アルゴリズム　計算量　　森下

教科書　Robert Sedgewick: “Algorithms in C”　Addison Wesley
（邦訳　近代科学社　１９９６年）

基礎
１　　はじめに
２　　C
３　　基本データ構造
４　　木
５　　再帰呼出し
６　　アルゴリズムの解析
７　　アルゴリズムの実現

整列
８　　初等的な整列法
９　　クイックソート
１０　基数整列法
１１　順序キュー
１２　マージソート

探索
１６　ハッシュ表

文字列処理
１９　文字列探索

C言語ではなく Java を使う

Homepage 講義資料と講義で使う Java program
http://mlab.cb.k.u-tokyo.ac.jp/~moris/lecture/upbsb/2005-system/

生物情報科学実験２の課題との関係

- shotgun sequencing：生物情報科学実験I で解読された配列についてソフトウエアを利用してつなげ評価したのちに、大規模塩基配列決定の仕組みを理解するためのプログラム製作を行う。
- タンパク質発現プロファイル：プロテインチップで収集した断片からデータベースを利用して本来のタンパク質を同定する作業を行った後に、データベースを検索するプログラムを作成する。
- DNA チップデータ解析：DNA チップを用いて観測された遺伝子発現量データを、機械学習手法（クラスタリング、クラス分類など）を用いて解析する。さらに初歩的な機械学習のプログラムを作成する。

最大公約数の計算　ユークリッドの互助法

u>v ならば （u と v の最大公約数）＝（v と u-v の最大公約数）

```
public class Gcd {
        public static int gcd(int u,int v){
                int t;
                while(u > 0){
                        if(u < v){
                                t = u;
                                u = v;
                                v = t;
                        }
                        u = u - v;
                }
                return v;
        }
        public static void main(String[] args) {
                System.out.println(gcd(36, 24));
                System.out.println(gcd(36, 45));
        }
}
```

３章　基本データ構造

配列　エラトステネスのふるい

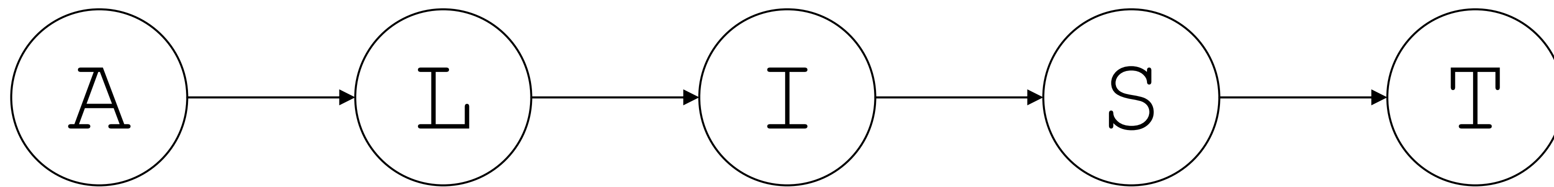
a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0

```
public class Eratosthenes {
        private static final int N=50;     // N=50 までの素数を出力

        public static void main(String[] args) {
                int[] a = new int [N+1];  // N+1の長さの配列を定義
                for(int i=1; i<=N; i++){  // i++ は i=i+1;
                        a[i]=1;                   // 1 は素数であることを意味
                }                                 // 最初は全て素数として初期化
                for(int i=2; i<=N/2; i++){
                        for(int j=2; j<=N/i; j++){
                                a[i*j]=0;         // 合成数であることがわかると 0
                        }
                }
                for(int i=1; i<=N; i++){
                        if(a[i]>0){
                                System.out.print(i+" ");
                        }
                }
        }
}
```

配列のインデックスの範囲に注意


```
public class Eratosthenes {
        private static final int N=50;        // N=50 までの素数を出力

        public static void main(String[] args) {
                int[] a = new int [N+1];      // N+1の長さの配列を定義
                for(int i=1; i<=N; i++){
                        a[i]=1;               // 1 は素数であることを意味
                }                             // 最初は全て素数として初期化
                for(int i=2; i<=N/2; i++){
                        for(int j=2; j<=N; j++){      // バグ
                                a[i*j]=0;     // N+1 < N/2 * N
                        }
                }
                for(int i=1; i<=N; i++){
                        if(a[i]>0){
                                System.out.print(i+" ");
                        }
                }
        }
}
```

リンクを使ったリスト表現

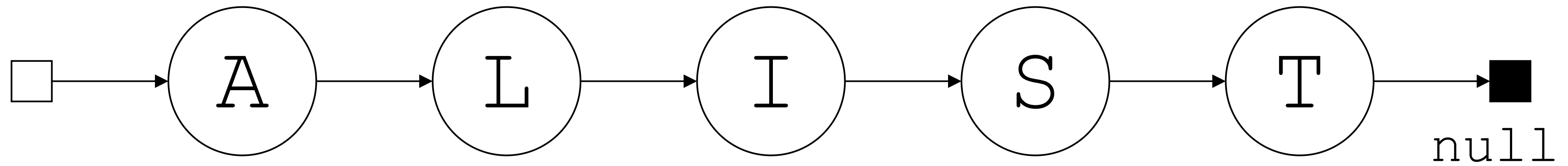


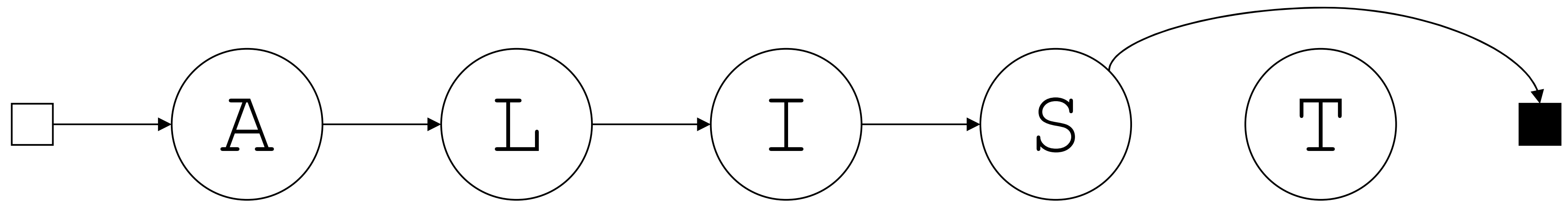
利点　実行中にデータ量に応じて伸縮できる
前もってデータ量を知らなくてよい

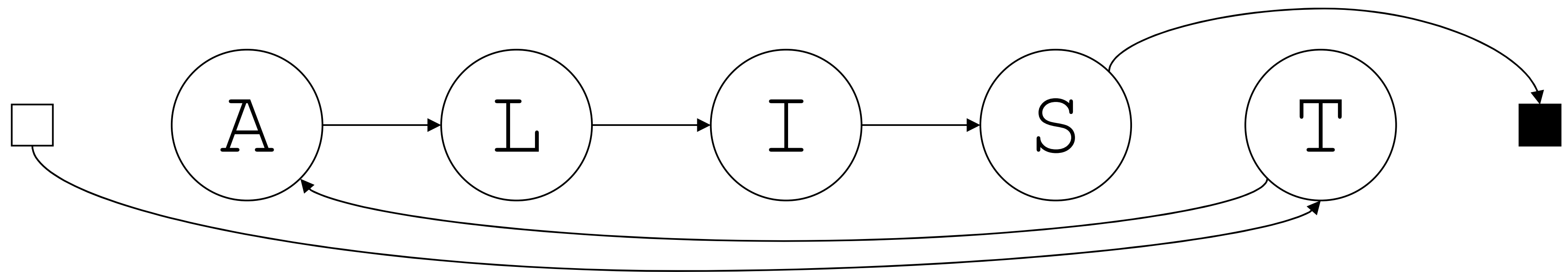
項目の並べ替えが楽

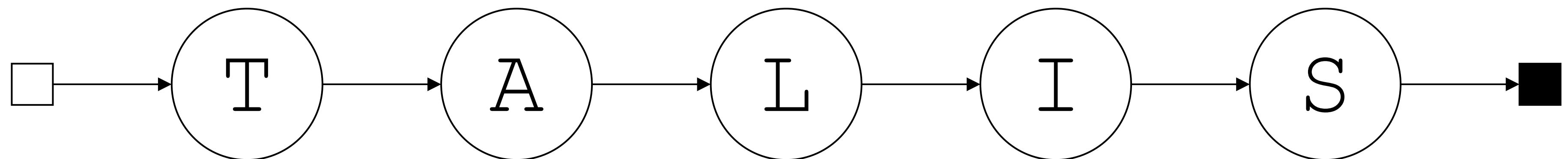
欠点　k 番目の項目を見つける操作
直前の項目を見つける操作

リンクを使ったリスト表現









リスト処理の実装


```
public class Node {
        public int key;
        public Node next;

        public Node(int n){
                key = n;
                next = null;
        }
        public void deleteNext(Node t){
                if(t.next != null){
                        t.next = t.next.next;
                }
        }
        public static void insertAfter(int v, Node t){
                Node x = new Node(v);
                x.next = t.next;      // step 1
                t.next = x;           // step 2
        }
        :
}
```

key → next

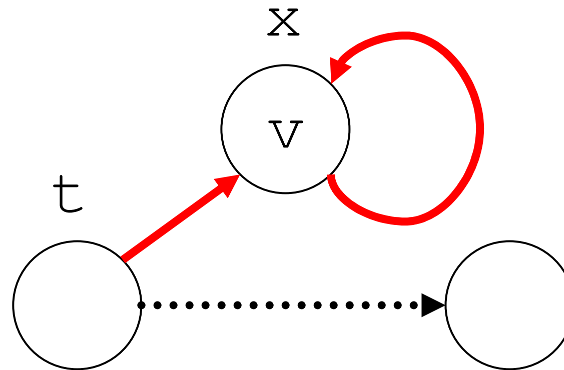
t　t.next　t.next.next

x　v　t　step 1　step 2

処理前　処理後

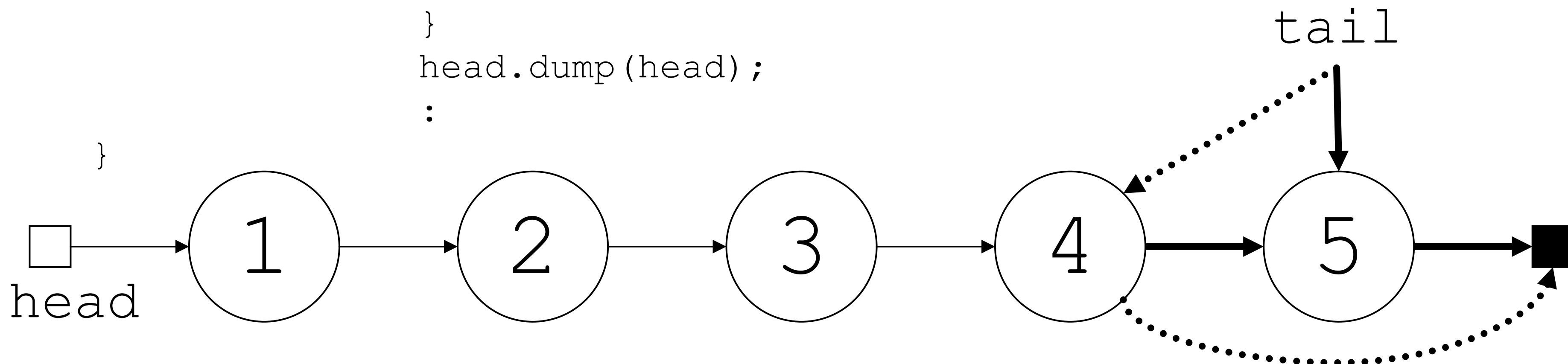
ポインターのつけかえ順序に注意


```
public Node insertAfter(int v, Node t){
        Node x = new Node(v);
        t.next = x;
        x.next = t.next;
        return x;
}
```



リスト処理の実装


```
          :
          public void dump(Node t){
                  System.out.println("Dumping");
                  for(Node x = t; x != null; x = x.next){
                          System.out.print(x.key+" ");
                  }
                  System.out.println();
          }

          public static void main(String[] args) {
                  Node head = new Node(0);
                  Node tail = head;
                  for(int i = 1; i < 20; i++){
                          insertAfter(i, tail);
                          tail = tail.next;
                  }
                  head.dump(head);
                  :
}
```

