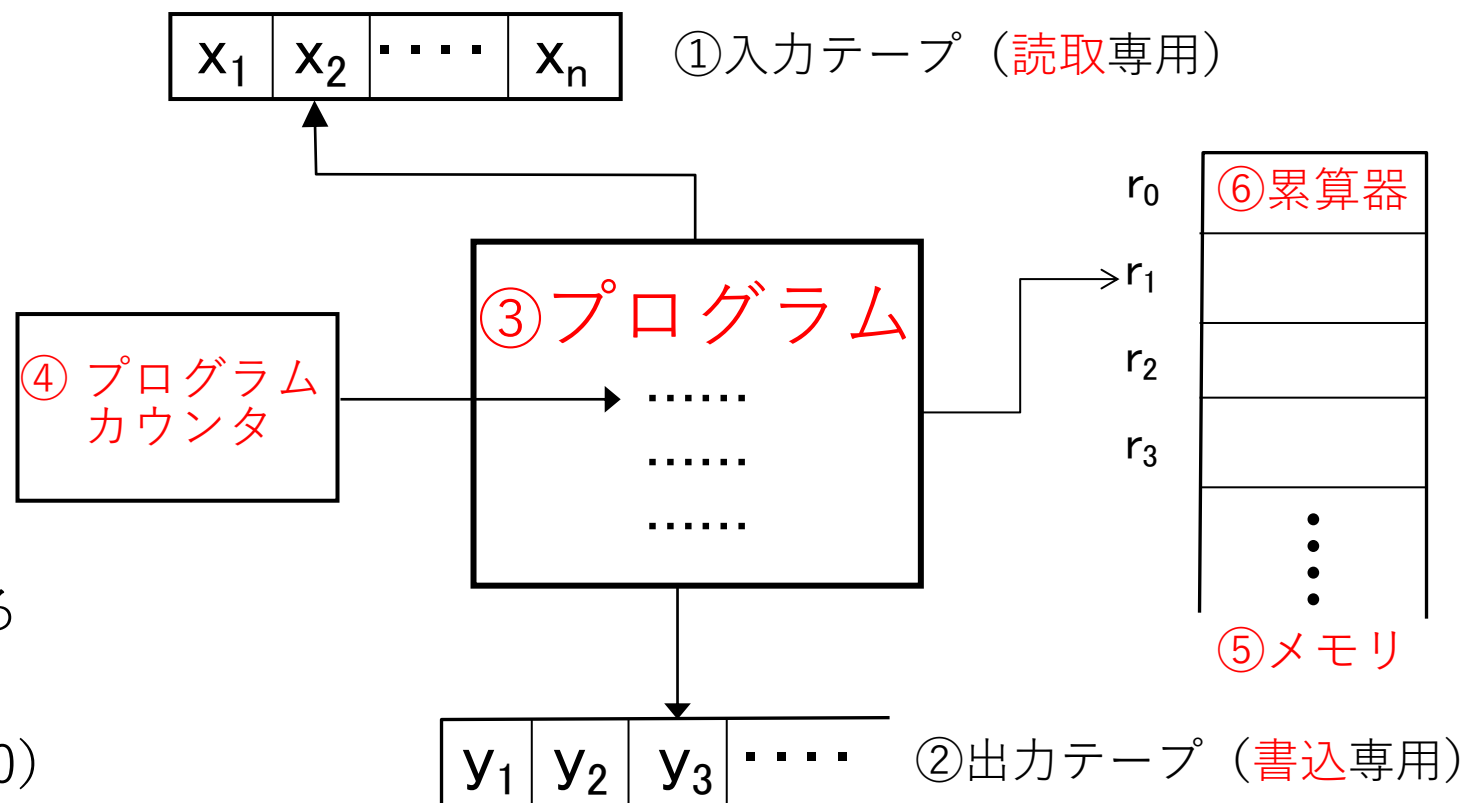
計算機のモデル

- プログラミング言語やハードウェアに依存しない議論がしたい
⇒ 計算機のモデルをつくり，ステップを数える
- 計算機のモデル
 - 例：
 - チューリング機械，ランダムアクセス機械（RAM），
λ 計算，帰納的関数など
 - 計算可能性について議論
 - アルゴリズムを一律に比較

ランダムアクセス機械 RAM

- 現在のコンピュータに近いモデル
- 構成

- ①入力テープ
 - ②出力テープ
 - ③プログラム
 - 命令の列
 - ④プログラムカウンタ
 - 現在命令中の命令
 - ⑤メモリ
 - 任意の大きさの整数を蓄える
 - ⑥累算器
 - 演算結果を累積（メモリの r_0 ）



ランダムアクセス機械のプログラム

- プログラム内の命令（単位時間で実行されると過程）
 - 算術命令：
 - 四則演算（累算器 r0 を介して行う）
 - 入出力命令：
 - 読み取り，書き込み（入力テープをそれぞれ右へ1つ進める）
 - 分岐命令：
 - ジャンプ，条件分岐，停止

RAMプログラム		
	READ	1
	LOAD	1
	JGTZ	pos
	WRITE	=0
	JUMP	endif
pos:	LOAD	1
	STORE	2
	SUB	=1
	STORE	3
while:	LOAD	3
	JGTZ	continue
	JUMP	endwhile
continue:	LOAD	2
	MULT	1
	STORE	2
	LOAD	3
	SUB	=1
	STORE	3
	JUMP	while
endwhile:	WRITE	2
endif:	HALT	

プログラムの例

プログラムの命令

- 命令の記述法

書き方： 命令コード オペランド（被演算子）
例： LOAD a

- オペランド a：算術・入出力命令（3種類の書き方）

=i 整数 i そのもの（リテラル）
i 第 i レジスタ（番地）の内容（ポインタ）
*i 間接番地：第 i レジスタの内容 j の，第 j レジスタの内容（ポインタのポインタ）

- オペランド a の値 $v(a)$ の定義（ $c(i)$ は第 i レジスタの整数）

$v(=i) = i$ オペランドの値 i そのものを表す
 $v(i) = c(i)$ 第 i レジスタの持つ値を表す
 $v(*i) = c(c(i))$ 第 i レジスタの持つ値の間接番地の値

- オペランド b：分岐命令の場所を表すラベル（このラベルは命令コードの前に記述）

JUMP b
...

b: LOAD 1

RAM プログラム

	READ	1
	LOAD	1
	JGTZ	pos
	WRITE	=0
	JUMP	endif
pos:	LOAD	1
	STORE	2
	SUB	=1
	STORE	3
while:	LOAD	3
	JGTZ	continue
	JUMP	endif
continue:	LOAD	2
	MULT	1
	STORE	2
	LOAD	3
	SUB	=1
	STORE	3
	JUMP	while
endif:	WRITE	2
	HALT	

プログラムの例

代表的な命令とその意味

命令コード

LOAD

STORE

STORE

ADD

MULT

SUB

READ

READ

WRITE

JUMP

JGTZ

HALT

オペランド

a

i

$*i$

a

a

a

i

$*i$

a

b

b

意味

0は累算器

$v(a) \rightarrow c(0)$

$c(0) \rightarrow c(i)$

$c(0) \rightarrow c(c(i))$

$c(0) + v(a) \rightarrow c(0)$

$c(0) * v(a) \rightarrow c(0)$

$c(a) - v(a) \rightarrow c(0)$

入力値 (入力ヘッドの値) $\rightarrow c(i)$

入力値 $\rightarrow c(c(i))$

$v(a) \rightarrow$ 出力ヘッド下のセル

b の表す番地 \rightarrow プログラムカウンタ

$c(0) > 0$ のとき、 $b \rightarrow$ プログラムカウンタ

プログラムの実行停止

$c(i)$ は第 i レジスタの値

$v(=i) = i$ オペランドの値 i そのもの

$v(i) = c(i)$ 第 i レジスタの持つ値

$v(*i) = c(c(i))$ 第 i レジスタの持つ値の間接番地の値

$a = \begin{cases} =i \\ i \\ *i \end{cases}$



命令の例：LOAD

- 命令

命令コード	オペランド	意味
LOAD	a	$v(a) \rightarrow c(0)$
LOAD	=3	r_0 に 3 をセット
LOAD	3	r_0 に 2 をセット
LOAD	<u>*3</u> <u>2</u> =1	r_0 に 1 をセット

r_0	
r_1	5
r_2	1
r_3	2
	⋮

=i	整数 i そのものを表す（リテラル）
i	第 i レジスタの内容を表す
*i	間接番地：第 i レジスタの内容が j のとき，第 j レジスタの内容

命令の例：STORE

- 命令

命令コード	オペランド	意味
STORE	i	$c(0) \rightarrow c(i)$
STORE	*i	$c(0) \rightarrow c(c(i))$
STORE	3	7 を r_3 にセット
STORE	*3	7 を r_2 にセット
STORE	=3 ×	あり得ない！

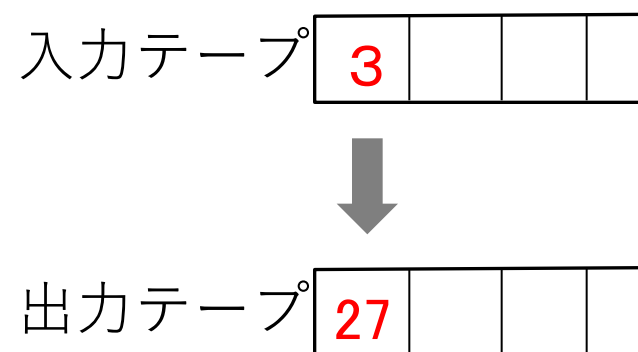
r_0	7
r_1	5
r_2	1
r_3	2
	⋮

=i	整数 i そのものを表す（リテラル）
i	第 i レジスタの内容を表す
*i	間接番地：第 i レジスタの内容が j のとき、第 j レジスタの内容



RAMプログラム例： $f(n) = n^n$

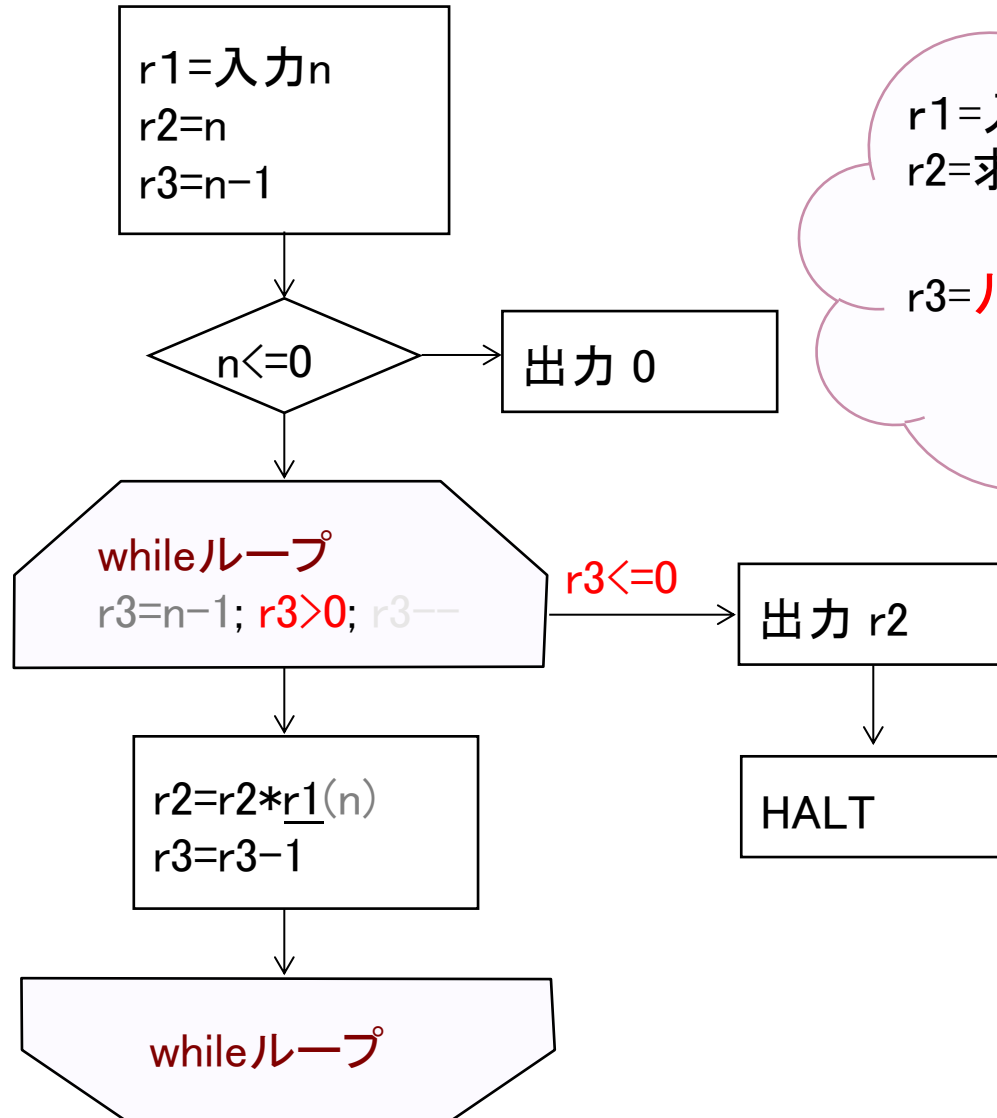
$$\begin{array}{ll} f(n) = n^n & \dots n > 0 \\ 0 & \dots \text{その他} \end{array}$$