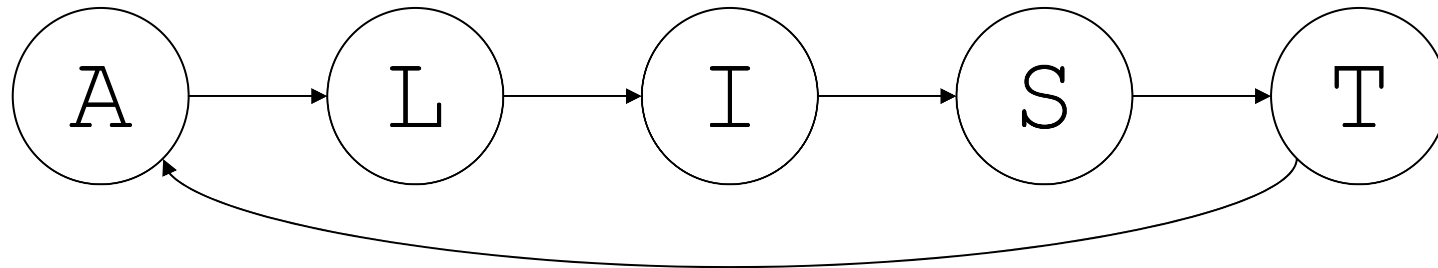


リストの拡張

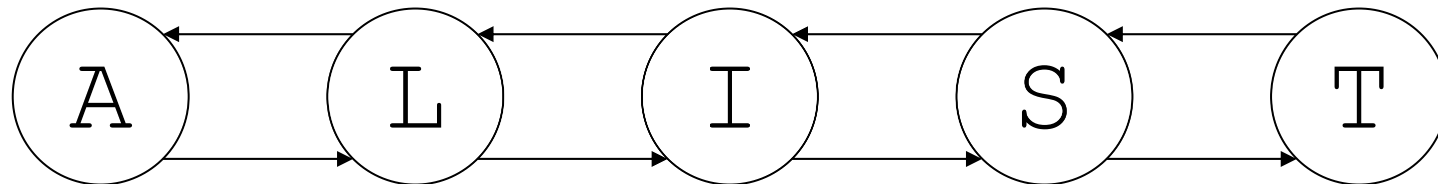
最初と最後に特殊なノードは必要か？

直前の項目を見つける操作が困難

循環リスト



両方向リスト

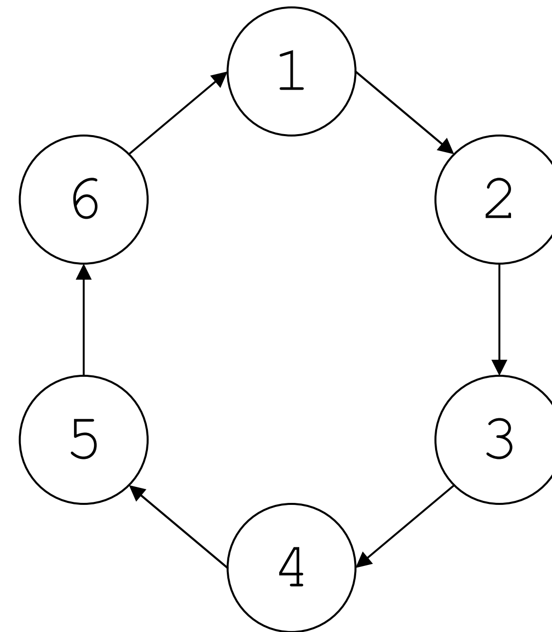


ジョセファスの問題

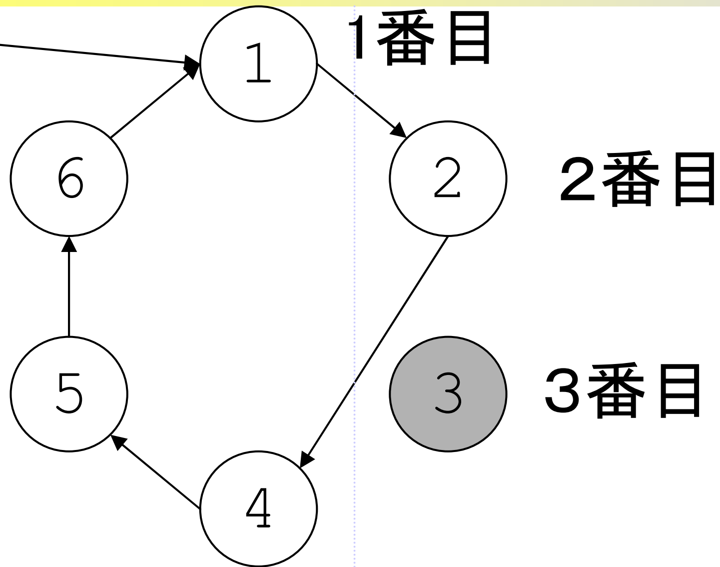
N個のノードを円周上に配置

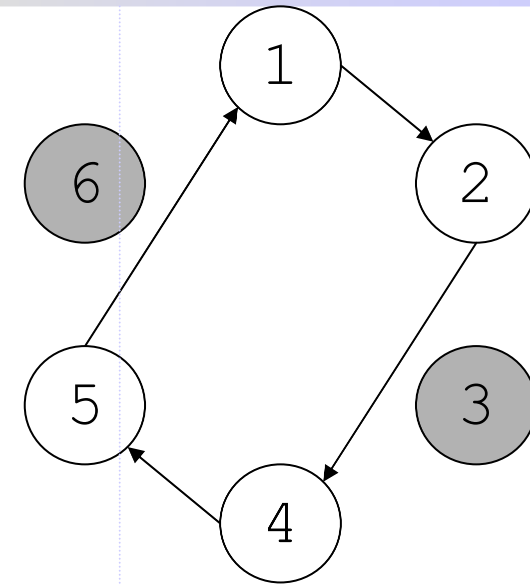
M番目ごとにノードを除去

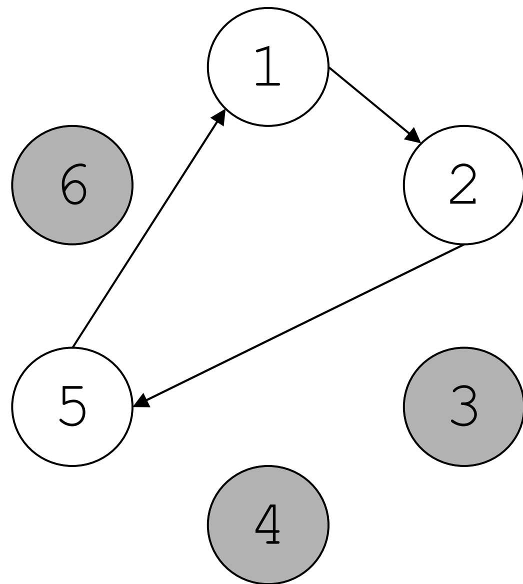
最後に残るノードは何か？

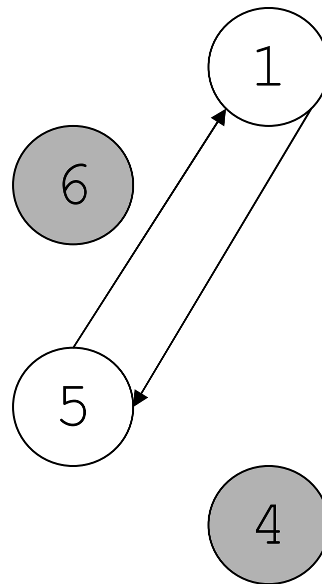


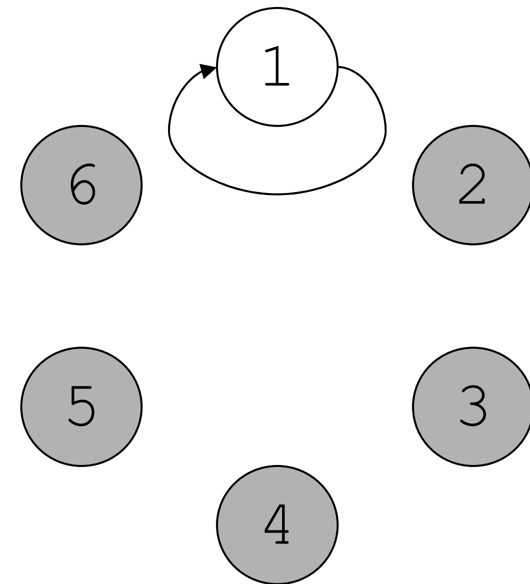
ジョセファスの問題



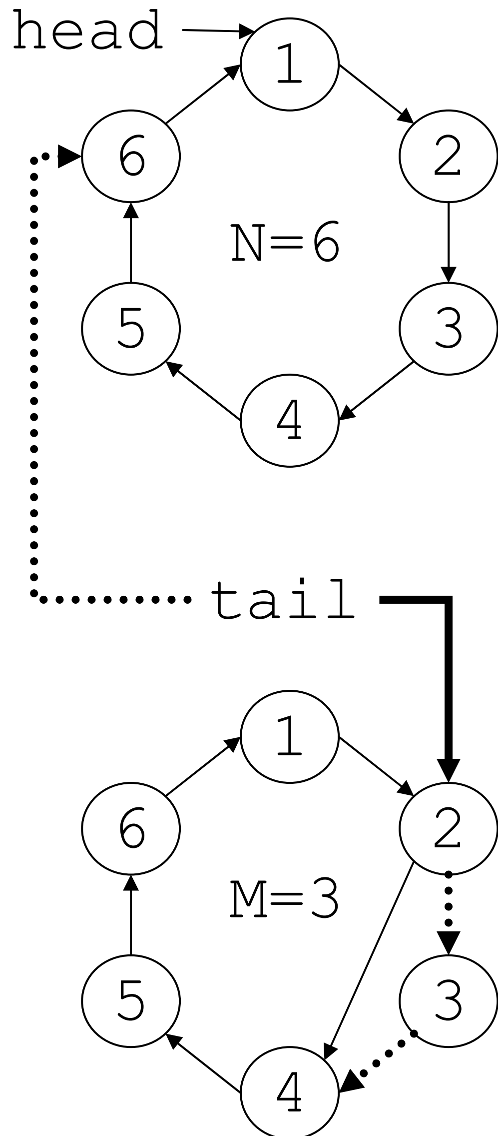








循環リストとジョセファスの問題

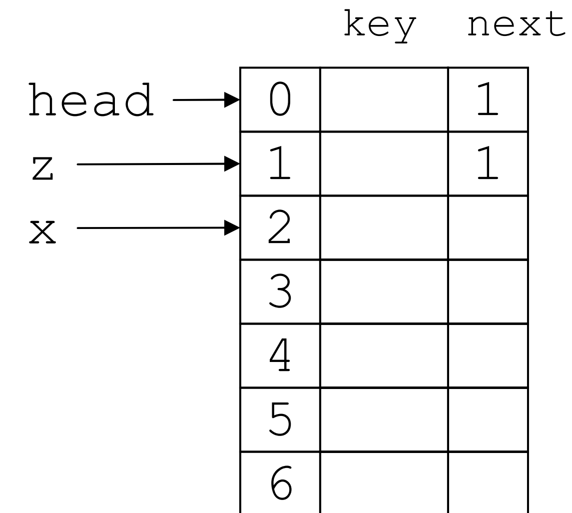


tail の次を削除

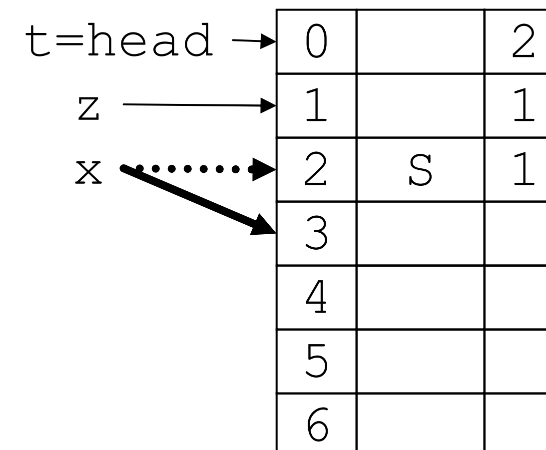
```
int n = 6;
int m = 3;

head = new Node(1);
tail = head;
for(int i = 2; i <= n; i++){
        insertAfter(i, tail);
        tail = tail.next;
}
tail.next = head;

while(tail != tail.next){
        for(int i = 1; i < m; i++){
                tail = tail.next;
        }
        System.out.println("Delete "+tail.next.key);
        tail.next = tail.next.next;
}
System.out.println("Answer = " + tail.key);
```

配列を使ったリストの実装


```
public class Node {
        private char[] key;
        private int[] next;
        private int head, z, x; // z は null に相当

        public Node(int size) {
                key = new char[size + 2];
                next = new int[size + 2];
                head = 0;
                z = 1;
                x = 2;
                next[head] = z;
                next[z] = z;
        }

        public void deleteNext(int t) {
                next[t] = next[next[t]];
        }

        public int insertAfter(char v, int t) {
                key[x] = v;
                next[x] = next[t];
                next[t] = x;
                return x++;
        }
        :
```



insertAfter('S', head)



配列を使ったリストの実装


```
public static void main(String[] args) {
        Node l = new Node(5);
        int t1 = l.insertAfter('S', l.head);
        int t2 = l.insertAfter('L', l.head);
        l.insertAfter('A', l.head);
        l.insertAfter('I', t2);
        l.insertAfter('T', t1);
        l.printList();
}
```

head	0		2
	1		1
t1	2	S	1
	3		
	4		
	5		
	6		

		3
		1
	S	1
t2	L	2

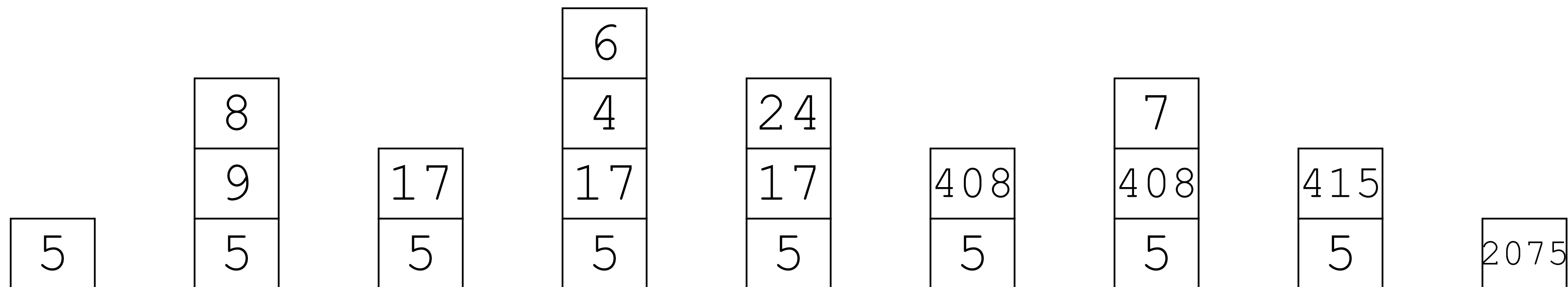
	4
	1
S	1
L	2
A	3

		4
		1
	S	1
t2	L	5
	A	3
	I	2

		4
		1
t1	S	6
	L	5
	A	3
	I	2
	T	1

スタック

5*(((9+8)*(4*6))+7)




```
push(5);
push(9);
push(8);
push(pop()+pop());
```

```
push(4);
push(6);
push(pop()*pop());
push(pop()*pop());
```

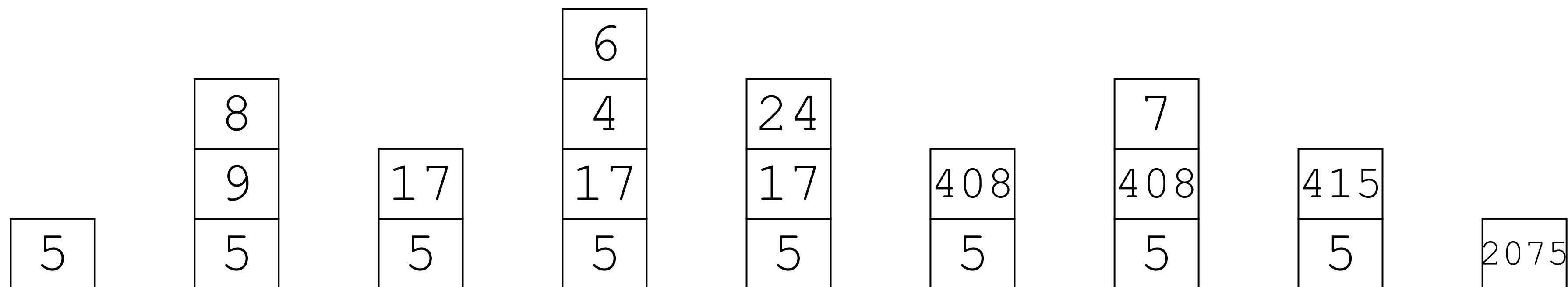
```
push(7);
push(pop()+pop());
push(pop()*pop());
```

計算の中間結果を貯える

逆ポーランド記法　演算子を後置する記法

5*(((9+8)*(4*6))+7)　中置記法

598+46**7+*　逆ポーランド記法

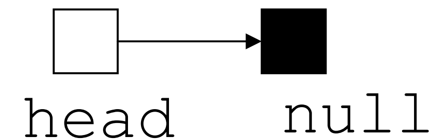


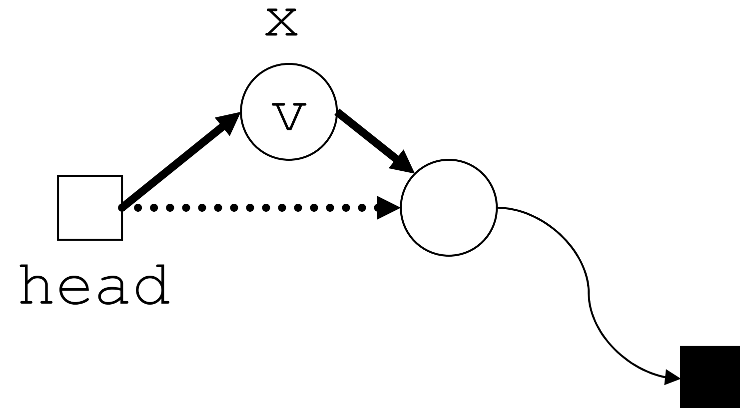

```
push(5);
push(9);
push(8);
push(pop()+pop());
push(4);
push(6);
push(pop()*pop());
push(pop()*pop());
push(7);
push(pop()+pop());
push(pop()*pop());
```

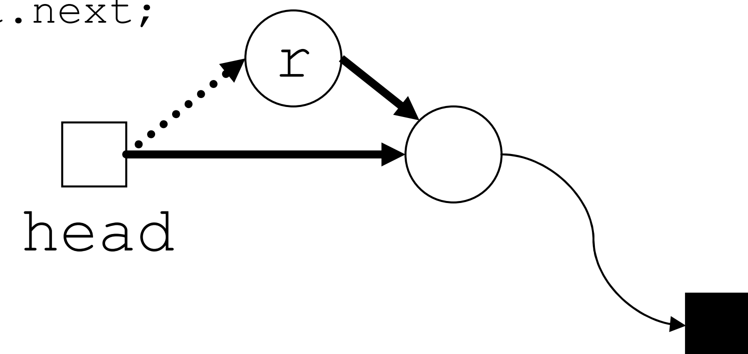
逆ポーランド記法はスタックでの計算を意識すれば書き下しやすい

リンクをつかったスタックの実装


```
public class Stack {
        private Node head;
        public Stack(){
                head = new Node(-1);
                head.next = null;
        }
        public void push(int v){
                Node x = new Node(v);
                x.next = head.next;
                head.next = x;
        }
        public int pop(){
                int r = -1;
                if( !stackEmpty() ){
                        r = head.next.key;
                        head.next = head.next.next;
                }
                return r;
        }
        public boolean stackEmpty(){
                return (head.next == null);
        }
        :
```







配列を使ったスタックの実装

```
public class Stack {
        private int MAX;
        private int[] stack;
        private int p;
        public Stack(int size) {
                MAX = size;
                stack = new int[MAX + 1];
                p = 0;
        }
        public void push(int v) {
                if (p < MAX) {
                        stack[p++] = v;  // stack[p]=v; p++;
                }
        }
        public int pop() {
                if ( !stackEmpty() ) {
                        return stack[--p]; // p=p-1; return stack[p];
                }else {
                        return -1;
                }
        }
        public boolean stackEmpty(){
                return (p == 0);
        }
        :
```

スタック (LIFO)
Last-In-First-Out
最後に入ったものを最初に処理

キュー
(FIFO) First-In-First-Out
最初に入ったものを最初に処理

キュー

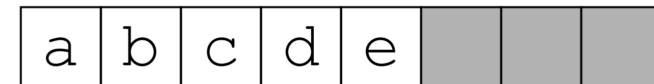

```
public class Queue {
        private int MAX;
        private char[] queue;
        private int head, tail;

        public Queue(int size){
                MAX = size;
                queue = new char[MAX+1];
                head = 0;
                tail = 0;
        }
        public void put(char v){
                queue[tail++] = v;
                if(tail > MAX){
                        tail = 0;
                }
        }
        public char get(){
                char t = queue[head++];
                if(head > MAX){
                        head = 0;
                }
                return t;
        }
```

MAX = 7

0 1 2 3 4 5 6 7

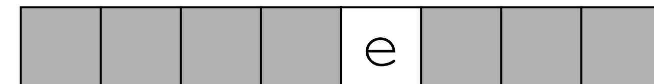
put(a); ..., put(e);

head=0
tail=5



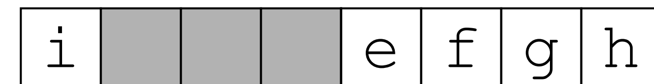
get();get();get();get();

head=4
tail=5



put(f); ..., put(i);

head=4
tail=1

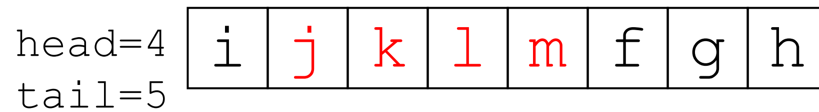


バグはどこ？

キュー

キューがあふれる場合

put(j);put(k);put(l);put(m);



空の場合

get();get();get();
get();get();get();

put(j); get();

入れたはずのjを取り出せない...

キュー

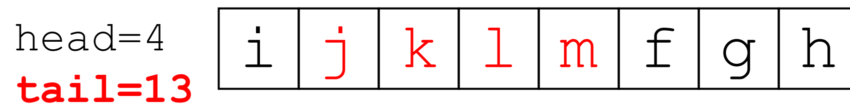
max を超えたとき 0 に戻さない方針

head=4
tail=9

キューがあふれる場合

put(j);put(k);put(l);put(m);

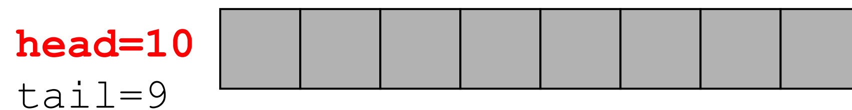
head=4
tail=13



空の場合

get();get();get();
get();get();get();

head=10
tail=9

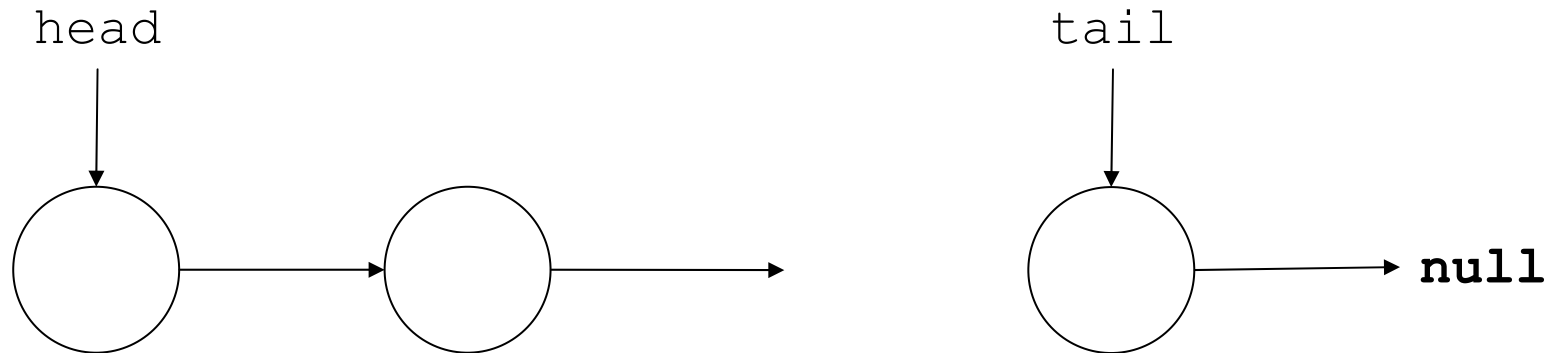


MAX = 7

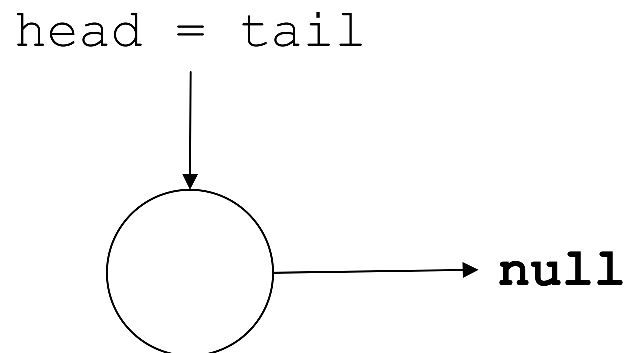
修正案 配列のインデックスを mod で計算

```
public void put(char v){
    if( tail < head + MAX ){
        // MAX 個より多く入れない
        queue[tail%(MAX+1)] = v;
        tail++;
    }
}

public char get(){
    char t = -1;
    if( head < tail ){
        t = queue[head%(MAX+1)];
        head++;
    }
    return t;
}
```

リンクを使ってキューを実装 するには？



空の場合 `head = tail = null`



リンクを使ってキューを実装 するには？


```
public class Queue {
        private Node head, tail;        // Node は以前の定義を借用
        public Queue(){ head = null; tail = null; }
        public void put(char c){
                Node x = new Node(c);
                if(queueEmpty()){ tail = x; head = x;}
                                // キューが空のとき head, tail を新ノードに
                else{ tail.next = x; tail = x; }  // 一番最後に x を付加
        }
        public char get(){
                char r = ‘ ’;           // キューが空の時には空白をかえすように
                if(!queueEmpty()){
                        r = (char) head.key;   // 空でないなら先頭の値を r へ
                        if(head == tail){ head = null; tail = null; }
                                                // 1つしか元がない場合
                        else{ head = head.next; } // 元が複数存在する場合
                }
                return r;
        }
        public boolean queueEmpty(){
                return (head == null);
        }
}
```

抽象データ型 (abstract data type)

スタック、キューはリストや配列で実装できる

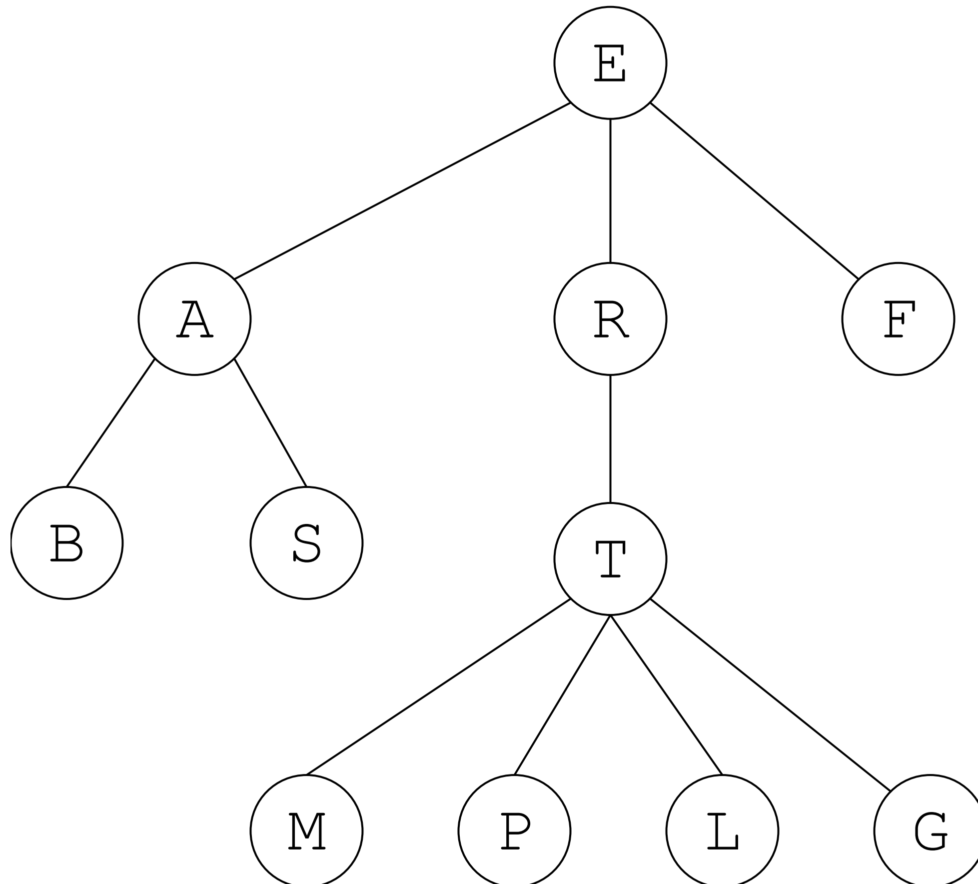
外部からみると、キューにおいては、
実装方式の詳細を知らなくても
`put, get` を使うことができれば便利

`put, get` のような操作だけを公開して、
内部の実装を参照させない

実装を変更しても外部への影響がない
大規模なソフトウエア開発では、むしろ好都合

4章　木構造

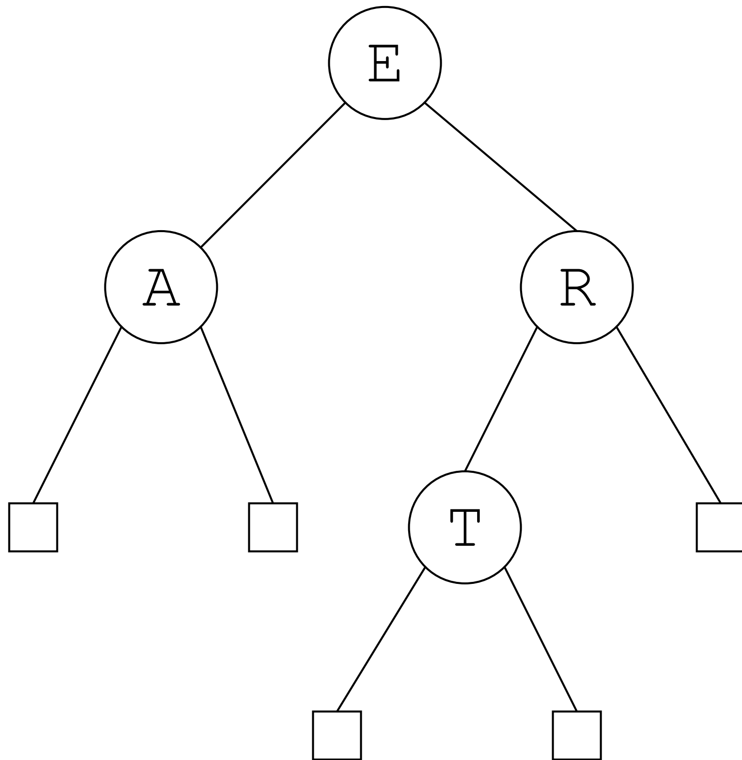
木

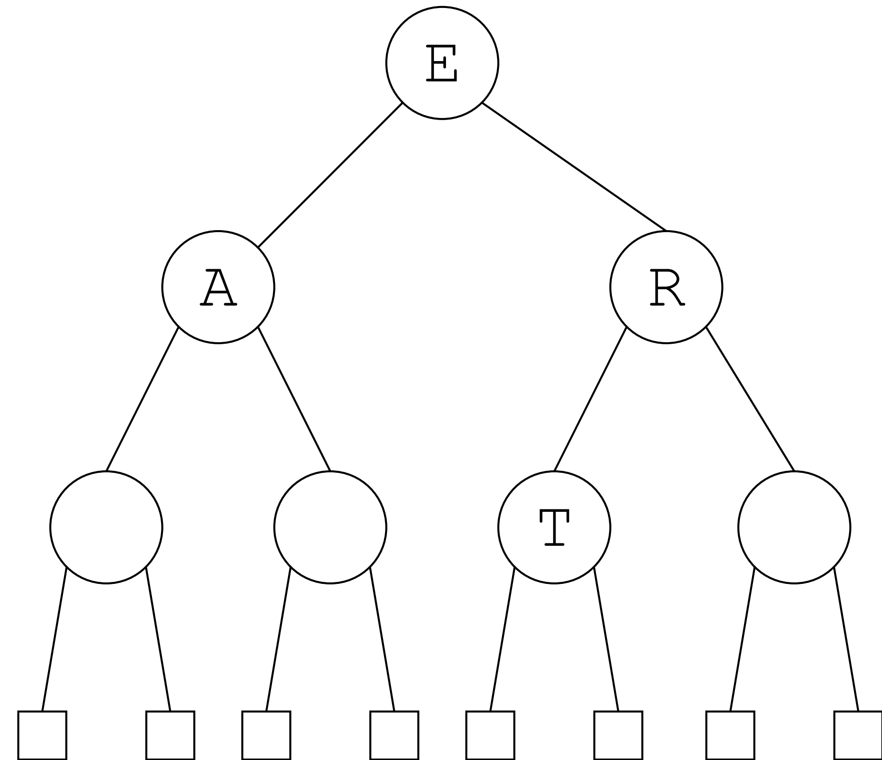


節点（ノード）／頂点
根ノード E
道（パス）
E→R→T→P
親 M に対する T
子 T に対する M,P,L,G
葉／終端点／外部節点
B,S,M,P,L,G,F
内部節点（葉でない）
E,A,R,T
部分木 T の部分木
T,M,P,L,G
レベル 根からの節点数
R=1, T=2, M=3
高さ 最大のレベル

2分木

2分木　　内部節点は2つの子をもつ
完全2分木　　一番下のレベル以外のレベルは内部節点で詰まっている

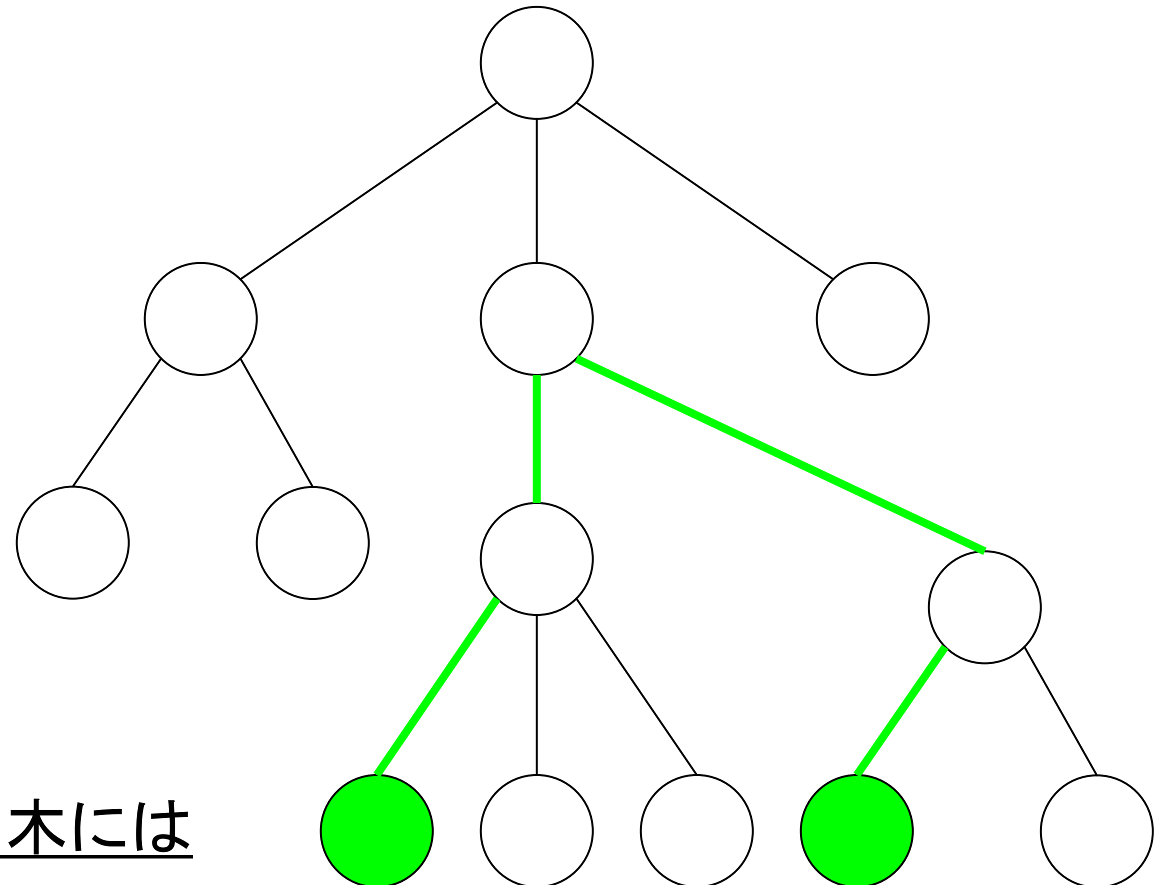




木の性質

<u>任意の2つの節点を結ぶ道はただ1つである</u>

最小共通祖先:
2つの頂点と根を結ぶ
2つの道に共通して
現れる節点で
最も葉に近い節点



最小共通節点を
経由して唯一の道が
みつかる

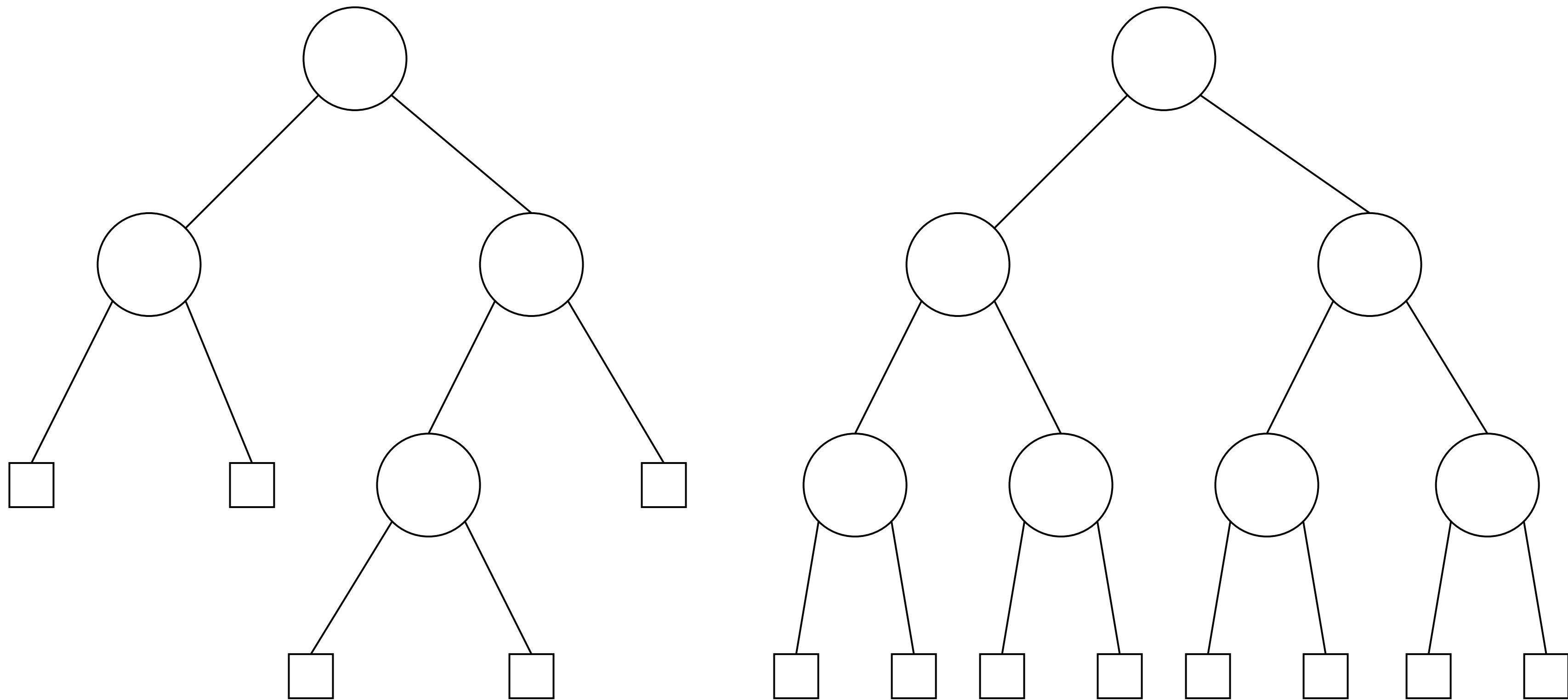
<u>節点の個数が N である木には
辺が N-1 個</u>

各節点には親への辺がただ1つ　　根には親がない

木の性質

内部節点数が N の2分木には、N+1 個の外部節点

内部節点数が N の完全2分木の高さは、約 $\log_2 N$

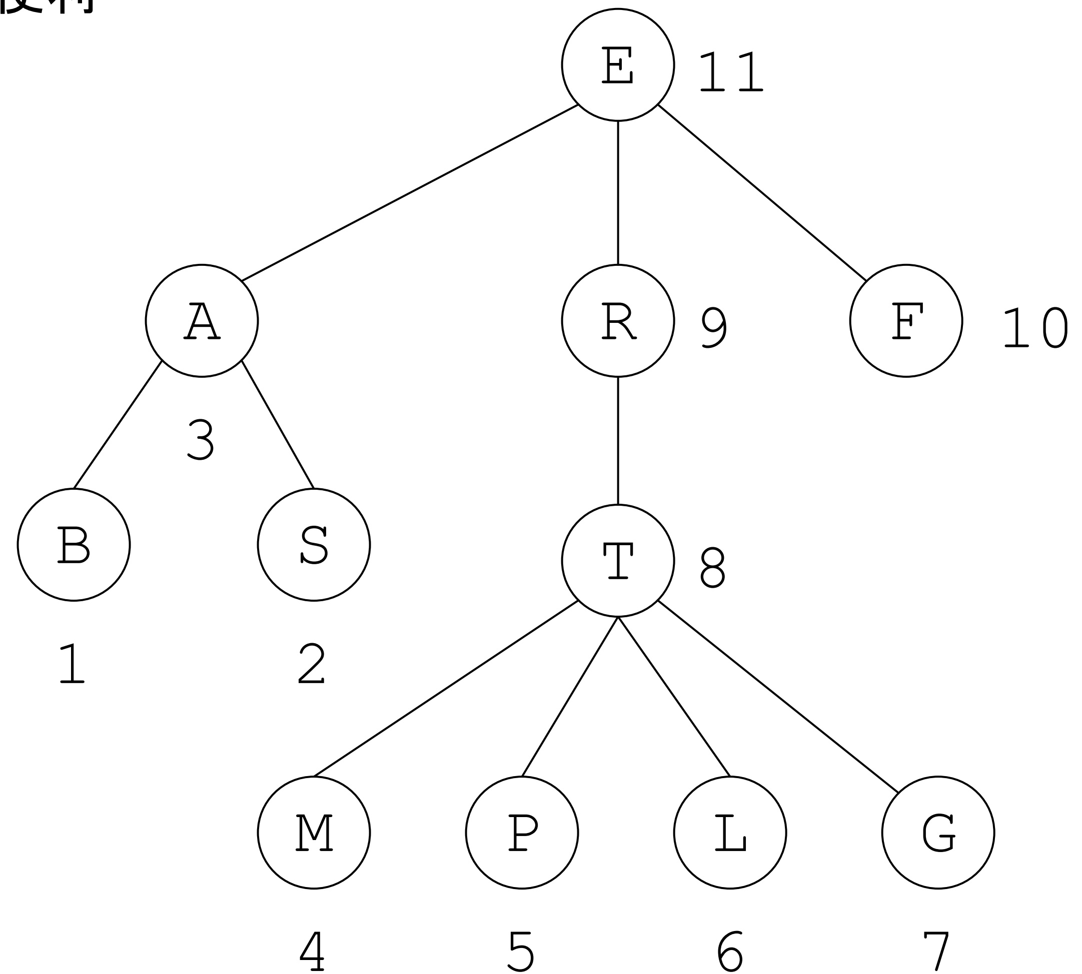


木の表現

親だけ覚えておく方式

下から上への移動に便利

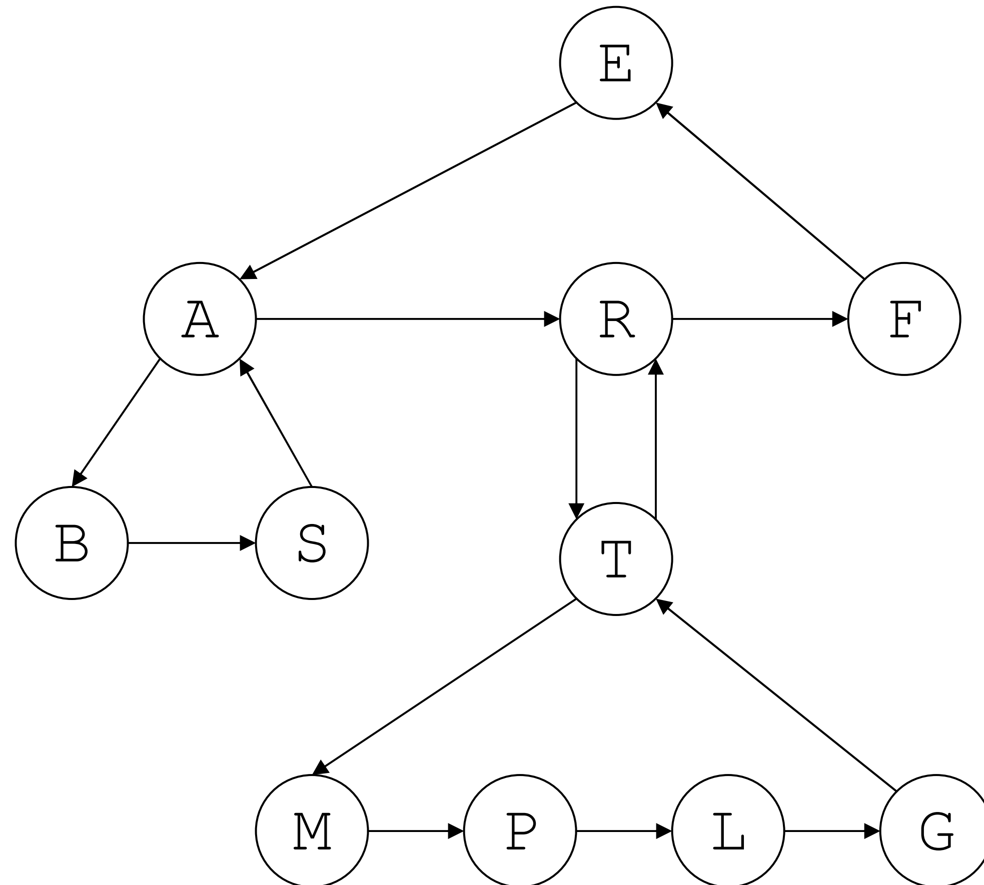
k	a[k]	dad[k]
1	B	3
2	S	3
3	A	11
4	M	8
5	P	8
6	L	8
7	G	8
8	T	9
9	R	11
10	E	11
11	F	11