- 分かり易さのために、RAMプログラムの右側に同じ機能のCプログラム, 日本語説明を記す

n^n のフローチャート

```
#include <stdio.h>
int r1, r2, r3;
int main(void){
    scanf("%d", &r1);
    r2 = r1;
    r3 = r1-1;
    if(r1<=0){
        printf("%d¥n", 0);
    }else{
        while(r3>0){
            r2 *= r1;
            r3 -= 1;
        }
        printf("%d¥n", r2);
    }
    return 0;
}
```



r1=入力n
r2=求める値 n^{i+1}
($i=1 \sim n-1$)
r3=ループカウンタ($n-i$)

RAMプログラム

```
READ      1
LOAD      1
JGTZ      pos
WRITE     =0
JUMP      endif
pos: LOAD   1
STORE     2
SUB       =1
STORE     3
while: LOAD 3
JGTZ      continue
JUMP      endwhile
continue: LOAD 2
MULT      1
STORE     2
LOAD      3
SUB       =1
STORE     3
JUMP      while
endwhile: WRITE 2
endif:    HALT
```

対応するC言語風表現

```
scanf("%d", &r1);

if (r1<=0)
    printf("%d",0);

else
    r2=r1;

    r3=r1-1;

while (r3>0) {
    r2=r2*r1;

    r3=r3-1;
}

printf("%d", r2);
};
```

日本語説明

入力テープの値をr1にセット
r1の値をr0にセット
r0の値が0より大きいとき posラベルへジャンプ
0を出力テープにセット
endifラベルへジャンプ
r1の値をr0にセット
r0の値をr2をにセット
r0の値から1を引いてr0にセット
r0の値をr3にセット
r3の値をr0にセット
r0が0より大きいとき continueへジャンプ
endwhileラベルへへジャンプ
r2の値をr0にセット
r0の値にr1の値をかけてr0にセット
r0の値をr2にセット
r3の値をr0にセット
r0の値から1を引いてr0にセット
r0の値をr3にセット
whileラベルへジャンプ
r0の値を出力テープにセット
プログラム実行停止

メモリ

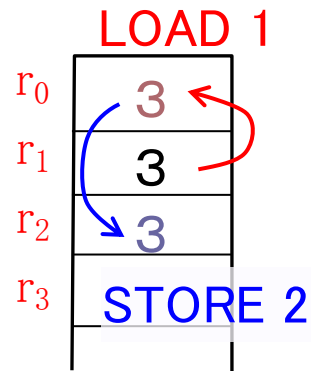
r0	累算器
r1	
r2	
r3	
	⋮
	•



$f(n) = n^n$: 初期設定

	READ	1	<code>scanf("%d", &r1);</code>	
	LOAD	1		
	JGTZ	pos		
	WRITE	=0	<code>if (r1<=0) printf("%d",0);</code>	
	JUMP	endif		
pos:	LOAD	1	<code>else {</code>	
	STORE	2	<code>r2=r1;</code>	
	SUB	=1		
	STORE	3	<code>r3=r1-1;</code>	

入力が0以下ならば、
0を出力して終了



$r_1:3$ 入力値
 $r_2:3^1$ 求める値
 $r_3:2(3-1)$ ループ

----- ここからループ -----

while: LOAD 3

$r_3:2(3-1)$ カウンタ

$r_3:2$ のループの始まり

$f(n) = n^n$: ループ1回目

while:	LOAD	3	-----	$r_3:2$ のループの始まり
	JGTZ	continue		$r_1:3$ 入力値
	JUMP	endwhile	while ($r_3 > 0$)	$r_2:3^1$ 求める値
continue:	LOAD	2	{	$r_3:2(3-1)$ 最初のループ カウンタ値
	MULT	1		
	STORE	2	$r_2 = r_2 * r_1;$	
	LOAD	3		$r_1:3$ 入力値
	SUB	=1		$r_2:9(3^2)$ 求める値
	STORE	3	$r_3 = r_3 - 1;$	$r_3:1(3-2)$ 次の カウンタ値
	JUMP	while	}-----	ループの最後 $r_3:1$

$f(n) = n^n$: ループ2回目

while:	LOAD	3	-----	$r_3:1$ のループの始まり
	JGTZ	continue		$r_1:3$ 入力値
	JUMP	endwhile	while ($r_3 > 0$)	$r_2:9(3^2)$ 求める値
			{	$r_3:1(3-2)$ ループ
continue:	LOAD	2		カウンタ値
	MULT	1		
	STORE	2	$r_2 = r_2 * r_1;$	
	LOAD	3		$r_1:3$ 入力値
	SUB	=1		$r_2:27(3^3)$ 求める値
	STORE	3	$r_3 = r_3 - 1;$	$r_3:0(3-3)$ 次の
	JUMP	while	}-----	カウンタ値
				ループの最後 $r_3:0$

$f(n) = n^n$: ループ3回目-最終

while: LOAD 3 -----
 JGTZ continue
 JUMP endwhile while (r3>0)
---- r₃=0 よりループ終了 : endwhileにとぶ----

continue: LOAD 2 {
 MULT 1
 STORE 2 r2=r2*r1;
 LOAD 3
 SUB =1
 STORE 3 r3=r3-1;
 JUMP while }

r₃:0のループの始まり
r₁:3 入力値
r₂:27(3³) 求める値
r₃:0(3-3) ループ
 カウンタ値



RAMプログラムの計算量

RAMプログラムの計算量



二つの計算量

- **時間**計算量の「実行時間の**単位**」: 命令ステップの時間
- **領域**計算量の「領域の**単位**」: レジスタの使用量

二つの計算量基準

- 一様コスト基準
- 対数コスト基準

一様コスト基準

- **一様コスト基準** (uniform cost criterion)
 - すべての演算において一定のコストを割り当てる
- 時間計算量
 - **どの**RAM命令も**1**単位の時間で実行 
- 領域計算量
 - **どの**レジスタも**1**単位の領域を占める 

通常, このコスト基準で論じられる

対数コスト基準

- 対数コスト基準 (logarithmic cost criterion)