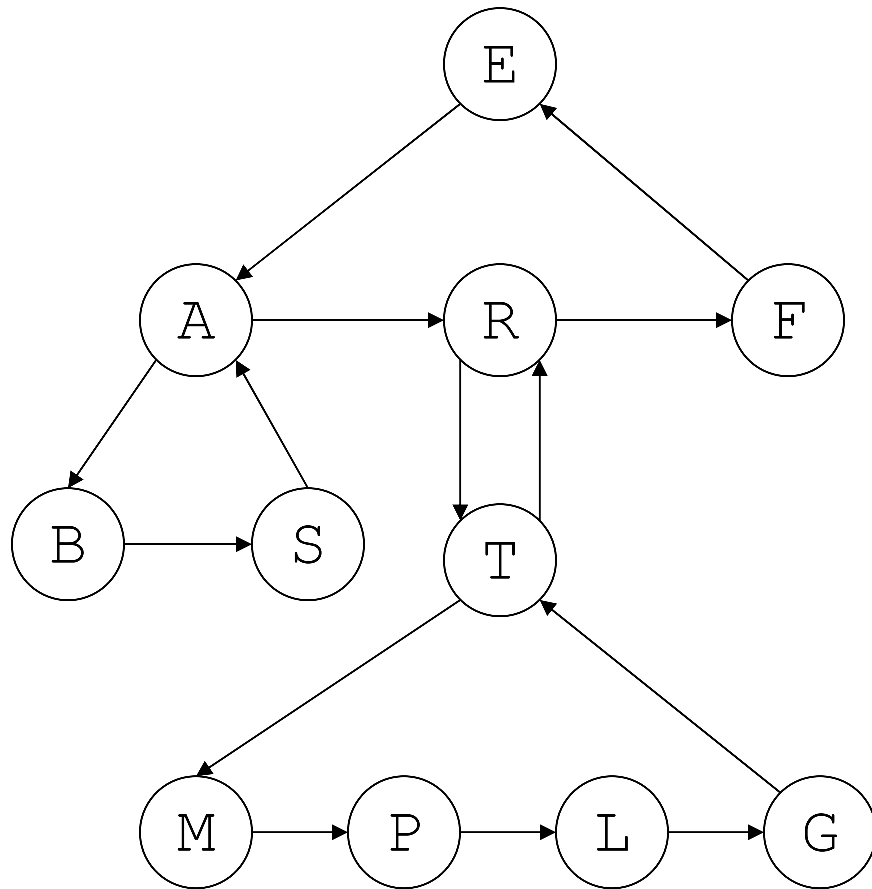
木の表現

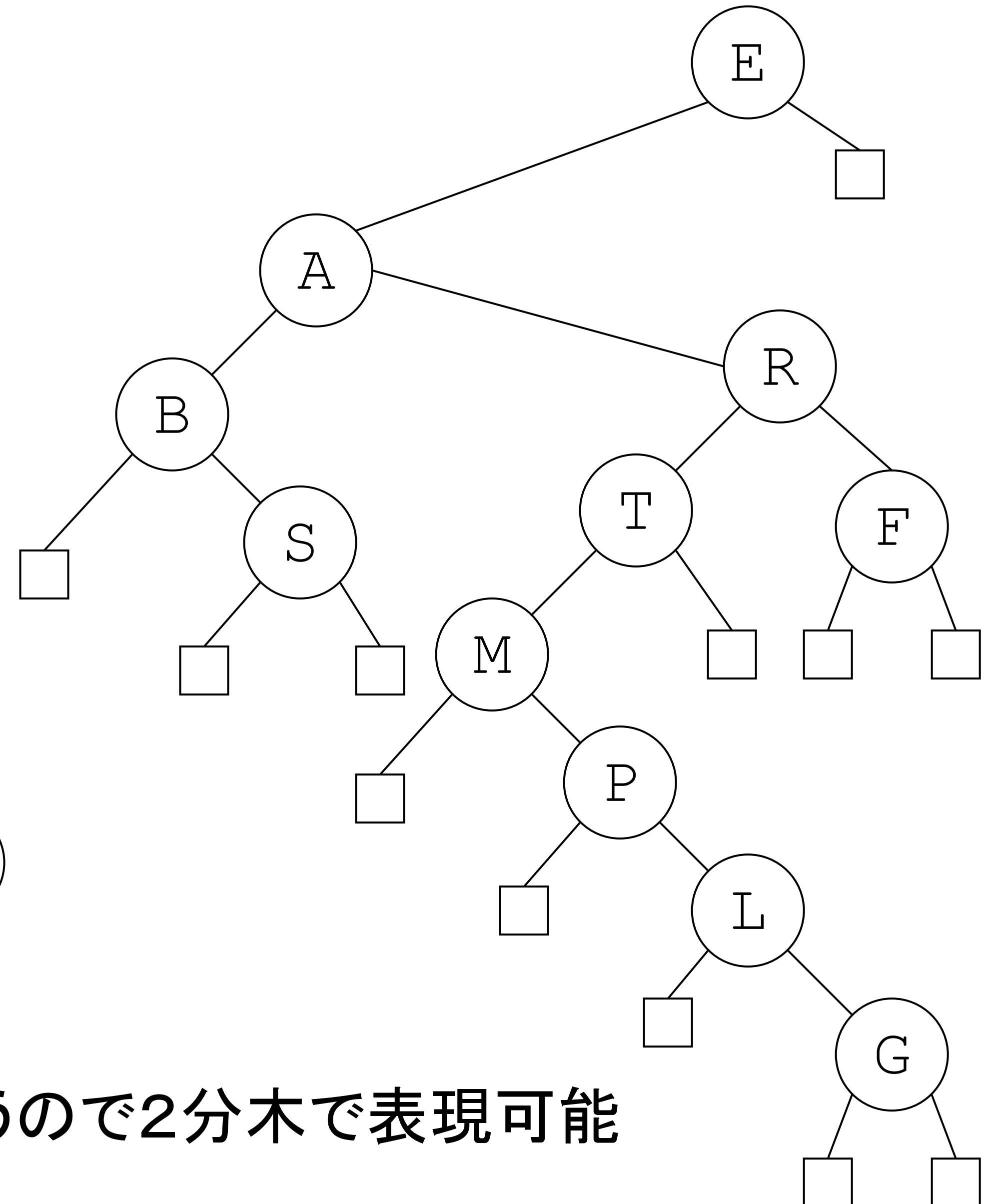
各ノードで
子へのリンクと
兄弟へのリンク
(末っ子は親へのリンク)
を使う方式

上から下への移動も容易



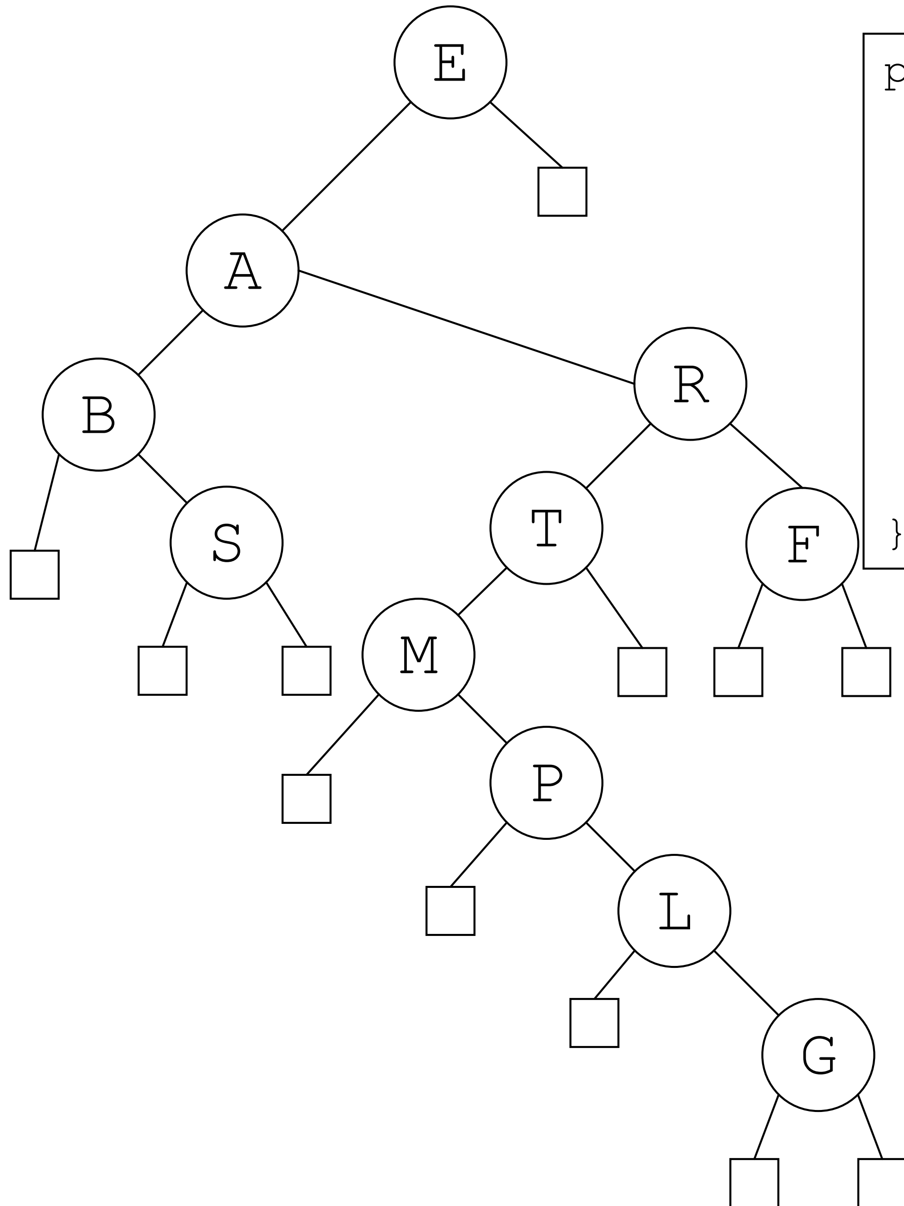
2分木による一般の木の表現





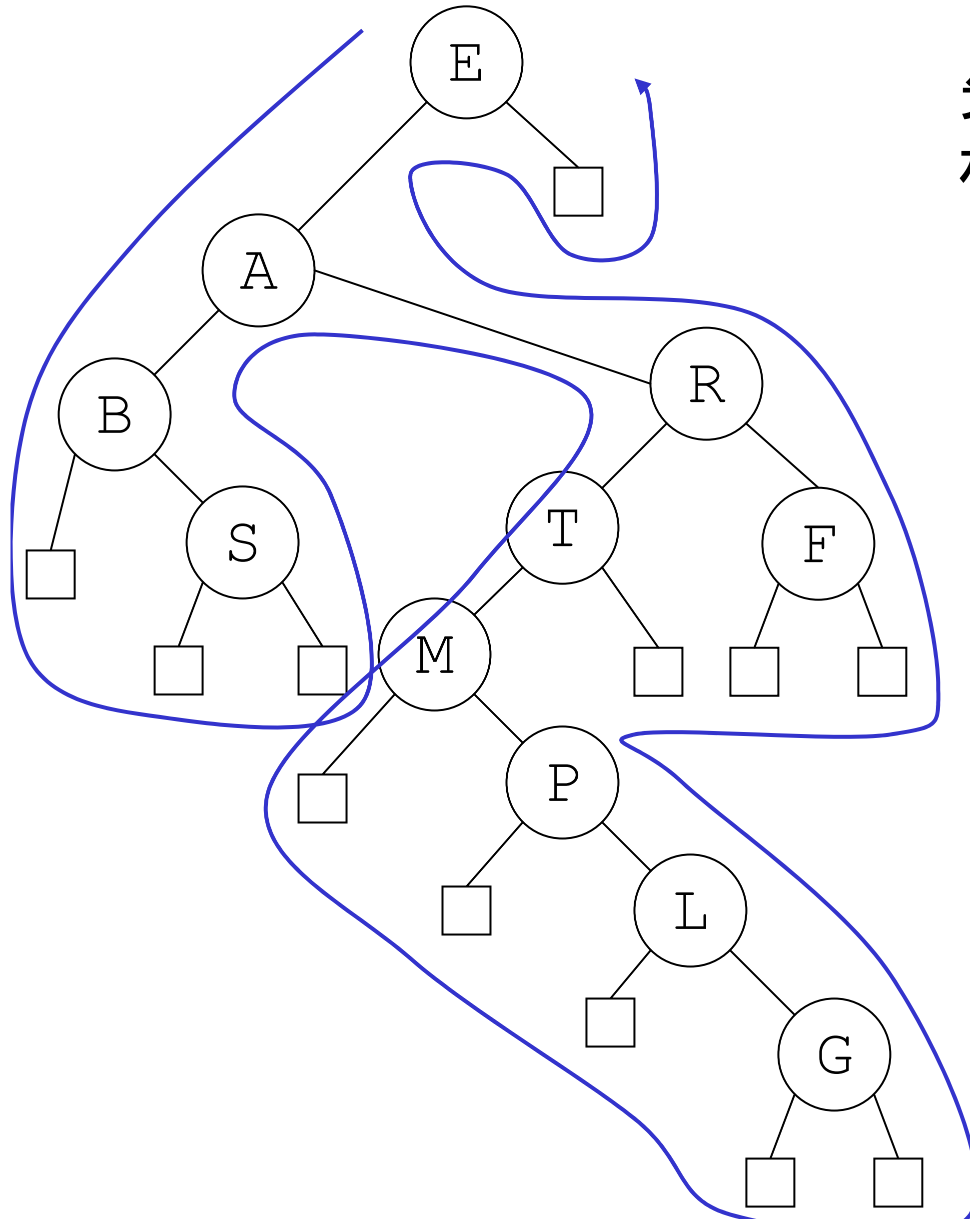
2つのリンクを使うので2分木で表現可能

2分木のデータ構造




```
public class Node {
    public char info;
    public Node left, right;

    public Node(char c, Node x, Node y){
        info = c;
        left = x;
        right = y;
    }
}
```

木の走査



先行順
根→左部分木→右部分木

```
public class Traverse {

   private static void traverse(Node r){
      if (r != null) {
         System.out.print(r.info + " ");
         traverse(r.left);
         traverse(r.right);
               }
   }
}
```

木の再帰的走査　スタックによる実装


```
public class Stack {
   final int MAX = 100;
   private Node[] stack;
   private int p;
   public Stack() {
        stack = new Node[MAX + 1];
        p = 0;
   }
   public void push(Node v) {
        if (p < MAX) {
           stack[p++] = v;
        }
   }
   public Node pop() {
        if ( !stackEmpty() ) {
           return stack[--p];
        }else {
           return null;
        }
   }
   public boolean stackEmpty(){
        return (p == 0);
   }
```

先行順 根→左部分木→右部分木

```
public void traverse(Node r){
   push(r);

   while( !stackEmpty() ){
        Node t = pop();
        System.out.print(t.info);

        if(t.right != null){
           push(t.right);
        }

        if(t.left != null){
           push(t.left);
        }
   }
}
```

5章　再帰呼び出し

漸化式と再帰的呼出し

フィボナッチ数列

fib(0) = fib(1) = 1

fib(N) = fib(N-2) + fib(N-1)　　N>=2

1 1 2 3 5 8 13 21 34 55 89 144 ...

```
private static int fib(int n){
        if(n <= 1){
                return 1;               // 負の数も 1
        }
        count++;
        return fib(n-2) + fib(n-1);
}
```

無駄な計算の重複

fib(4)

fib(2) + fib(3)

(fib(0)+fib(1)) + (fib(1)+fib(2))

(fib(0)+fib(1)) + (fib(1)+(fib(0)+fib(1)))

fib(N) を計算するのに
fib(N)-1 回 fib を呼び出す無駄

for ループでの書き換え


```
private static int iterFib(int n){
        if(n <= 1){
                return 1;       // 負の数も 1
        }
        int p2 = 1;             // 2つ前
        int p1 = 1;             // 1つ前
        int tmp;
        for(int i = 2; i <= n; i++){
                tmp = p1;
                p1 = p2 + p1;
                p2 = tmp;
                count++;
        }
        return p1;
}
```

1 1 2 3 5 8 13 21 34 55 89 144 ...

再帰的呼び出しの例　分割数

自然数（1以上の整数と定義）n を
いくつかの自然数の和として表現する仕方の個数を分割数 pn(n)

```
pn(2) = 2   2, 1+1
pn(3) = 3   3, 2+1, 1+1+1
pn(4) = 5   4, 2+2, 3+1, 2+1+1, 1+1+1+1
pn(5) = 7   5, 3+2, 4+1, 2+2+1, 3+1+1, 2+1+1+1, 1+1+1+1+1

pn(2) = 2                                2,       1+1
pn(3) = 3                       3,       2+1,     1+1+1
pn(4) = 5          4,    2+2,   3+1,     2+1+1,   1+1+1+1
pn(5) = 7   5, 3+2, 4+1, 2+2+1, 3+1+1, 2+1+1+1, 1+1+1+1+1
```

再帰的呼び出しの例　分割数

pn(n, i) : n を i 個の自然数の和で表現する場合の数

pn(n, 1) = 1　　n
pn(n, n) = 1　　1 + 1 + ... + 1

$1 < i < n$ のとき

- 分割された自然数のうち少なくとも一つは1である場合
  (n-1 を i-1 個に分割した和) + 1 の形なので分割数は pn(n-1, i-1)

- 分割された自然数がすべて 2 以上の場合
  $n = x1 + x2 + ... + xi$ ならば、
  $n - i = (x1 - 1) + (x2 - 1) + ... + (xi - 1)$ へ変形可能
  分割数は pn($n - i$, i)
  $n - i < i$ となる可能性があるので $n < i$ ならば pn(n, i) = 0 と約束

pn(n, 1) = 1, pn(n, n) = 1
pn(n, i) = pn(n-1, i-1) + pn(n-i, i) ただし $1 < i < n$ のとき
pn(n, i) = 0　ただし $n < i$ のとき
pn(n) = pn(n, 1) + pn(n, 2) + ... + pn(n, n)

再帰的呼び出しの例　分割数


```
public static int pn(int n){
        int sum = 0;
        if(n > 0){
                for(int i=1; i<=n; i++)
                        sum += pn(n,i);
        }
        return sum;
}
public static int pn(int n, int i){
        if(n < i)
                return 0;
        if(n == i || i == 1)
                return 1;
        return pn(n-i,i)+pn(n-1,i-1);
}
```

分割数　for ループでの書き換え

i →

n ↓	1	2	3	4	5	6	7	8	9	10	分割数
1	1										1
2	1	1									2
3	1	1	1								3
4	1	2	1	1							5
5	1	2	2	1	1						7
6	1	3	3	2	1	1					11
7	1	3	4	3	2	1	1				15
8	1	4	5	5	3	2	1	1			22
9	1	4	7	6	5	3	2	1	1		30
10	1	5	8	9	7	5	3	2	1	1	42

$\mathrm{pn}(n, i)$

$\mathrm{pn}(n, 1) = 1, \mathrm{pn}(n, n) = 1$

$1 < i < n$ のとき
$\mathrm{pn}(n, i) = \mathrm{pn}(n-1, i-1) + \mathrm{pn}(n-i, i)$

$n < i$ のとき
$\mathrm{pn}(n, i) = 0$

$\mathrm{pn}(n) = \mathrm{pn}(n, 1) + \ldots + \mathrm{pn}(n, n)$

分割数　for ループでの書き換え


```
// 再帰的呼出を使わないで iteration で計算する場合
int[][] a = new int[MAX+1][MAX+1];
        // int[0][*], int[*][0], 上三角成分を利用しない無駄な使い方ではある

for(int n=1; n<=MAX; n++){
        for(int i=1; i<=n; i++){
                if(i == 1 || i == n){
                        a[n][i] = 1;
                }else{
                        if(n-i < i){    // a[n-i][i]==0 として加算しない
                                a[n][i] = a[n-1][i-1];
                        }else{
                                a[n][i] = a[n-i][i] + a[n-1][i-1];
                        }
                }
        }
        int answer = 0;
        for(int j=1; j<=n; j++)
                answer += a[n][j];
        System.out.println(answer);
}
```

$\mathrm{pn}(n, 1) = 1, \mathrm{pn}(n, n) = 1$

$1 < i < n$ のとき

$\mathrm{pn}(n, i) = \mathrm{pn}(n\text{-}1, i\text{-}1) + \mathrm{pn}(n\text{-}i, i)$

$n < i$ のとき

$\mathrm{pn}(n, i) = 0$

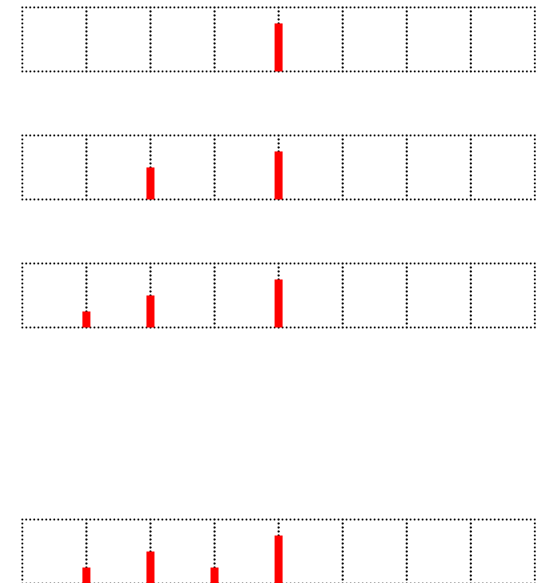
$\mathrm{pn}(n) = \mathrm{pn}(n, 1) + \ldots + \mathrm{pn}(n, n)$

分割統治 (divide-and-conquer)の再帰的呼出しによる記述

分割統治： 小さな問題に分割（通常は2分割）して解き、あとでまとめる

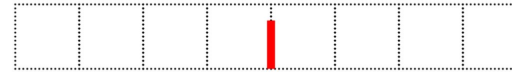
```
static void rule
    (int l, int r, int h){

    int m = (l+r)/2;

    if(h > 0){
        mark(m, h);
        rule(l, m, h-1);
        rule(m, r, h-1);
    }
}
```

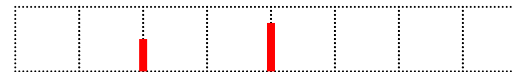
```
rule(0,8,3)
  mark(4,3)
  rule(0,4,2)
    mark(2,2)
    rule(0,2,1)
      mark(1,1)
      rule(0,1,0)
      rule(1,2,0)
    rule(2,4,1)
      mark(3,1)
      rule(2,3,0)
      rule(3,4,0)
  rule(4,8,2)
```

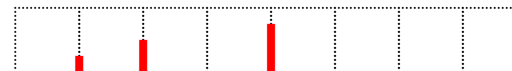


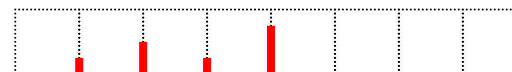
分割統治

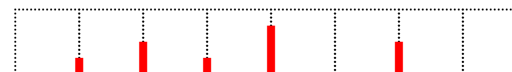

```
rule(0,8,3)
    :
    :
    :
    :
    :
    :
  rule(4,8,2)
    mark(6,2)
    rule(4,6,1)
      mark(5,1)
      rule(4,5,0)
      rule(5,6,0)
    rule(6,8,1)
      mark(7,1)
      rule(6,7,0)
      rule(7,8,0)
```