- 数値の大きさ (ビット数) に比例したコストを割り当てる

$$l(i) = \begin{cases} \log_2 |i| + 1 & i \neq 1 \\ 1 & i = 0 \end{cases}$$

ビットは二進数

- オペランド **a** の種類に応じた対数コスト $t(\mathbf{a})$

オペランド **a**

コスト $t(\mathbf{a})$

=i

$l(i)$

i

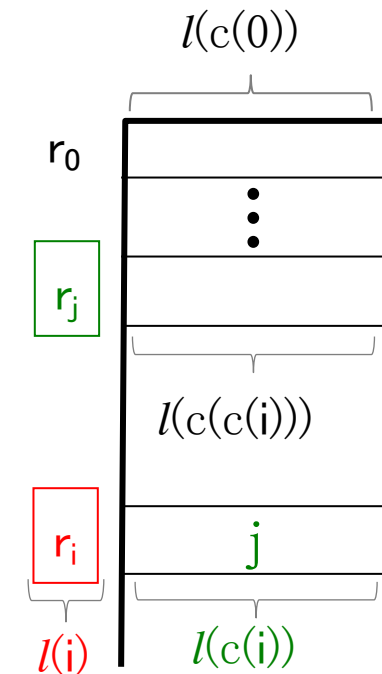
$l(i) + l(c(i))$

***i**

$l(i) + l(c(i)) + l(c(c(i)))$

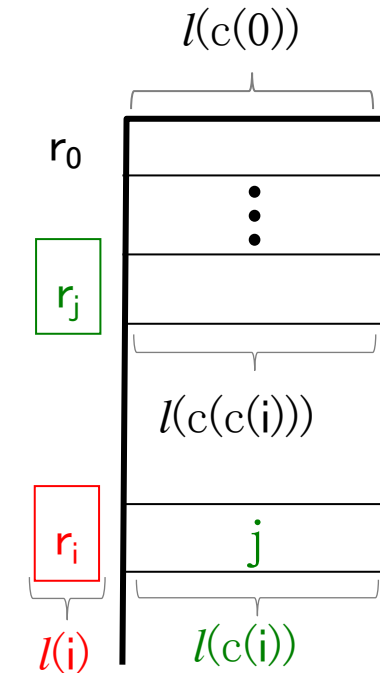


- 大きな数字を扱うことが本質的な問題の時に用いられる



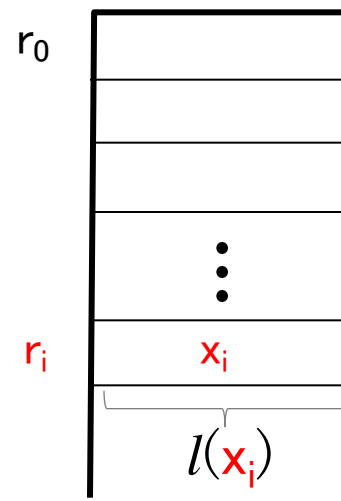
各命令の対数コスト基準時間計算量

命令		コスト
LOAD	a	$t(a)$
STORE	i	$l(c(0)) + l(i)$
STORE	$*i$	$l(c(0)) + l(i) + l(c(i))$
ADD	a	$l(c(0)) + t(a)$
SUB	a	$l(c(0)) + t(a)$
MULT	a	$l(c(0)) + t(a)$
READ	i	$l(\text{入力記号}) + l(i)$
WRITE	a	$t(a)$
JUMP		1
JGTZ		$l(c(0))$
HALT		1



対数コスト基準領域計算量

- 計算中にレジスタ r_i に貯えられる整数の絶対値の最大数を x_i とすると、 $\sum l(x_i)$



漸近的計算量：大きいオー0と大きいオメガ Ω

時間計算量と領域計算量：入力サイズ n の関数

入力サイズが大きくなったときのアルゴリズムの振舞？



計算量の上界(大きいオー)と下界(大きいオメガ)で論じる

大きいオーO（オーダー）

- 関数 $C(n)$ の上界: 大きいオーで論じる
- 関数 $C(n)$ に対し、ある正定数 k と n_0 が存在して n_0 以上の n に対して、常に $C(n) \leq k f(n)$ が成立するとき『大きいオーO』を使い、 $C(n) = O(f(n))$ と記す
…『 $C(n)$ は、 $f(n)$ のオーダーである』という。
- 大きいオーは、関数 $C(n)$ の「上界」を表す
- $C(n)$ の「上界」は無数に存在する
- 注意: $C(n) = O(f(n))$ の等号は厳密な意味での等号ではない！
- 注意: $= O(f(n))$ は必ず右辺に書く

$$C(n) \leq k f(n)$$

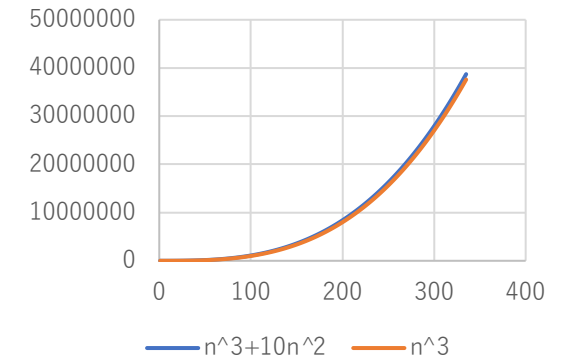
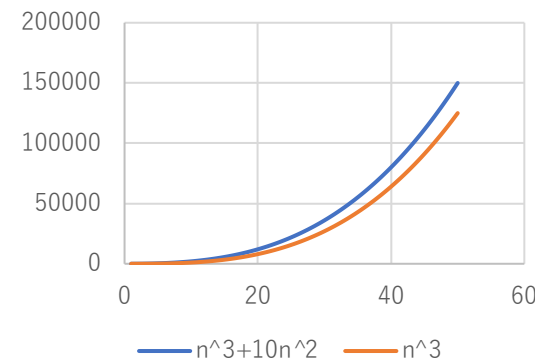
$$\frac{C(n)}{f(n)} \leq k$$

n の関数のオーダーの例

- $f(n) \leq k f(n)$ より、 $k f(n) = O(f(n))$
- $3000 = O(1) \cdots$ 定数時間 (定数オーダー)
 - $f(n)=1$, $k=3000$ とすればよい
- $100 n = O(n) \cdots$ 線形時間
 - $f(n)=n$, $k=100$ とすればよい (係数は無視できる)
- $\ln n = O(\log_2 n) \cdots$ 対数時間
 - $\ln n = (\ln 2) \log_2 n \leq \log_2 n$ 底変換しても変わらない (定数を省略して書くことも)
- $10n^2 + n^3 = O(n^3)$
 - $10n^2 + n^3 \leq 11n^3 \leq 11n^4 \leq 11n^5 \leq \cdots$ より
 - $10n^2 + n^3 = O(n^3) = O(n^4) = O(n^5) = \cdots$
 - 最も大きな次数に吸収される

$$C(n) \leq k f(n)$$

$$\frac{C(n)}{f(n)} \leq k$$



大きいオメガ Ω

- 関数 $C(n)$ の下界: 大きいオメガで論じる
- 関数 $C(n)$ に対し、ある正定数 k が存在して、無限個の n に対して、 $k f(n) \leq C(n)$ が成立するとき、
『大きいオメガ Ω 』を使い、 $C(n) = \Omega(f(n))$ と記す
- 大きいオメガは、関数 $C(n)$ の「下界」を表す
- $C(n)$ の「下界」は無数に存在する
- 注意: $C(n) = \Omega(f(n))$ の等号も厳密な意味での等号ではない!

大きいシータ Θ

- 関数 $C(n)$ の下界: 大きいオメガで論じる
- $C(n)$ の「上界」と「下界」は無数にあるため、 $C(n) = O(f(n))$ かつ $C(n) = \Omega(f(n))$ なる $f(n)$ が存在する.
これを $C(n) = \Theta(f(n))$ と記し、「上限」とする。
- 注意: $C(n) = \Theta(f(n))$ の **等号** も厳密な意味での等号ではない!
- それぞれの意味 (O , Ω , Θ)
 - $O(f(n))$ 上界: **速くても** $f(n)$ と同じくらいで発散する
 - $\Omega(f(n))$ 下界: **遅くても** $f(n)$ と同じくらいで発散する
 - $\Theta(f(n))$ 上限: **だいたい** $f(n)$ と同じスピードで発散する

〔例〕 $C(n) = 3n^2 + 2n$

- $C(n) = O(n^2), O(n^3), O(n^4), \dots$ が成立
- $C(n) = \Omega(n^2), \Omega(n), \dots$ も成立する
- $C(n) = \Theta(n^2)$
 n^2 は $3n^2 + 2n$ の「上限」になる！

$f(n)=n^n$ の計算量は？

RAMプログラム

	READ	1
	LOAD	1
	JGTZ	pos
	WRITE	=0
	JUMP	endif
pos:	LOAD	1
	STORE	2
	SUB	=1
	STORE	3
while:	LOAD	3
	JGTZ	continue
	JUMP	endwhile
continue:	LOAD	2
	MULT	1
	STORE	2
	LOAD	3
	SUB	=1
	STORE	3
	JUMP	while
endwhile:	WRITE	2
endif:	HALT	

対応するC言語風表現

```
scanf("%d", &r1);
```

```
if (r1<=0) printf("%d",0);
```

```
else {
    r2=r1;
```

```
    r3=r1-1;
```

```
    while (r3>0) {
```

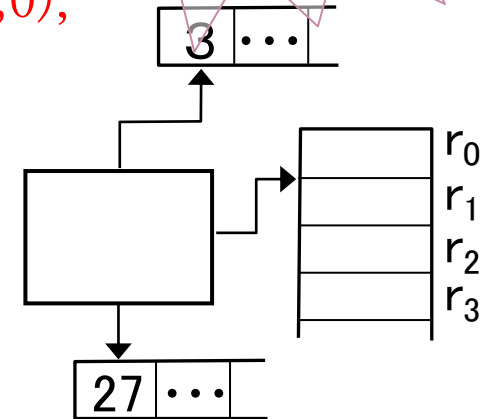
```
        r2=r2*r1;
```

```
        r3=r3-1; }
```

```
printf("%d", r2); }
```

RAMプログラム

$f(n)=n^n$



$r_1:3$ 入力値
 $r_2:3^1$ 求める値
 $r_3:2(3-1)$ ループ
 カウンタ



$f(n)=n^n$ の~~一~~様コスト基準時間計算

$f(n)=n^n$ の**一様**コスト基準**時間**計算 (ループ外)

	READ	1	<code>scanf("%d", &r1);</code>
	LOAD	1	
	JGTZ	pos	<code>if (r1<=0)</code>
	WRITE	=0	<code>printf("%d",0);</code>
	JUMP	endif	
pos:	LOAD	1	<code>else {</code>
	STORE	2	<code> r2=r1;</code>
	SUB	=1	
	STORE	3	<code> r3=r1-1;</code>
while:	LOAD	3	<code> while (r3>0) {</code>
		<code> </code>
	JUMP	while	<code> }</code>
endwhile:	WRITE	2	<code> printf("%d", r2); }</code>
endif:	HALT		

- ループ以外の最初と最後の命令は、**7+2**回実行される(入力が0の時は6回)
- 1命令の時間計算量は、1単位時間より、ループ以外の時間は**9単位**時間。

$f(n)=n^n$ の一樣コスト基準時間計算 (ループ内)

while:	LOAD	3	
	JGTZ	continue	
	JUMP	endwhile	while (r3>0) {
continue:	LOAD	2	
	MULT	1	
	STORE	2	r2=r2*r1;
	LOAD	3	
	SUB	=1	
	STORE	3	r3=r3-1; }
	JUMP	while	

- 1命令は、1単位時間より、1 ループ 9 単位時間
- ループは $n-1$ 回周るので、時間計算量 $9(n-1)+3$ 。
- このプログラムの一樣コスト基準での総時間計算量
 - $T(n) = 9(n-1) + 3 + 9 = 9n+3 = O(n)$ 。