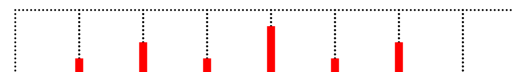










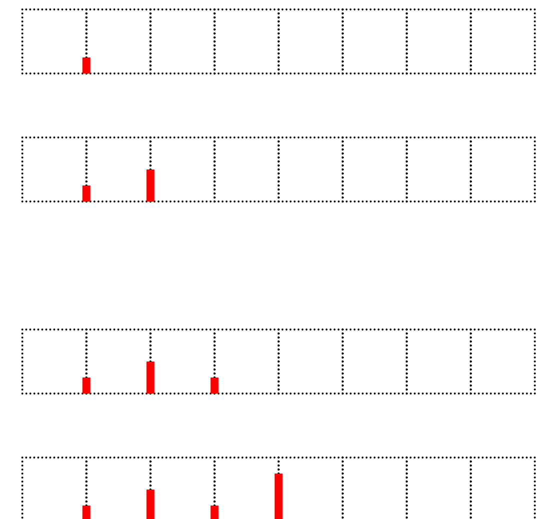


分割統治

目盛を書く順番を変更しても結果に変化なし

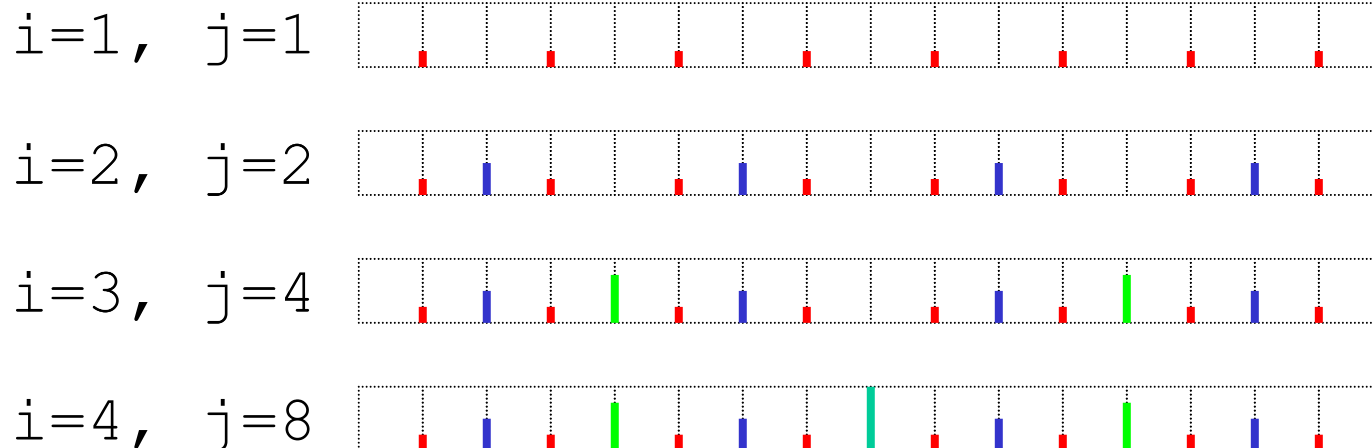
```
static void rule
    (int l, int r, int h){

    int m = (l+r)/2;

    if(h > 0){
        rule(l, m, h-1);
        mark(m, h);
        rule(m, r, h-1);
    }
}
```

```
rule(0,8,3)
  rule(0,4,2)
    rule(0,2,1)
      rule(0,1,0)
      mark(1,1)
      rule(1,2,0)
    mark(2,2)
    rule(2,4,1)
      rule(2,3,0)
      mark(3,1)
      rule(3,4,0)
  mark(4,3)
  rule(4,8,2)
```



for ループでの書き換え


```
int l = 0;
int r = 16;
int h = 4;
int i,j;
for(i=1, j=1; i <= h; i++, j+=j){
        // i=1,2, ... ,h  j=1,2,4,8, ...
        for(int t = 0; t < (l+r)/(2*j); t++){
                mark(l+j+2*j*t, i);
        }
}
```



６章　アルゴリズムの解析

計算時間の評価

平均の場合
典型的なデータに対して期待される計算時間

最悪の場合
最もたちの悪い種類の入力に対して必要な計算時間

アルゴリズムの分類

N をデータ数　　関数 $f(N)$ に比例する計算時間

$f(N)$　　$\lg N, \lg^2 N, N^{1/2}, N, N \lg N, N \lg^2 N, N^2, N^3, 2^N$

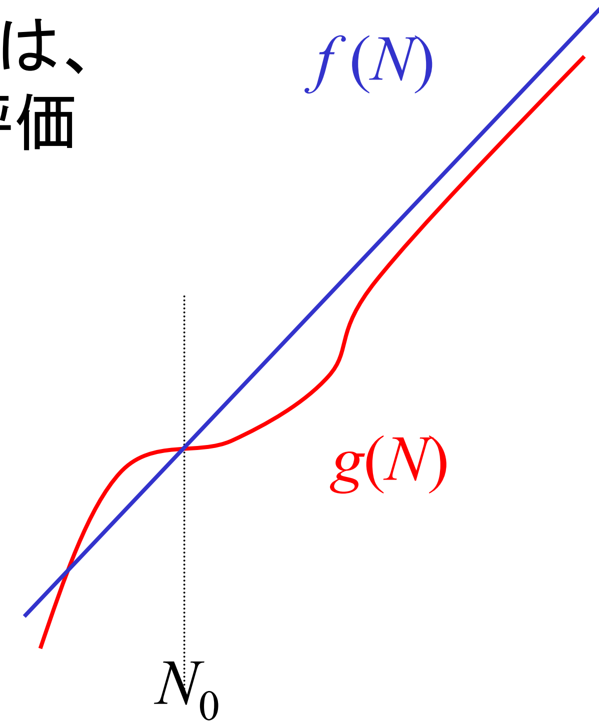
ただし　$\log_2 N = \lg N$,　$\log_e N = \ln N$

「比例する」計算時間　　定数倍は気にしない
log の底は 2 でも自然対数の底 e でもよい

最悪計算量

アルゴリズム性能を理論的に解析するさいには、
定数係数を無視して、最悪の場合の性能を評価

厳密に数学的結果を証明できる良さ



「比例する」という言葉の定義

O-notation （計算時間の漸近的上界を示す）

ある定数 c_0 と N_0 が存在して、 $N > N_0$ である任意の N に対して $g(N) < c_0 f(N)$ ならば、関数 $g(N)$ は $O(f(N))$
（オーダー $f(N)$ と読む ）であるという。

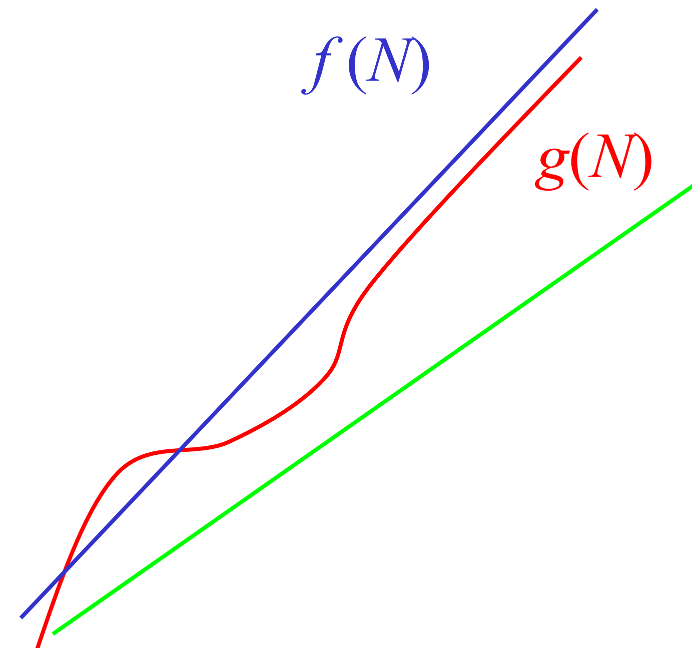
最悪計算量

$O(f(N))$ は上界を示したに過ぎない

$g(N)$ が $f(N)$ より小さいとは

$$\lim_{N \to \infty} g(N) / f(N) = 0$$

$f(N)$ より小さい $h(N)$ が存在して
$g(N)$ は $O(h(N))$ であると、
$f(N)$ はギリギリの上界でない



良質な上界を
得るのは難しい

しかし、このような $h(N)$ が存在しない
ことを証明するのは通常難しい

計算量の下界

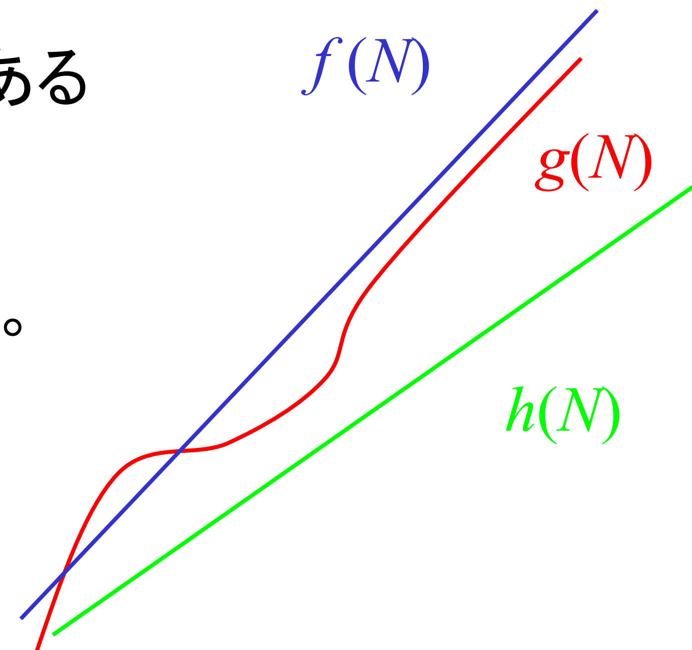
Ω-notation （計算時間の漸近的下界を示す）

ある定数 c_0 と N_0 が存在して、 $N > N_0$ である任意の N に対して

$$0 \leqq c_0 h(N) \leqq g(N)$$

ならば、関数 $g(N)$ は Ω$(h(N))$であるという。

良質な下界を得るのも難しい



漸近的な差と実用性

N	$(1/4)N\lg^2 N$	$(1/2)N\lg^2 N$	$N\lg^2 N$	$N^{3/2}$
10	22	45	**90**	30
100	900	1,800	**3,600**	1,000
1,000	20,250	40,500	**81,000**	31,000
10,000	422,500	845,000	**1,690,000**	1,000,000
100,000	6,400,000	12,800,000	25,600,000	**31,600,000**
1,000,000	90,250,000	180,500,000	361,000,000	**1,000,000,000**

平均の場合の解析

命令時間の正確な見積りが難しい

コンパイラ、ハードウエアによる命令の並列実行など

解析が複雑になりがち　定数の正確な見積り

漸近的計算量は定数を無視することで問題を簡単にしている

入力モデルが現実を反映していない場合が多い

ランダムに生成されたデータはプログラムに容易な例となりがち
現実にはデータは偏っている

O 記法(O-notation) による近似

$$a_0 N \lg N + a_1 N + a_2 \qquad a_0 N \lg N + O(N)$$

$$f(N) + O(g(N))$$

$f(N)$ が漸近的に $g(N)$ より大きくなるとき、
$f(N)+O(g(N))$ は約 $f(N)$ という

基本漸化式

$$C(N) = C(N-1) + N \quad N \geq 2$$
$$C(1) = 1$$

$$C(N) = \sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

$$C(N) = C(N/2) + 1 \quad N \geq 2$$
$$C(1) = 0$$

$N = 2^n$の場合

$$C(2^n) = C(2^{n-1}) + 1 = C(2^0) + n = n$$

$2^n \leq N < 2^{n-1}$の場合

$$n \leq C(N) < n+1$$

$$C(N) = C(N/2) + N \quad N \geq 2$$
$$C(1) = 0$$

$$C(N) = N + N/2 + N/4 + \ldots \approx 2N$$

$$C(N) = 2C(N/2) + N \quad N \geq 2$$
$$C(1) = 0$$

$N = 2^n$のときNで両辺を割る

$$\frac{C(2^n)}{2^n} = \frac{C(2^{n-1})}{2^{n-1}} + 1$$

$$= \frac{C(2^{n-2})}{2^{n-2}} + 1 + 1 = \ldots$$

$$= n = \lg N$$

$$C(N) = N \lg N$$

８章　初等的な整列法

Selection Sort

最小の元を選択

BIOINFORMATICS
AIOINFORMBTICS
ABOINFORMITICS
ABCINFORMITIOS
ABCFNIORMITIOS
ABCFINORMITIOS
ABCFIIORMNTIOS
ABCFIIIRMNTOOS
ABCFIIIMRNTOOS
ABCFIIIMNRTOOS
ABCFIIIMNOTROS
ABCFIIIMNOORTS
ABCFIIIMNOORTS
ABCFIIIMNOORST

約$N^2/2$回の比較と最悪約N回の交換

```
public void selection() {
    for (int i = 0; i < N - 1; i++) {
        int min = i;
        for (int j = i + 1; j < N; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        // a[i], a[min] を交換
        int tmp = a[min];
        a[min] = a[i];
        a[i] = tmp;
    }
}
```

Bubble Sort

隣り合う元の大小関係を修正　最小の元は左側に

```
BIOINFORMATICS
ABIOINFORMCTIS
ABCIOINFORMITS
ABCFIOINIORMST
ABCFIIOINMORST
ABCFIIIOMNORST
ABCFIIIMONORST
ABCFIIIMNOORST
ABCFIIIMNOORST
ABCFIIIMNOORST
ABCFIIIMNOORST
ABCFIIIMNOORST
ABCFIIIMNOORST
ABCFIIIMNOORST
```

```
public void bubble() {
    for (int i = 0; i < N; i++) {
        for (int j = N - 1; i < j; j--) {
            if (a[j - 1] > a[j]) {
                // a[j-1] と a[j] を交換
                int tmp = a[j - 1];
                a[j - 1] = a[j];
                a[j] = tmp;
            }
        }
    }
}
```

約$N^2/2$回の比較と
最悪約$N^2/2$回の交換

Insertion Sort

整列された列に新たな元を挿入

BIOINFORMATICS
BIOINFORMATICS
BIOINFORMATICS
BIIONFORMATICS
BIINOFORMATICS
BFIINOORMATICS
BFIINOORMATICS
BFIINOORMATICS
BFIIMNOORATICS
ABFIIMNOORTICS
ABFIIMNOORTICS
ABFIIIMNOORTCS
ABCFIIIMNOORTS
ABCFIIIMNOORST

最悪約$N^2/2$回の比較と$N^2/4$回の交換
計算コストが最悪の例と最小の例は？

```
public void insertion() {
    for (int i = 1; i < N; i++) {
        int v = a[i];
        int j;
        for (j = i;
             0 < j && v < a[j - 1];
             j--) {
            a[j] = a[j - 1];
        }
        a[j] = v;
    }
}
```

insertion sort
要素の交換数が多くなる傾向
shellsort
離れた要素間での交換を予備的に実行

Shellsort　h要素分はなれた要素の集まりを insertion sort

h=13

BIOINFORMATICS

h=4

BIOINFORMATICS
BIOINFORMATICS
BFOINIORMATICS
BFOINIORMATICS
BFOINIORMATICS
BFOIMIORNATICS
BAOIMFORNITICS
BAOIMFORNITICS
BAOIMFOINITRCS
BAOICFOIMITRNS

h=1

BAOICFOIMITRNS
ABOICFOIMITRNS
ABOICFOIMITRNS
ABIOCFOIMITRNS
ABCIOFOIMITRNS
ABCFIOOIMITRNS
ABCFIOOIMITRNS
ABCFIIOOMITRNS
ABCFIIMOOITRNS
ABCFIIIMOOTRNS
ABCFIIIMOOTRNS
ABCFIIIMOORTNS
ABCFIIIMNOORTS
ABCFIIIMNOORST

Shellsort


```
public void shell() {
    int h;
    int f = 3;
    for (h = 1; h <= N / f; h = f * h + 1);
    // h = 1, 4, 13, 40, 121, ... < N

    for (; 0 < h; h /= f) {
            // h を f で割った商(整数部分)を h に代入.
            // h = ..., 121, 40, 13, 4, 1

        for (int i = h; i < N; i++) {
            int v = a[i];
            int j;
            for (j = i;  h <= j && v < a[j - h]; j = j - h) {
                a[j] = a[j - h];
            }
            a[j] = v;
        }
    }
}
```

9章　Quick Sort

分割統治法によるソート


```
public void quick(){
    quick(0, N-1);
}

public void quick(int l, int r) {
    if (l < r) {
        int i = partition(l, r);
        quick(l, i - 1);
        quick(i + 1, r);
    }
}
```

```
i = partition(l,r)
```

a[]を以下のように分割

- a[l],...,a[i-1] は a[i] を超えない
- a[i+1],...,a[r] は a[i] を下回らない

BIOINFORMATI**C**S

BIOINFORMA**C**I**T**S

BIOINFORMACI**ST**

B**I**OINFORMA**CI**

B**C**OINFORMA**I**I

BC**O**INFORM**A**II

BC**A**INFORM**O**II

BCA**I**N**F**ORMOII

BCA**F**N**I**ORMOII

BCAF**I**IORMOI**N**

partition の実装


```
public int partition(int l, int r) {
    int v = a[r];
    int i = l - 1; // i は左側から
    int j = r; // j は右側から
    int tmp; // 値の退避用
    for (;;) {
        // v より小さい左の要素はとばす
        while (a[++i] < v) {}
        // v より大きい右の要素はどばす
        while (v < a[--j] && i < j){}
        // 交差したら終了
        if (i >= j) break;
        // a[i] と a[j] を交換
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
    tmp = a[i];
    a[i] = a[r];
    a[r] = tmp;
    return i;
}
```

S ≦ → ← ≦ S

BIOINFORMATICS

BIOINFORMACITS

BIOINFORMACIST

I ≦ → ← ≦ I

BIOINFORMACI

BCOINFORMAII

BCOINFORMAII

BCAINFORMOII

BCAINFORMOII

BCAFNIORMOII

BCAFIIORMOIN

partition の実装


```
public int partition(int l, int r) {
    int v = a[r];
    int i = l - 1; // i は左側から
    int j = r; // j は右側から
    int tmp; // 値の退避用
    for (;;) {
        // v より小さい左の要素はとばす
        while (a[++i] < v) {}
        // v より大きい右の要素はどばす
        while (v < a[--j] && i < j){}
        // 交差したら終了
        if (i >= j) break;
        // a[i] と a[j] を交換
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
    tmp = a[i];
    a[i] = a[r];
    a[r] = tmp;
    return i;
}
```

S ≦ → ← ≦ S

BIOINFORMACITS

i

j

BIOINFORMACIST

I ≦ → ← ≦ I

BCAFNIORMOII

i

j

BCAFIIORMOIN

quick sort の動作例

BCAF

BCA F

ACB F

A CB F

A BC F

IIORMOIN

IIIRMOON

IIIRMOON

IIIMROON

IIIMNOOR

IIIM NOOR

A BC F IIIM NOOR ST

性能

最悪の場合
a[] が既にソートされている
$N^2/2$ に比例する計算時間

A B C D E F G H I J
A B C D E F G H I
A B C D E F G H
:

配列が常に丁度約半分に分割される場合： 比較回数 C_N

$$C_N = 2C_{N/2} + N \approx N \lg N$$

$$\lg = \log_2 \quad \ln = \log_e$$

性能

ランダムな配列では 平均 $2N \ln N$ 回の比較

$$C_1 = C_0 = 0$$

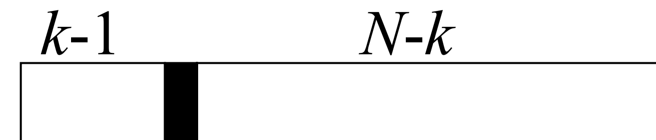
$$C_N = N + 1 + \frac{1}{N} \sum_{1 \le k \le N} \left(C_{k-1} + C_{N-k} \right)$$