$f(n)=n^n$ の対数コスト基準時間計算

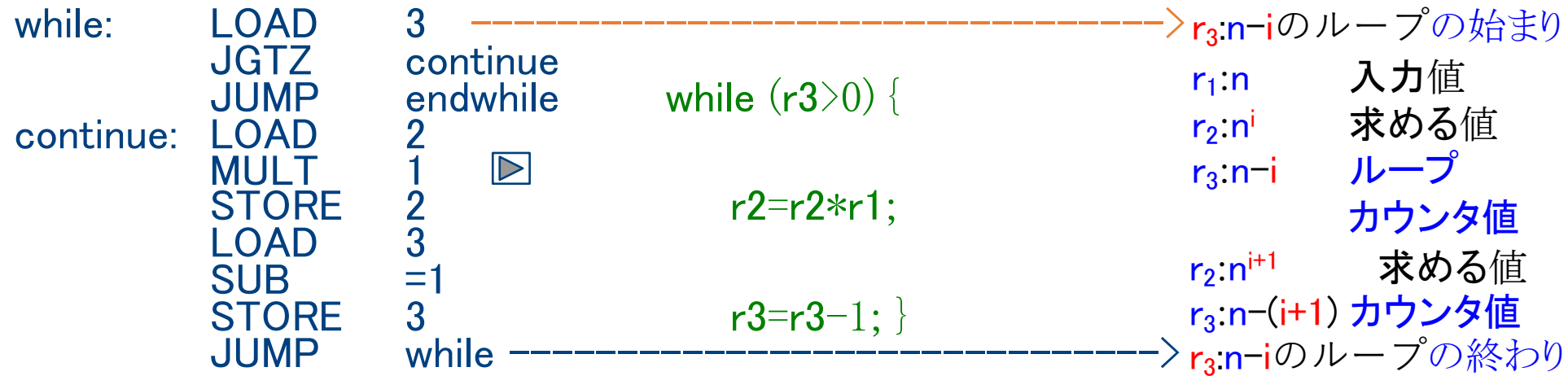
$f(n)=n^n$ の対数コスト基準時間計算量 (ループ外)

	READ	1	<code>scanf("%d",&r1);</code>	$l(\text{入力値}n) + l(1)$
	LOAD	1		
	JGTZ	pos		
	WRITE =0		<code>if (r1<=0) printf("%d",0);</code>	
	JUMP	endif		
pos:	LOAD	1	<code>else {</code>	$l(1) + l(n)$
	STORE	2	<code> r2=r1;</code>	$l(2) + l(n)$
	SUB	=1		$l(n) + l(1)$
	STORE	3	<code> r3=r1-1;</code>	$l(3) + l(n-1)$
while:	LOAD	3	<code> while (r3>0) {</code>	
		<code> </code>	
	JUMP	while	<code> } </code>	
endwhile:	WRITE 2		<code> printf("%d", r2); }</code>	$l(2)+l(n^n)$
endif:	HALT			

$$l(i) = \lfloor \log_2 |i| \rfloor + 1$$

- **SUB =1**の実行時間は、 $l(c(0))=l(n)$ と $l(1)$ との和より $\log_2 n + 2$ 。
- 他の命令も高々 $\log_2 n + \text{定数}$ である
- 最後の**WRITE 2**が $n \log_2 n + 3$
- ゆえに、 $O(n \log_2 n)$ 。

$f(n)=n^n$ の対数コスト基準時間計算量 (ループ内)



- ループ中のMULT 1の第*i*回ループ時の実行時間は $l(c(0))+t(1)$
 - $r_0 = r_2 = n^i, r_1 = n$ より $l(n^i) + l(1) + l(n) = (i+1) \log_2 n + 3 = O(i \log_2 n)$
- 次のSTORE 2は、 $O(i \log_2 n)$ 、LOAD 3 は $O(\log_2 n)$
- 第*i*回目のループの実行時間は、ある定数*k*で $k i \log_2 n$
- ループの実行時間 $= \sum_{i=1}^{n-1} k i \log_2 n = k \frac{n(n-1)}{2} \log_2 n = O(n^2 \log_2 n)$
- プログラムの対数コスト基準の総時間計算量 $T(n)=O(n^2 \log_2 n)$

$$l(i) = \lfloor \log_2 |i| \rfloor + 1$$

$f(n)=n^n$ の領域計算量

$f(n)=n^n$ の領域計算量

- 一様コスト基準の領域計算量は、使用するレジスタ数が 4

- $O(1)$

$$k = O(1)$$

- 対数コスト基準の領域計算量は、各レジスタに入る整数を考えると、

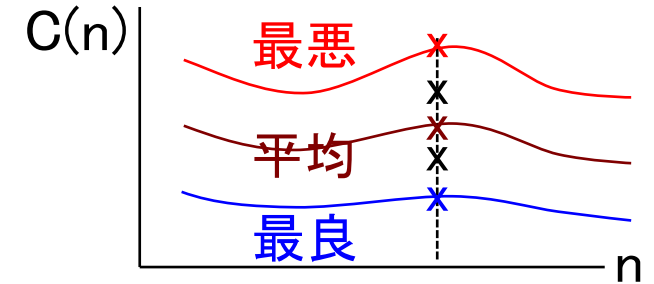
• 累算器	$r_0: r_2$ と同じ	$l(n^n) = n \log_2 n + 1$
• 入力値	$r_1: n$	$l(n) = \log_2 n + 1$
• 求める値	$r_2: n \sim n^n$	$l(n^n) = n \log_2 n + 1$
• ループカウンタ	$r_3: 0 \sim n-1$	$l(n-1) = \log_2(n-1) + 1$

$$l(i) = \log_2 |i| + 1$$

- $O(n \log_2 n)$

最も発散の速い項

最悪・平均・最良の計算量



- アルゴリズムの効率の評価
 - 最悪、平均、最良の場合の計算量
 - **平均**計算量(expected complexity)
 - 与えられた入力サイズの全ての入力に対する計算量の**平均値**
 - **最良**計算量(best-case complexity)
 - 与えられた入力サイズの全ての入力に対する計算量の**最小値**
 - **最悪**計算量(worst-case complexity)
 - 与えられた入力サイズの全ての入力に対する計算量の**最大値**
- アルゴリズムの効率は**平均計算量**で評価したいが、求めることが難しい
- ↓
- アルゴリズムの効率を**最悪計算量**で評価
⇒ **オーダー**で議論する

関数 $f(n)=n^n$ の計算量をまとめると

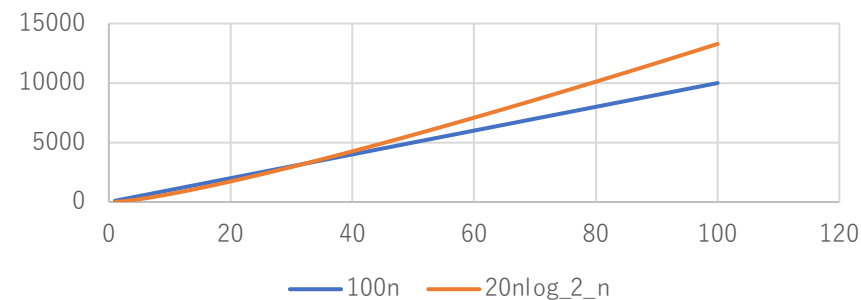
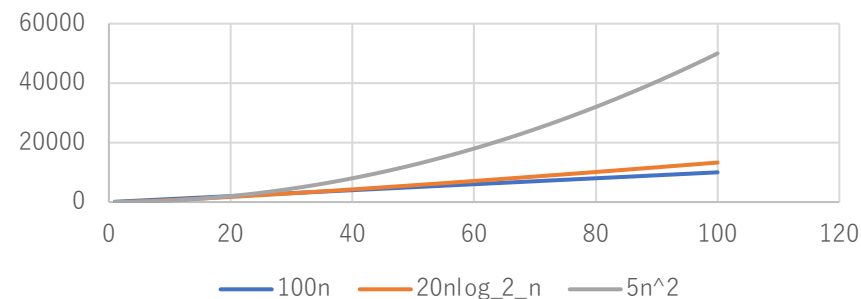
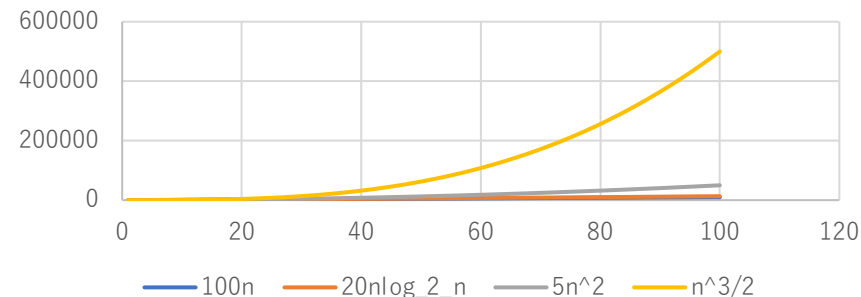
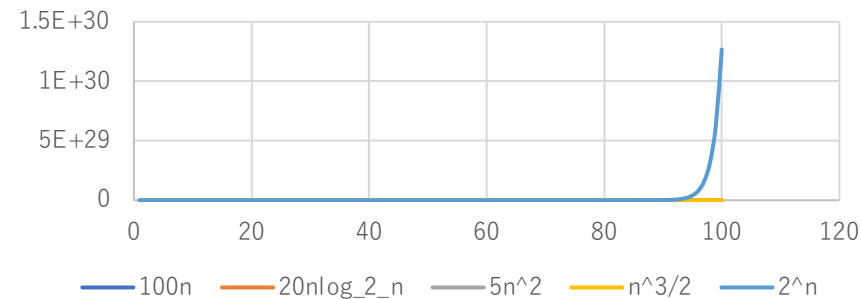
- このプログラム（アルゴリズム）では
 - 入力はずnなので
 - 最悪計算量＝平均計算量＝最良計算量 である
- 時間計算量
 - 一様コスト基準では $O(n)$
 - 対数コスト基準では $O(n^2 \log_2 n)$
- 領域計算量
 - 一様コスト基準では $O(1)$
 - 対数コスト基準では $O(n \log_2 n)$



オーダーの影響

- 計算機速度が10倍になったときや、計算時間を10倍かけたときの、各アルゴリズムの時間計算量の**入力サイズ**の**改善率**

		時間計算量 $T(n)$	10^3 で解ける問題 の大きさ n_1	10^4 で解ける問題 の大きさ n_2	改善率 n_2/n_1
アルゴリズム	1	$100n$	10	100	10.0
	2	$20n\log_2 n$	13	79	5.9
	3	$5n^2$	14	45	3.2
	4	$n^3/2$	13	27	2.3
	5	2^n	10	13	1.2



C言語プログラムの計算量

- C言語などの高級プログラミング言語で、**アルゴリズム**を記述して**計算量**を求める
 - **入力サイズに依存しない代入文・条件判定式**は、**1単位時間**で実行
 - **変数**は**1単位領域**を占めると考える
- ⇒ 『C言語プログラムの計算量』と『RAMプログラムの**一様コスト基準**の計算量』とは、**定数倍**しか変わらない。
- ⇒ 問題をC言語プログラムで記述し、その計算量を求める
- ⇒ **オーダー**で論じる限り「RAMプログラムの**一様コスト基準**の計算量」と「C言語プログラムの計算量」とは**等しい**

RAMプログラム

```
      READ      1
      LOAD      1
      JGTZ      pos
      WRITE=0
      JUMP      endif
pos:   LOAD      1
      STORE     2
      SUB       =1
      STORE     3
while: LOAD      3
      JGTZ      continue
      JUMP      endwhile
continue: LOAD    2
      MULT      1
      STORE     2
      LOAD      3
      SUB       =1
      STORE     3
      JUMP      while
endwhile: WRITE  2
endif:  HALT
```

対応するC言語風表現

```
scanf("%d", &r1);
```

```
if (r1<=0) printf("%d",0);
```

```
else {
```

```
    r2=r1;
```

```
    r3=r1-1;
```

```
    while (r3>0) {
```

```
        r2=r2*r1;
```

```
        r3=r3-1; }
```

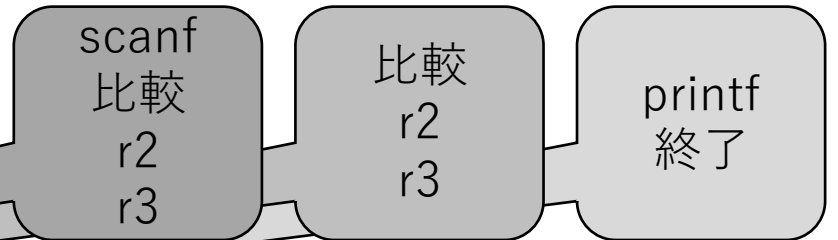
```
    printf("%d", r2); }
```

RAMプログラム
 $f(n)=n^n$

〔例〕 $f(n)=n^n$ のCプログラム

```
{
    scanf("%d", &r1);    /* r1=入力値n */
    if (r1<=0) printf("%d",0);
    else {
        r2=r1;
        r3=r1-1;
        while (r3>0) { /* r3=n-1から1で以下をループ */
            r2=r2*r1;
            r3=r3-1; }
        printf("%d", r2); }
}
```

- 時間計算量は $4 + 3(n-1) + 2 = 3n+3 : O(n)$ 。
- 領域計算量は 変数r1, r2, r3の3つ: $O(1)$ 。



時間計算量のオーダーでの議論

- 時間計算量を**オーダー**で議論するのは、
 - 入力サイズ**n**が**大きい**ときの**振舞**を調べる他に、
 - プログラムの**実行時間**は
 - 使用した**コンパイラ**（ソフトウェア）や **計算機**（ハードウェア）に**依存**する
- => 秒などの**具体的単位**では表わせず、『実行時間は **$n^2 \log_2 n$** に比例する』ということしか言えない
- => 実際に動かさなくても『実行時間は **$n^2 \log_2 n$** に比例する』ということがわかる