$C_0 + \cdots + C_{N-1} = C_{N-1} + \cdots C_0$ なので

$$C_N = N + 1 + \frac{2}{N} \sum_{1 \le k \le N} C_{k-1}$$

両辺にNを掛ける

$$NC_N = N(N+1) + 2 \sum_{1 \le k \le N} C_{k-1}$$



k-1 と N-k に分割される確率は $1/N$

性能

$$NC_N = N(N+1) + 2\sum_{1\le k\le N} C_{k-1}$$

$N-1$の場合

$$(N-1)C_{N-1} = (N-1)N + 2\sum_{1\le k\le N-1} C_{k-1}$$

引く

$$NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}$$

$$NC_N = (N+1)C_{N-1} + 2N$$

$N(N+1)$で割る

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1} = \cdots = \frac{C_2}{3} + \sum_{3\le k\le N} \frac{2}{k+1}$$

$$\approx 2\sum_{1\le k\le N} \frac{1}{k} \approx 2\int_1^N \frac{1}{x}dx = 2\ln N = 1.38\lg N$$

$$C_N \approx 2N\ln N$$

再帰呼出しの除去　スタックを使った実装


```
public void iterQuick(){    // iteration で quick sort を実現
    int l = 0;       // 左端は 0
    int r = N-1;     // 右端は N-1
    Stack s = new Stack(2*N);
    for( ; ; ){
        while(l < r){
            int i = partition(l, r);
            if(i-l > r-i){       // 大きい方の分割を push
                s.push(l);       // 大きい分割がスタックに積まれるのは高々log N
                s.push(i-1);
                l = i+1;
            }else{
                s.push(i+1);
                s.push(r);
                r = i-1;
            }
        }
        if(s.stackEmpty())
            break;
        r = s.pop();             // 積んだ順番の逆順に取り出す
        l = s.pop();
    }
}
```

なぜ大きい方の分割をスタックに push するのか？

スタックの節約

大きい分割がスタックに積まれる回数は高々約 $\lg N$

スタックに積まれる最初の分割の大きさは　$N/2$ 以上
次は $N/4$ 以上
次は $N/8$ 以上
:

小さい分割をスタックに積むと、最悪約 N

小さい部分配列の処理

- 短い部分配列については `insertion sort` の方が実質的には速い
- `quicksort` と `insertion sort` の組合せが経験的には高速

分割用要素の選択

分割が偏る

配列の最後の要素？

要素をランダムに選択
偏りのない選択
（乱数生成ルーチン）

３つの要素を選び
中央値を利用

BIOINFORMATICS
BIOINFORMACITS
BIOINFORMACIST

BIOINFORMACI
BCOINFORMAII
BCOINFORMAII
BCAINFORMOII
BCAINFORMOII
BCAFNIORMOII
BCAFIIORMOIN

k番目 (k=0,1,...) に小さい要素を計算する方法

配列をソートせずに計算

選択整列法の改良
最初の k 回目までを利用
Nk に比例する時間

平均で N に比例する時間
で計算する方法

quick sort の変更
観察: i = partition(l,r) で
i が l から k 番目の要素 (i=l+k-1) ならば終了

BIOINFORMATICS
AIOINFORMBTICS
ABOINFORMITICS
ABCINFORMITIOS
ABCFNIORMITIOS

$k=5$

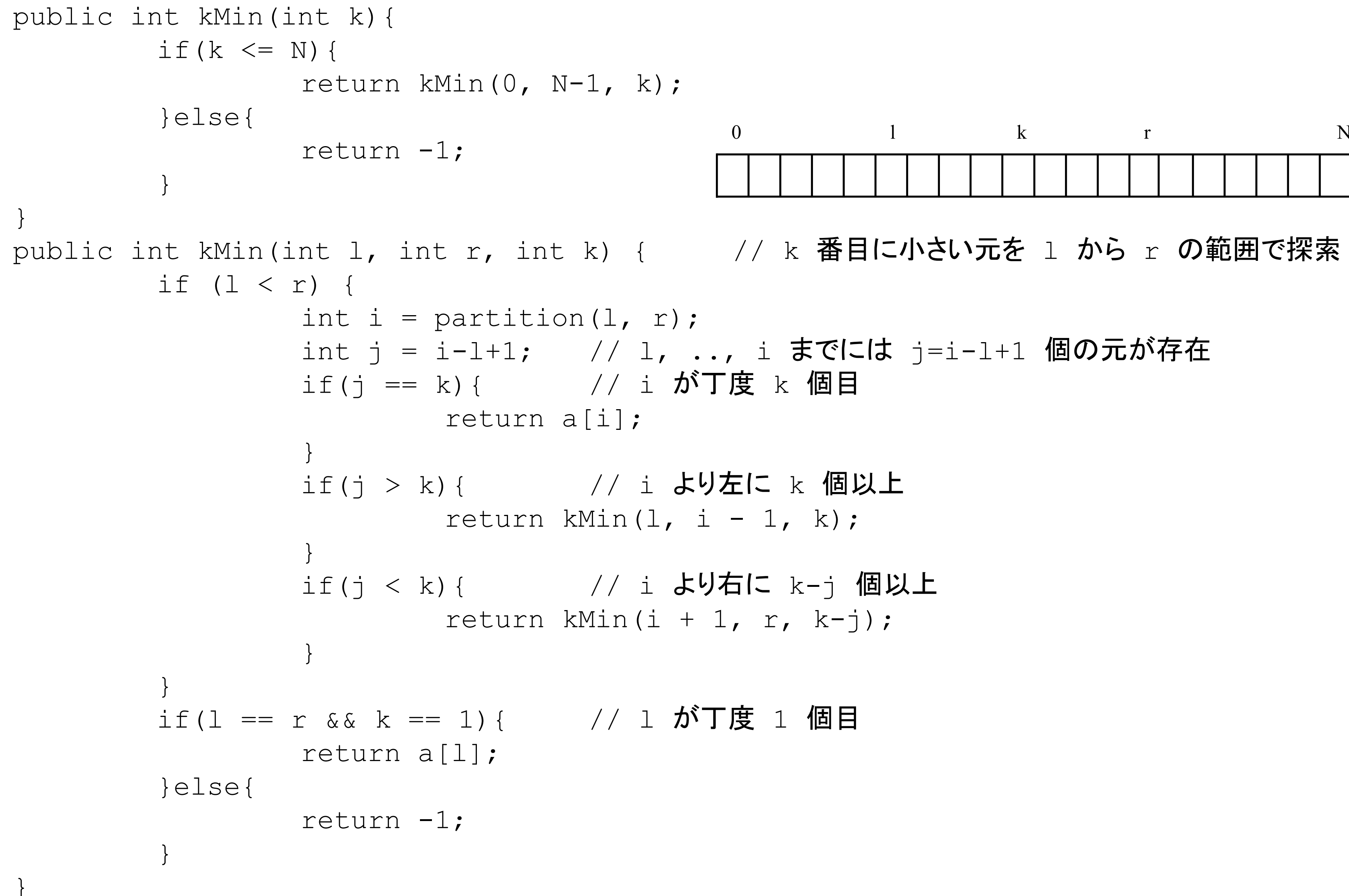
ABCFINORMITIOS
ABCFIIORMNTIOS
ABCFIIIRMNTOOS
ABCFIIIMRNTOOS
ABCFIIIMNRTOOS
ABCFIIIMNOTROS
ABCFIIIMNOORTS
ABCFIIIMNOORTS
ABCFIIIMNOORST

k番目に小さい要素を計算する方法


```
                 0       8    13
kMin(0, 13, 8)   BIOINFORMATICS
                       :
                 BIOINFORMACIST
kMin(0, 11, 8)   BIOINFORMACI
                       :
                 BCAFIIORMOIN
kMin(4, 11, 4)       IIORMOIN
                       :
                     IIIMNOOR
```

a[] のソートは未完成
k番目に小さい要素は求まる

k番目に小さい要素を計算する方法




```
public int kMin(int k){
        if(k <= N){
                return kMin(0, N-1, k);
        }else{
                return -1;
        }
}
public int kMin(int l, int r, int k) {      // k 番目に小さい元を l から r の範囲で探索
        if (l < r) {
                int i = partition(l, r);
                int j = i-l+1;    // l, .., i までには j=i-l+1 個の元が存在
                if(j == k){       // i が丁度 k 個目
                        return a[i];
                }
                if(j > k){        // i より左に k 個以上
                        return kMin(l, i - 1, k);
                }
                if(j < k){        // i より右に k-j 個以上
                        return kMin(i + 1, r, k-j);
                }
        }
        if(l == r && k == 1){     // l が丁度 1 個目
                return a[l];
        }else{
                return -1;
        }
}
```

遺伝子発現解析

- 生物情報実験２
- 遺伝子発現量パターンが近い順序でソート
- １からk番目まで近いパターンを列挙

10章　Radix Sort

ビット演算

c の右から b ビット目を値として返す

```
B 00010
I 01001
O 01111
I 01001
N 01110
F 00110
O 01111
C 00011
M 01101
A 00001
I 01001
T 10100
R 10010
S 10011
```

```
public int bits(int c, int b){
    int x = c;
    int y = 0;
    for(int i = 0; i < b; i++){
        y = x%2;
        x = x/2;
    }
    return y;
}
```

```
    87654321
c = 00101001    bits(c,6)
```

基数交換法 (radix exchange sort)

ビットを左から右へと比較

1	2	3	4	5	6
B00010	B00010	B00010	B00010	A00001	A00001
I01001	I01001	A00001	A00001	B00010	B00010
O01111	O01111	C00011	C00011	C00011	C00011
I01001	I01001	F00110	F00110	F00110	F00110
N01110	N01110	N01110	I01001	I01001	I01001
F00110	F00110	I01001	I01001	I01001	I01001
O01111	O01111	O01111	I01001	I01001	I01001
R10010	C00011	O01111	O01111	M01101	M01101
M01101	M01101	M01101	M01101	O01111	N01110
A00001	A00001	I01001	O01111	O01111	O01111
T10100	I01001	I01001	N01110	N01110	O01111
I01001	T10100	T10100	S10011	S10011	S10011
C00011	R10010	R10010	R10010	R10010	R10010
S10011	S10011	S10011	T10100	T10100	T10100

radix exchange sort の実装


```
public void radixExchange(){
    radixExchange(0, N-1, 5);
}
public void radixExchange(int l, int r, int b){
    if(l < r && b > 0){
        int i = l;                  // i は左側から
        int j = r;                  // j は右側から
        while(i < j){
            while( bits(a[i], b) == 0 && i < j ){
                i++;                // 0 のうちは i は右へ
            }
            while( bits(a[j], b) == 1 && i < j ){
                j--;                // 1 のうちは j は左へ
            }
            int tmp = a[i];         // a[i] == 1 と a[j] == 0 を交換
            a[i] = a[j];
            a[j] = tmp;
        }
        if( bits(a[r], b) == 0 ){ j++; }
        radixExchange(l, j-1, b-1);
        radixExchange(j, r, b-1);
    }
}
```

radix exchange sort の弱点

最初の何ビットかが、どの文字も同じだと無駄

```
0000 0000 0011 1010 1010
0000 0000 0010 0010 1011
0000 0000 0011 1110 1010
0000 0000 0001 1110 1011
:
```

比較するビットをランダムに選ぶことができれば
理想的には $\lg N$ ビット調べたところで終了
（N は配列の要素数）

直接基数法（straight radix sort） ビットを右から左へと比較

B00010	B00010	T10100	I01001	A00001	A00001
I01001	N01110	I01001	I01001	B00010	B00010
O01111	F00110	I01001	A00001	R10010	C00011
I01001	R10010	M01101	I01001	C00011	F00110
N01110	T10100	A00001	B00010	S10011	I01001
F00110	I01001	I01001	R10010	T10100	I01001
O01111	O01111	B00010	C00011	F00110	I01001
R10010	I01001	N01110	S10011	I01001	M01101
M01101	O01111	F00110	T10100	I01001	N01110
A00001	M01101	R10010	M01101	I01001	O01111
T10100	A00001	O01111	N01110	M01101	O01111
I01001	I01001	O01111	F00110	N01110	R10010
C00011	C00011	C00011	O01111	O01111	S10011
S10011	S10011	S10011	O01111	O01111	T10100

注目するビットの値が同じ文字は、既存の順番を保存して移動

straight radix sort の実装方針

a[]	b[]
B 00010	B 00010
I 01001	N 01110
O 01111	F 00110
I 01001	R 10010
N 01110	T 10100
F 00110	I 01001
O 01111	O 01111
R 10010	I 01001
M 01101	O 01111
A 00001	M 01101
T 10100	A 00001
I 01001	I 01001
C 00011	C 00011
S 10011	S 10011

分配計数法
対象ビットの値ごとにヒストグラムを生成

右端ビットの場合、各 a[i] に対して

```
count[bits(a[i], 1)]++;
```

bits(a[i], 1) は 0 か 1

ヒストグラム → 累積をとる

ヒストグラム	累積
count[0]=5	5
count[1]=9	14

- a[13]=S の右端は 1
  count[1]=14 なので b[14-1]=a[14];
  count[1]--;
- a[12]=C の右端は 1
  count[1]=13 なので b[13-1]=a[13];
  count[1]--;
- a[10]=T の右端は 0
  count[0]=5 なので b[5-1]=a[10]
  count[0]--;

straight radix sort の実装


```
public void straightRadix(){
        int nBits = 5;
        int[] b = new int[N];
        int[] count = new int[2];

        for(int pass = 1; pass <= nBits; pass++){
                count[0] = 0;
                count[1] = 0;
                for(int i = 0; i < N; i++){
                        count[bits(a[i], pass)]++;
                }
                count[1] += count[0];
                for(int i = N-1; 0 <= i; i--){
                        b[ count[bits(a[i], pass)]-1 ] = a[i];
                        // count[bits(a[i], pass)] より 1 小さくしないと
                        // 正確なインデックスにならない
                        count[bits(a[i], pass)]--;
                }
                for(int i = 0; i < N; i++){
                        a[i] = b[i];
                }
        }
}
```

基数整列法 (radix sort) の性能

基数整列法 (radix sort)
- 基数交換法 (radix exchange sort)
- 直接基数法 (straight radix sort)

基数交換法では平均約 N lg N ビットの比較
(quick sort と同様の解析)

どちらの基数整列法も b ビットのキー N 個の整列に
Nb 回以下のビットしか調べない

radix sort はゲノム配列のソートに向いている

ゲノム配列のソート

ACGTCATCGTCGATCGTACG …

A 00
T 01
G 10
C 11

A	C	G	T	C	A	T	C	G	T	…

0011100111000111001…

ヒトゲノムの部分配列（長さ50）のソートをするなら
- 最初の長さ10塩基前後で radix sort
- 最初の長さ10塩基前後が共通の部分配列を quick sort

11章　順序キュー

順序キューとは

キューは先着順に並んでおり、最初に入ったデータを最初に取出す
(FIFO) First-In-First-Out

順序キューはデータが降順（もしくは昇順）で並んでいるキュー

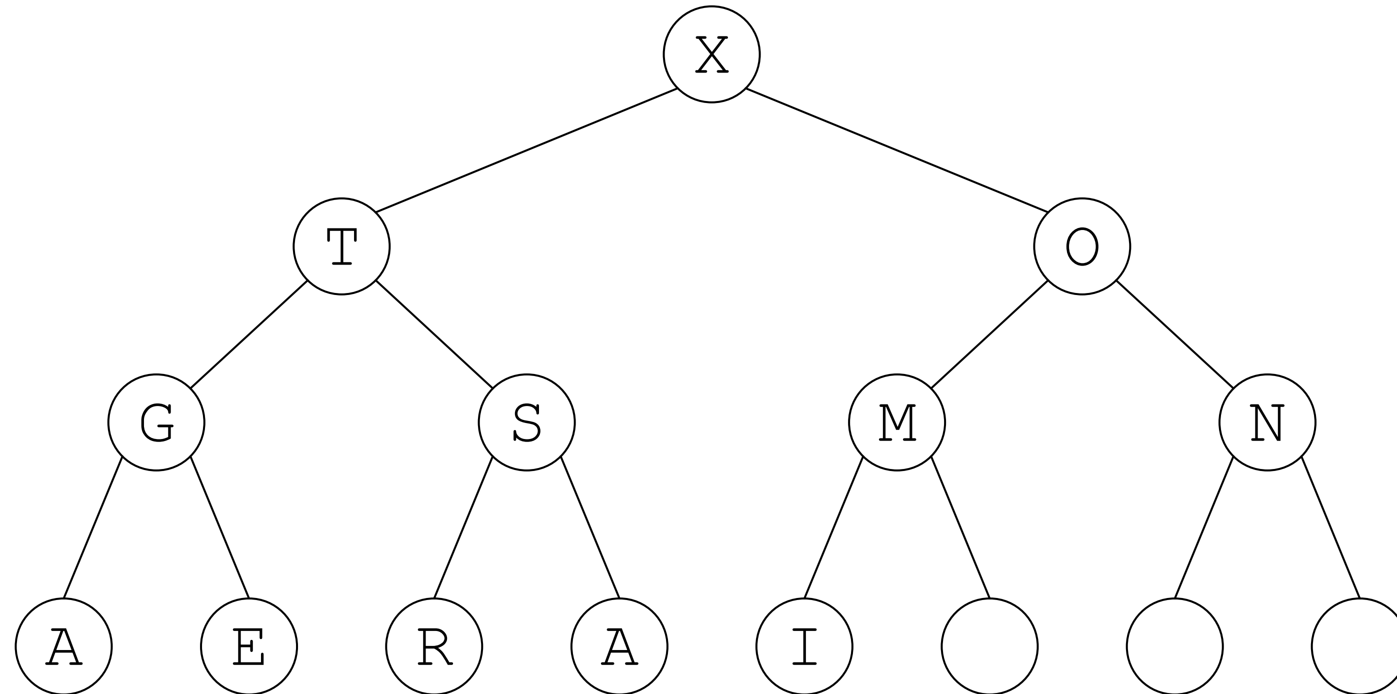
最大値を直ぐに取出せる

順序キューに入っているデータ総数を N とすれば、
データの追加と最大値の取出しが O(lg N) 時間で実行可能

オンライン処理向き

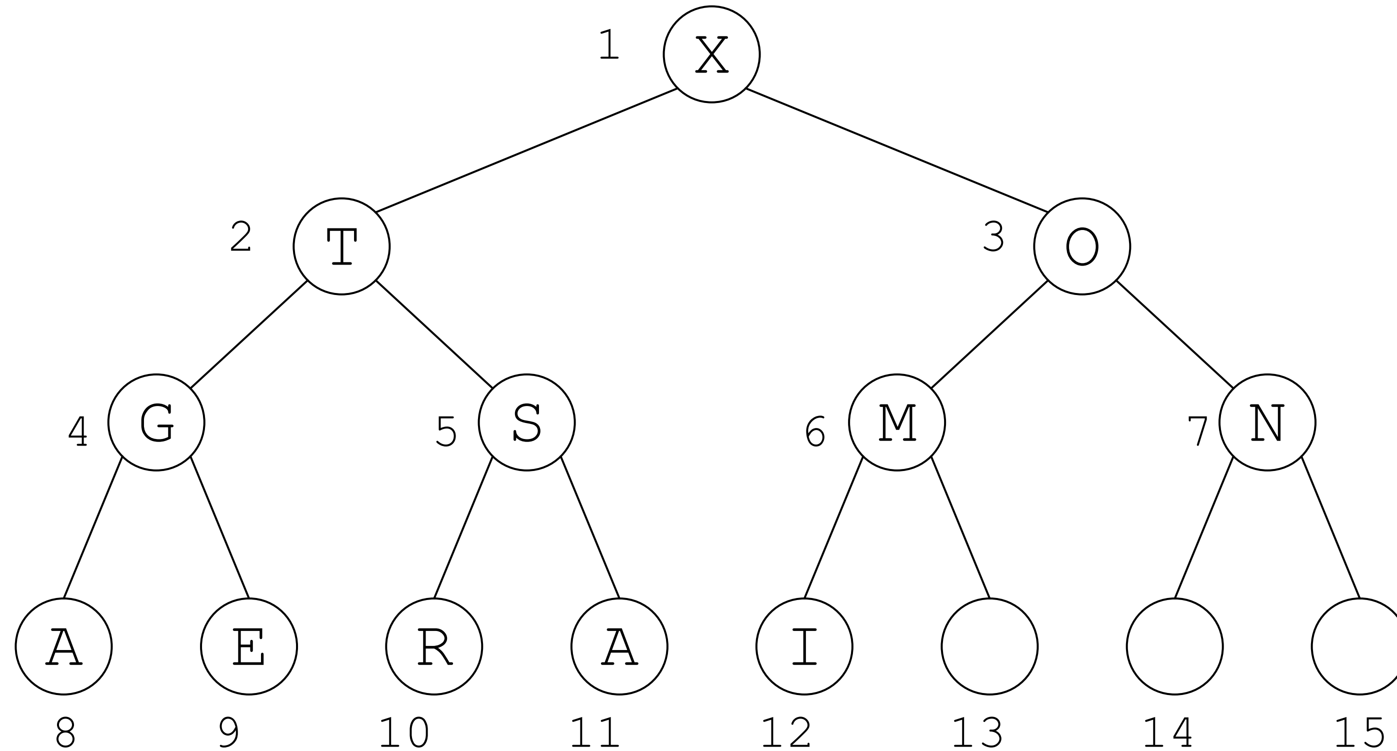
順序キューに N 個のデータを追加すれば自然にソートでき
O(N lg N) 時間で実行可能

ヒープ



- 完全2分木の各ノードを、上から下へ、左から右へデータを埋める
- 各ノードの値は、子ノードの値（存在すれば）より大きいか等しい
- 根ノードの値が最大
- たとえば G より大きい R がレベルが低くても問題にしない

ヒープの実装



k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	X	T	O	G	S	M	N	A	E	R	A	I			

N=12 N は値のある最後の場所のインデックス

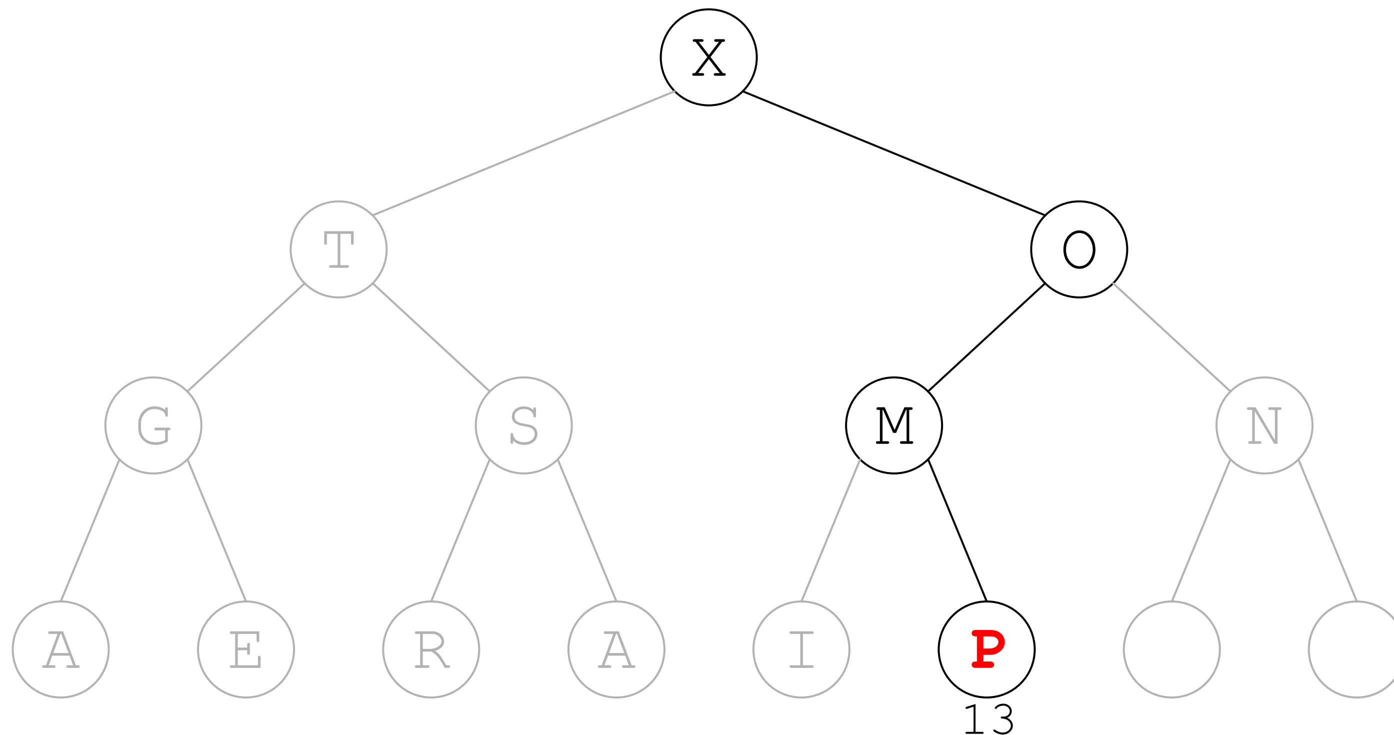
親ノードへのアクセス k/2 (商 小数点以下切捨て)

k の子ノードは k*2, k*2+1

Heap のデータ構造

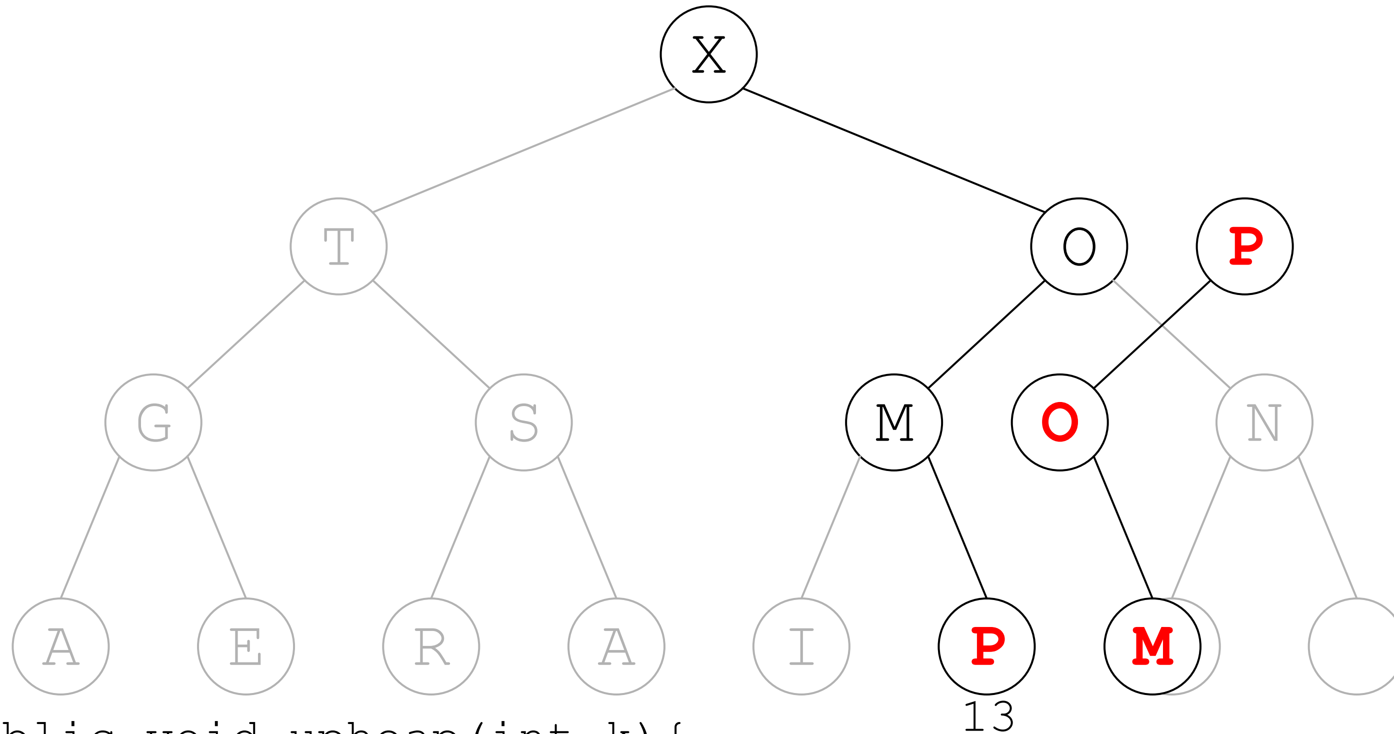

```
public class Heap {
    private int N, size;      // 1 <= N <= size
    private int[] a;

    public Heap(int k){
        size = k;        // heap の大きさ
        a = new int[size+1];
        N = 0;
        // N = 0 は空 N+1の場所
        // (1,2,3,...) に新しい元を入れる
    }
```

ヒープへの挿入



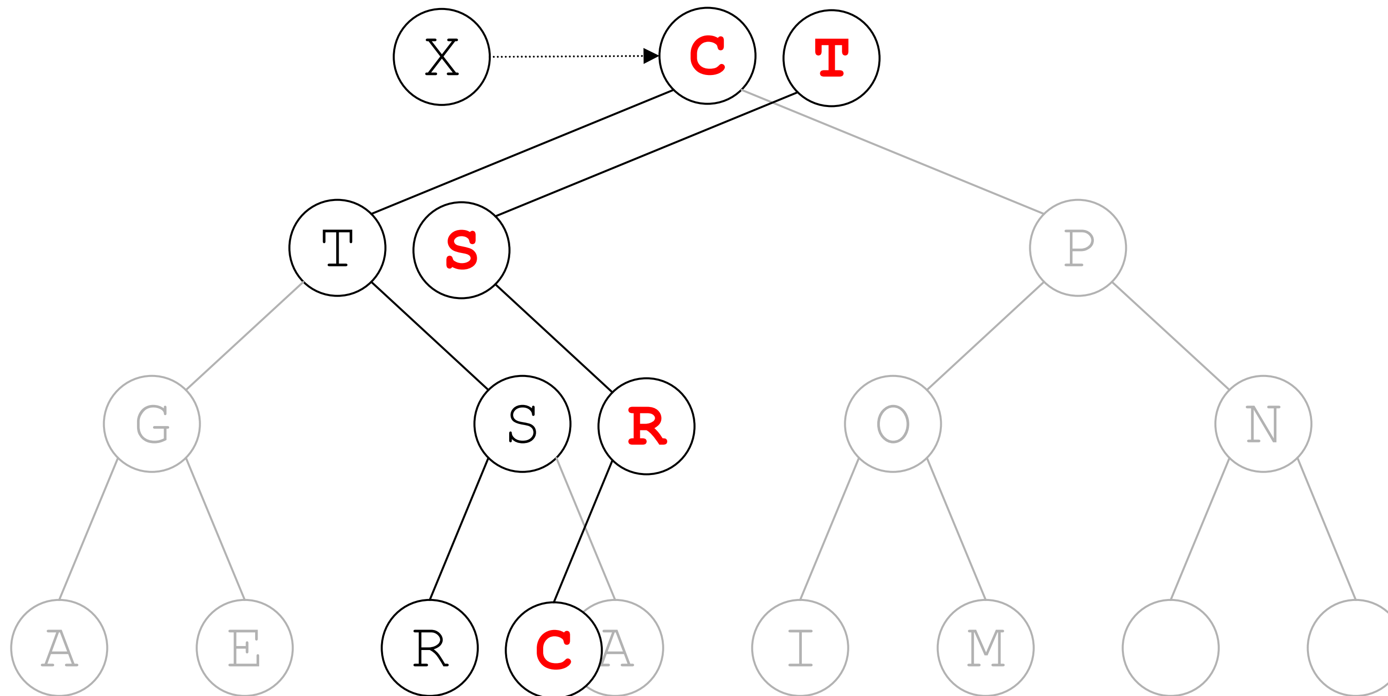

```
public void insert(int v){
        if(N < size){          // 入れる余裕があるか?
                a[++N] = v;    // N を増やして空の場所をみつけて挿入
                upheap(N);
        }
}
```

ヒープへの挿入




```
public void upheap(int k){
        int v = a[k];
        int i;
        for(i = k; 1 <= i/2 && a[i/2] < v; i = i/2){
                                        // 昇順にするには ">" に変更
                a[i] = a[i/2];
        }
        a[i] = v;
}
```

downheap



downheap の実装

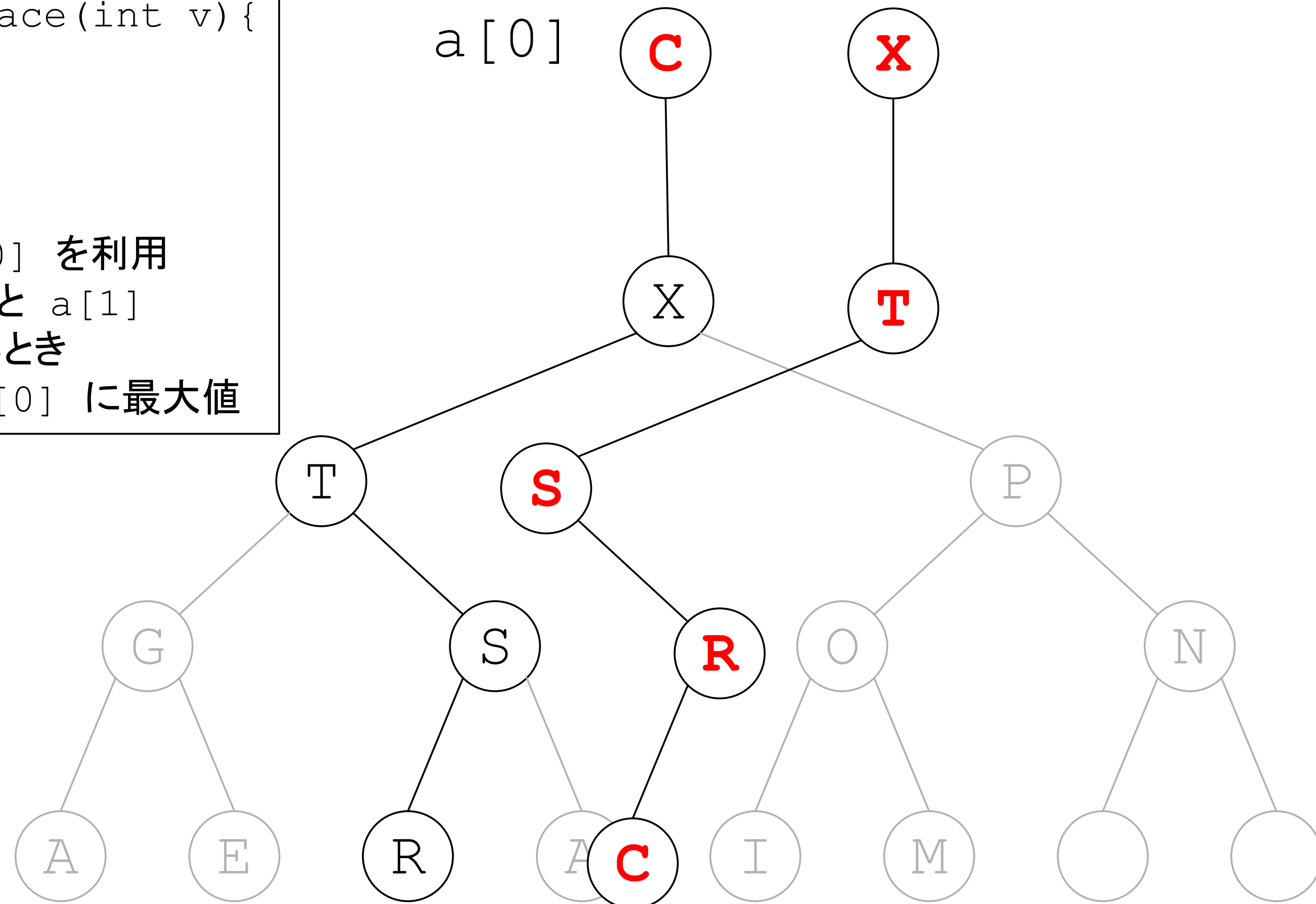

```
public void downheap(int k){
        int v = a[k];
        int i = k;
        while(i <= N/2){        // i には少なくとも左の子は存在
                int j = i*2;    // j は i の左の子

                if(j < N && a[j] < a[j+1]){
                                // 昇順にするには a[j] > a[j+1]
                                // i の右の子 j+1 の方が左の子 j より大きい
                        j++;    // 大きい方の子が j になるように変更
                }
                if(v >= a[j]){ // v の downheap は終了  昇順にするには “<=“
                        break;
                }else{
                        a[i] = a[j];// 子 j を親にあげ、子 j の下へすすむ
                        i = j;
                }
        }
        a[i] = v;       // v の落ち着き先 i を見つけたので置き換える
}
```

置き換え

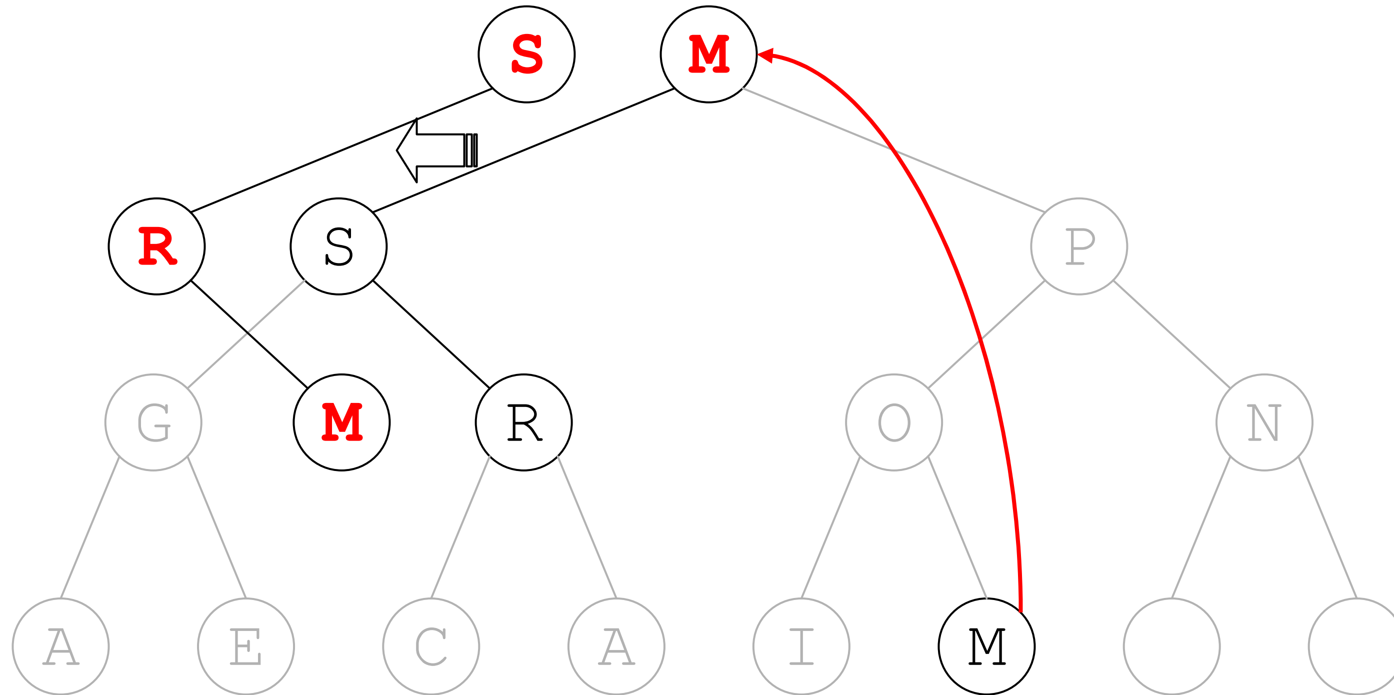
最大値を、それを超えない
新しい値 v で置き換える操作

```
public void replace(int v){
  a[0] = v;
  downheap(0);
}
```

値の入っていない a[0] を利用
a[0] の子は a[0] と a[1]
v が最大値を超えないとき
downheap(0) は a[0] に最大値



最大値の削除




```
public int remove(){
      int v = a[1];
      a[1] = a[N--];
      downheap(1);
      return v;
}
```

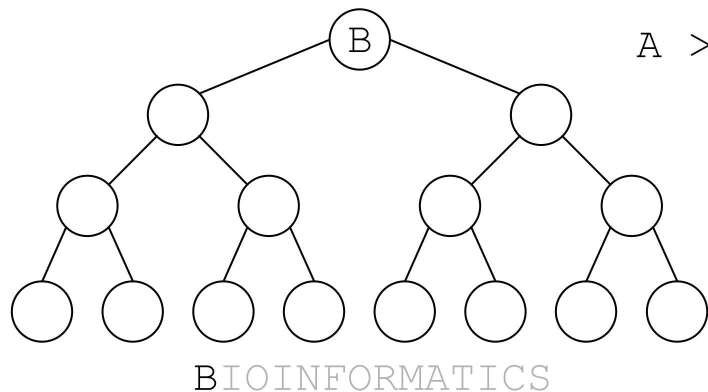
ヒープへの操作の性能

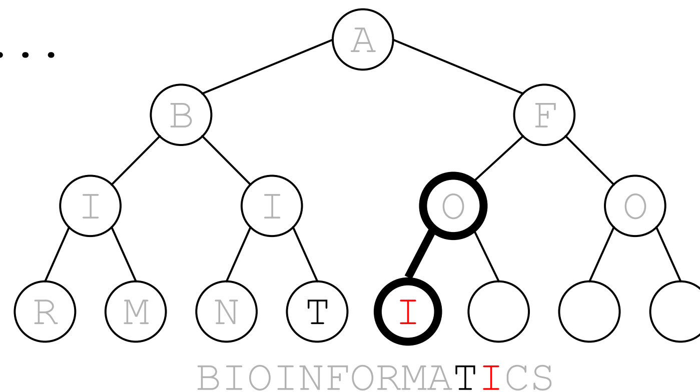
insert, remove, replace は N 要素のヒープに対して (2 lg N) 回より少ない比較操作で実行可能

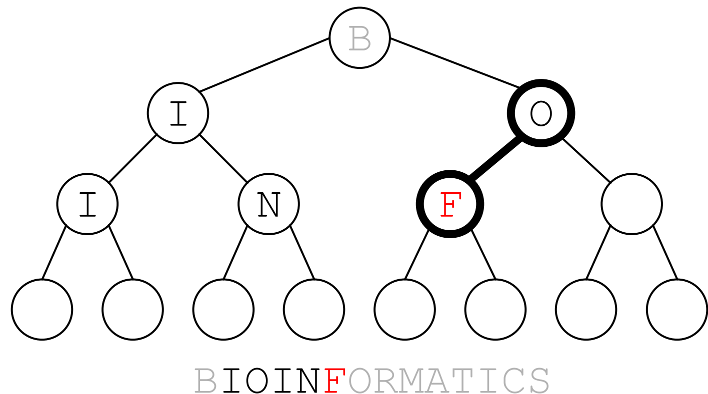
ヒープの高さは lg N
downheap での比較回数は高々この2倍
(2つの子との比較)

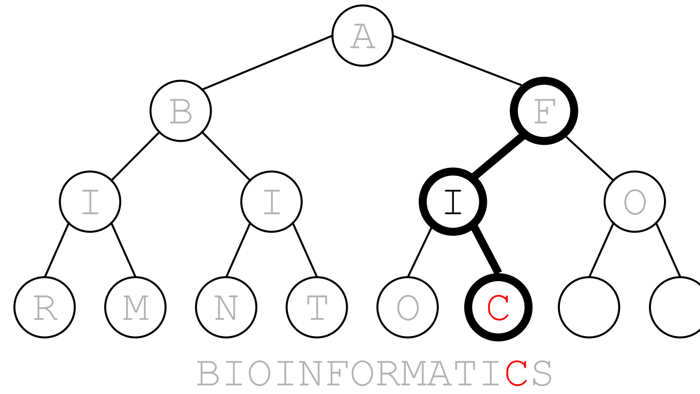
ヒープソート　　空のヒープにＮ個の要素を挿入

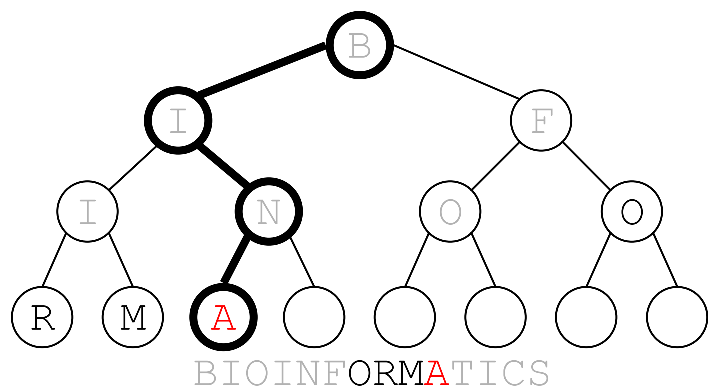
A > B > C > ...