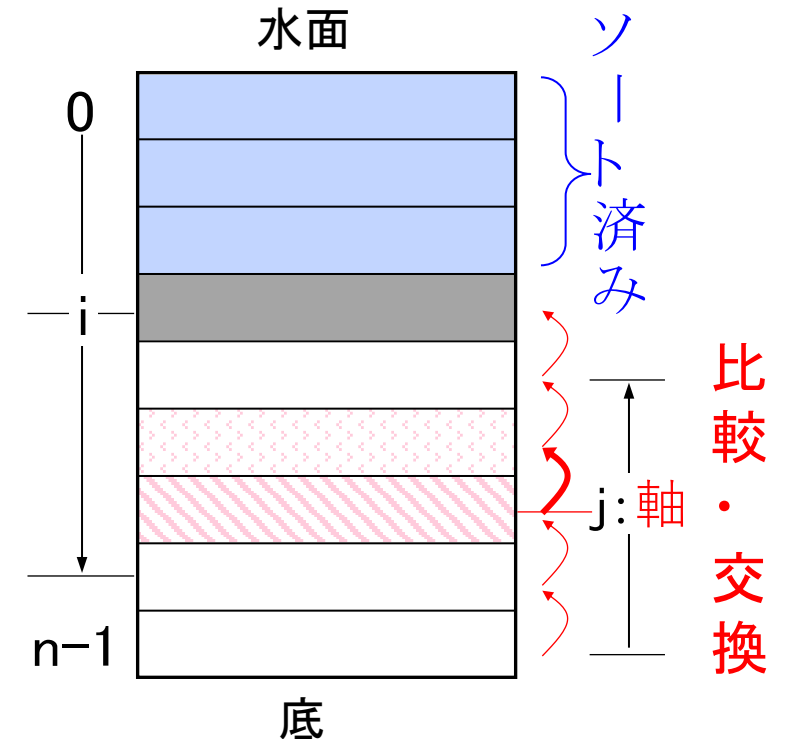
アルゴリズムと計算量の例

バブルソート

- 上下に数値データを入れる
- 上の要素と比較し大きな値を上にする
- これを下から順にやっていく
- 最後に最も大きい値が来る
- 一番上を除いてもう一度行う（繰り返す）
- 泡が上っていくように見える



Wikipediaより



バブルソートの第*i*フェーズ

バブルソート (C言語)

```
void BubbleSort(int A[]) {  
    int i,j,temp;  
    for(i=0; i<(n-1); i++)  
        for(j=n-1; j>i; j--)  
            if( A[j-1]>A[j] ){  
                temp=A[j-1];  
                A[j-1]=A[j];  
                A[j]=temp; }  
}
```

i=0～**n-2**の繰返し (iの**設定・比較は2回**)
j=n-1～i+1の**n-i-1**回繰返し (jの**設定・比較は2回**)
比較1回 必ず行われる
代入3回 最悪時必ず交換
最良時はこれらの代入は行われない

- **入力サイズ**:ソートされる配列要素数 **n**

最悪時間計算量 $\sum_{i=0}^{n-2} (2 + 6(n - i - 1)) = 3n^2 - n - 2$

最良時間計算量 $\sum_{i=0}^{n-2} (2 + 3(n - i - 1)) = \frac{3}{2}n^2 + \frac{1}{2}n - 2$

バブルソートは、最悪で n^2 のオーダーの**時間計算量**を要する。
領域計算量は、配列要素数の n と変数 $i, j, temp$ より $n+3$ 。

〔例〕 バブルソートの第0フェーズ

配列 44 55 06 42 94 18 12 67

第0フェーズ

0	i→44	44	44	44	44	44	44	<u>06</u>
1	55	55	55	55	55	55	06 ←j	44
2	06	06	06	06	06	06	06 ←j	55
3	42	42	42	42	12 ←j	12	12	12
4	94	94	94	12 ←j	42	42	42	42
5	18	18	12 ←j	94	94	94	94	94
6	12	12 ←j	18	18	18	18	18	18
7	比較	67 ←j	67	67	67	67	67	67

第1ステップ

第7ステップ 結果

〔例〕 バブルソートの過程

配列 44 55 06 42 94 18 12 67

0	i→44	→ <u>06</u>	06	06	06	06	06	06
1	55	i→44	→ <u>12</u>	12	12	12	12	12
2	06	55	i→44	→ <u>18</u>	18	18	44	44
3	42	12	55	i→44	→ <u>42</u>	42	55	55
4	94	42	18	55	i→44	<u>44</u>	44	44
5	18	94	42	42	55	i→55	<u>55</u>	55
6	12	18	94	→ <u>67</u>	67	67	i→67	<u>67</u>
7	67	67	67	94	94	94	94	94

第0フェーズ

第3フェーズ

第6フェーズ

結果

階乗計算 (C言語)

```
int Factorial(int n)
{
    if(n<=1) return( 1 );
    else return( n * Factorial(n-1) );
}
```

Factorial(n) の時間計算量 $T(n)$

- 入力サイズ: 引数の値: n
- 時間計算量:
 - $T(n) = \begin{cases} 4 + T(n-1) & n > 1 \\ 3 & \text{otherwise} \end{cases}$
漸化式を解くと
 - $T(n) = 4(n-1) + 3 = 4n - 1$
- 領域計算量
 - Factorial が呼び出される度に、仮引数 n (領域数1) がとられる
 - 再帰呼び出しは n 回 よって領域計算量は n になる。
- $O(n)$ の時間計算量と領域計算量をもつ

まとめ

- アルゴリズム
 - a^n の計算
- 計算量
 - ランダムアクセス機械
 - 漸近的計算量（おおよその計算量）
 - n^n の計算量
 - バブルソート，階乗計算の計算量

アルゴリズムの速度を体感してみる

- 実際にプログラムを書いて、 n を変えて実行速度を測る
- 本当に実行時間の関数は、オーダーに比例する関数になるかな？
- **プログラムの実行時間の測定**
- Unixのtimeコマンドを使ってプログラムの実行時間をはかることができる

```
$ time ./a.out
real 0m0.011s
user 0m0.004s
sys 0m0.002s
```

 - real: プログラムの呼び出しから終了までにかかった時間
 - user: プログラム自体の処理時間（ユーザCPU時間）
 - sys: プログラムの処理するために、OSが処理をした時間（システム時間）
- プログラムの処理時間はuserに反映される