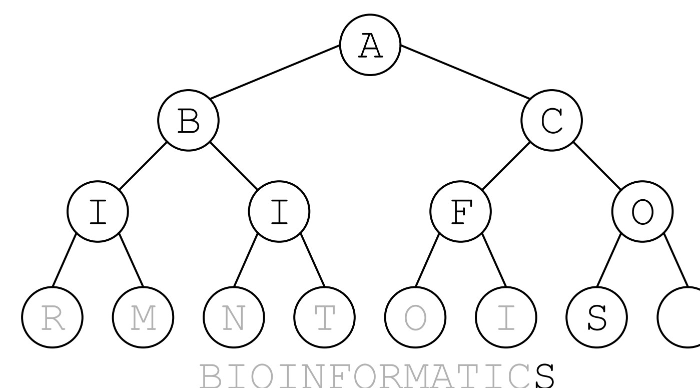




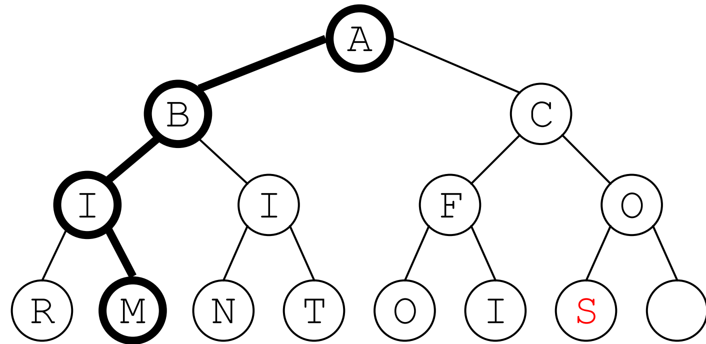


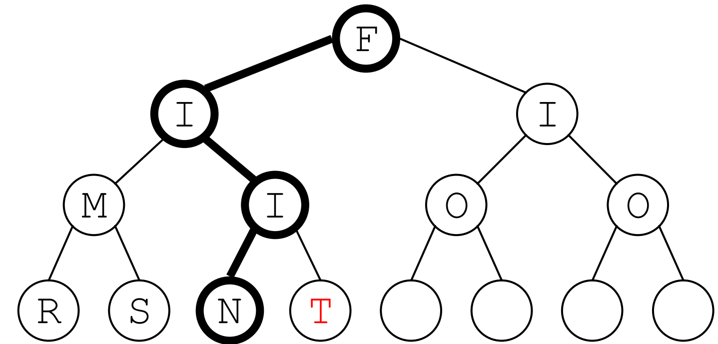


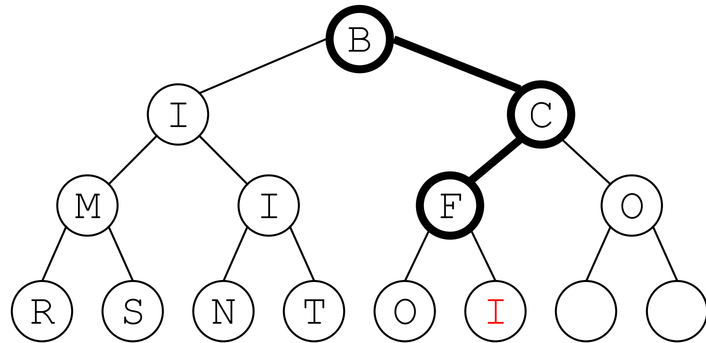


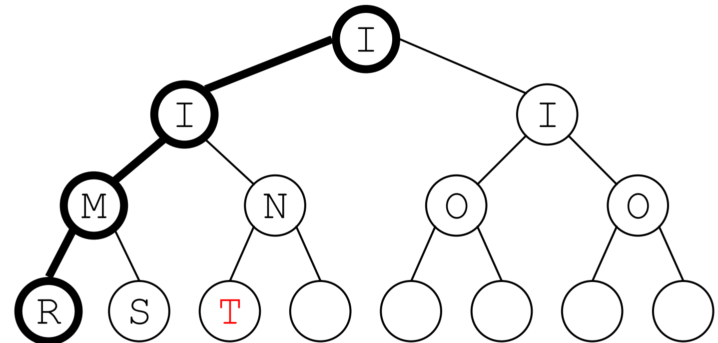


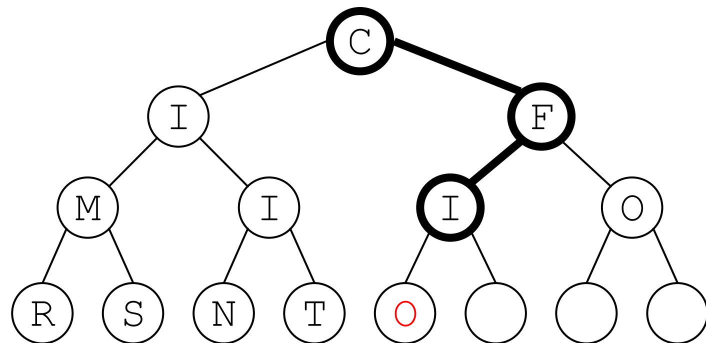
ヒープソート　最大値の削除を N 回くりかえす

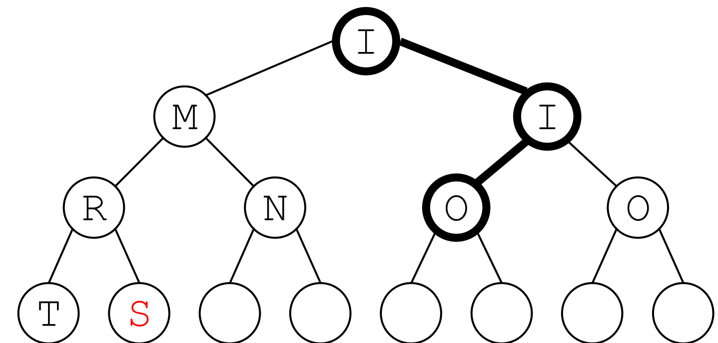






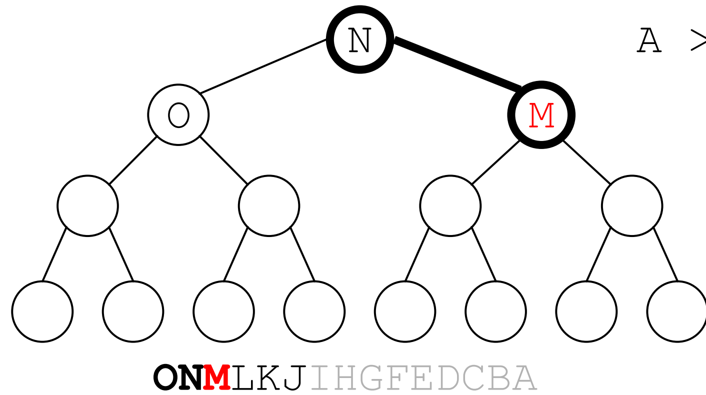


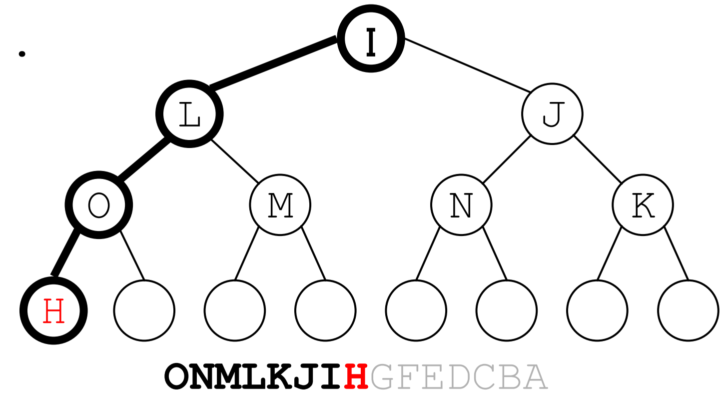


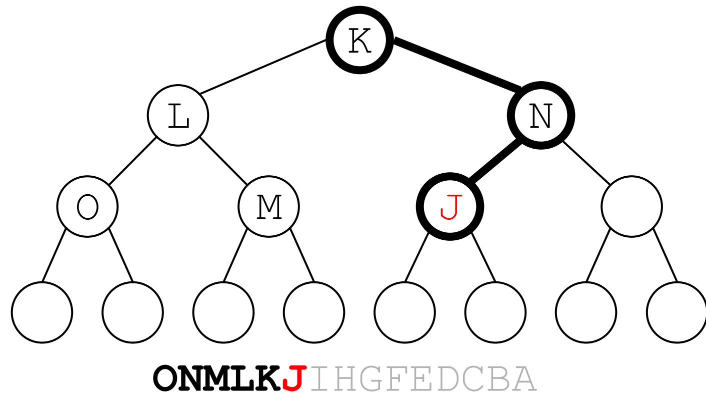


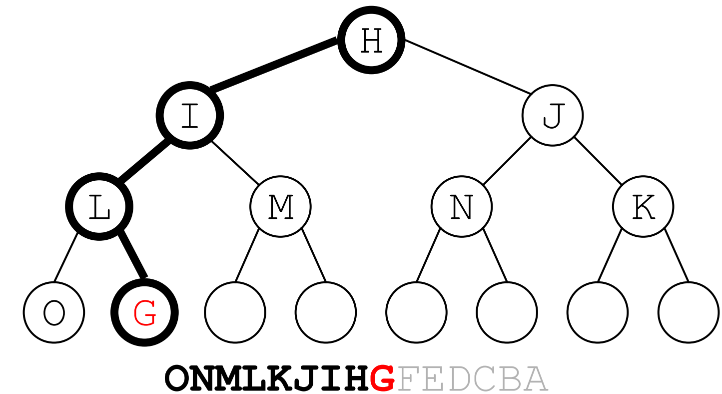
ヒープ作成で最も upheap が発生する場合

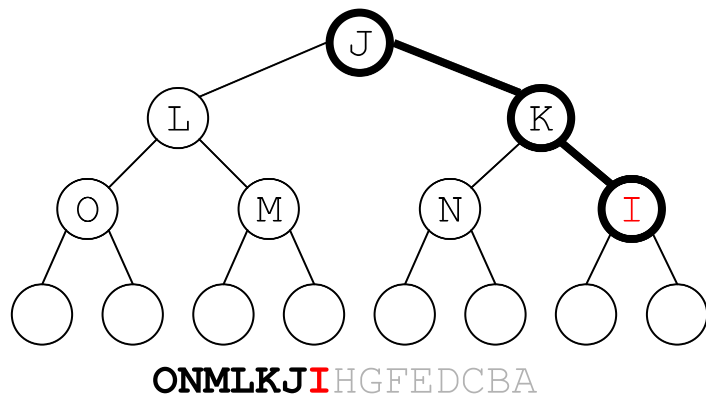
A > B > C > ...

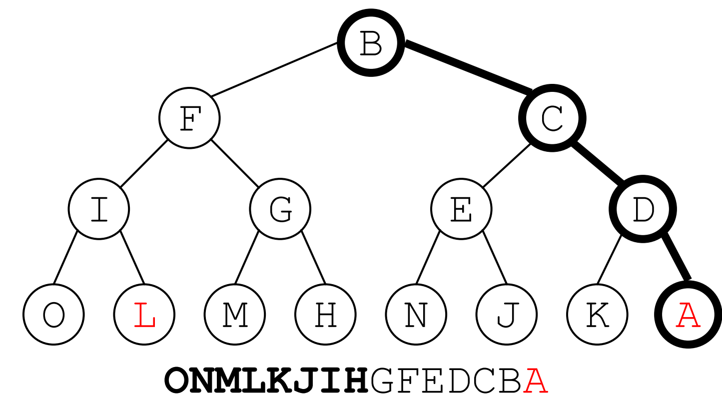












ヒープ作成で最も upheap が発生する場合の計算コスト

$$N = 2^d$$

$$C(N) = (d-1)2^{d-1} + (d-2)2^{d-2} + \ldots + 3\cdot 2^3 + 2\cdot 2^2 + 1\cdot 2^1$$

$$C(N) = 2C(N) - C(N)$$

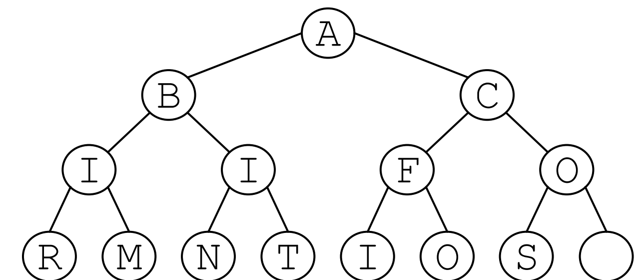
$$= (d-1)2^d - (2^{d-1} + 2^{d-2} + \ldots + 2^3 + 2^2 + 2^1)$$

$$= (d-1)2^d - (2^d - 2)$$

$$= (d-2)2^d + 2$$

$$\cong N \lg N$$

ボトムアップによるヒープの生成



1．　入力を完全2分木の上から下へ、左から右へ格納
2．　内部ノードを右から左、下から上に巡り
　　　downheapを実行
3．　この巡回により、内部ノード以下の部分木は
　　　常にヒープ（downheap　の安全な適用が可能）

ボトムアップによるヒープソートの計算コスト

ボトムアップによるヒープの生成は最悪でも要素数 N の線形時間
upheap を使う方法より減らせた！

$$N = 2^4 - 1 \quad C(N) = 3\cdot 2^0 + 2\cdot 2^1 + 1\cdot 2^2 + 0\cdot 2^3$$

$$N = 2^n - 1 \quad C(N) = \sum_{d=0}^{n-1} (n-1-d)2^d$$

$$C(N) = 2C(N) - C(N) = -(n-1) + 2^1 + \ldots + 2^{n-1} = 2^n - n - 1 < N$$

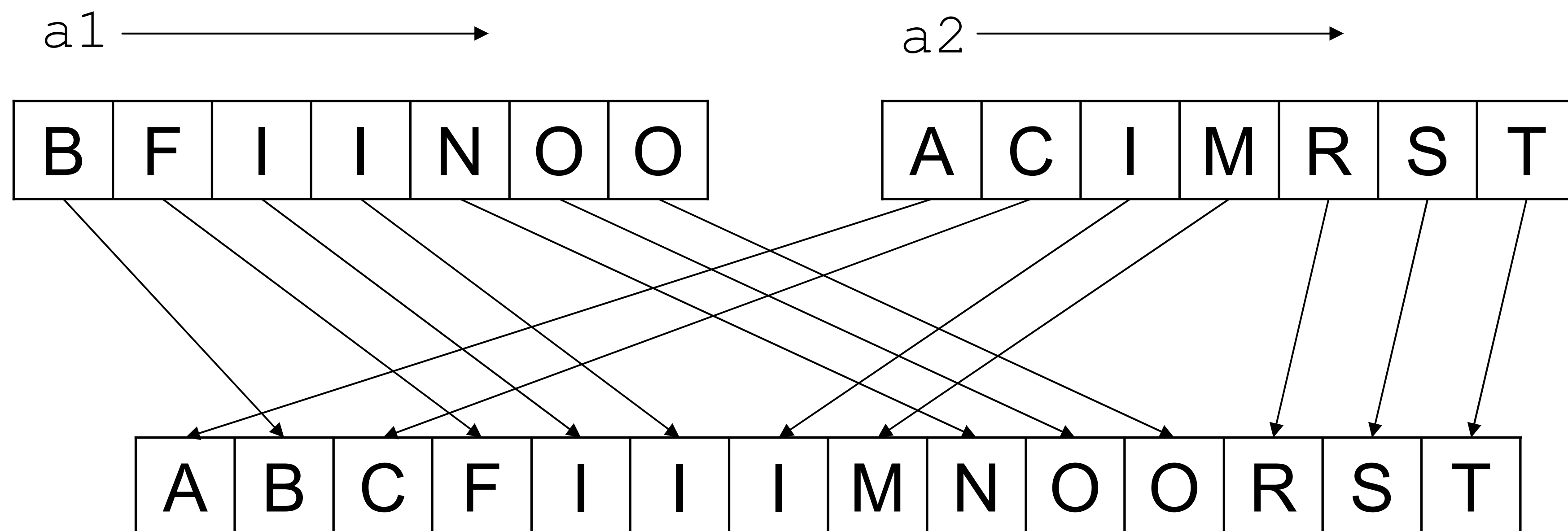
ヒープソート全体で、 N 個の要素の整列に、
最悪でも 2N lg N 回以下の比較しか実行しない

selection, insertion, bubble, quick sort
はどれも最悪の場合 N^2 に比例する計算時間がかかる

12章　Merge Sort

着想　整列済み配列の併合

- 配列 a1, a2 はソート済みであることを仮定
- 残っている先頭の2つの値で小さいほうを配列 b に代入



- 配列ではなく、リンクをつかった実装方法も容易

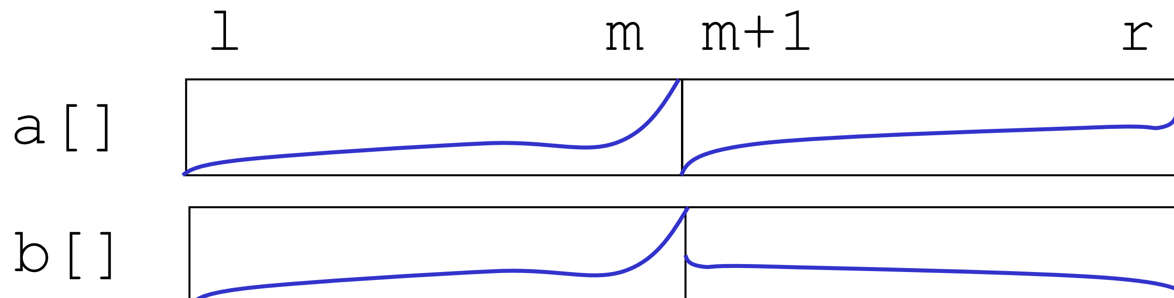
マージソート

BIOINFORMATICS
BIIONFORMATICS
BIIONFORMATICS
BIIOFNORMATICS
BIIOFNORMATICS
BIIOFNORMATICS
BFIINOORMATICS

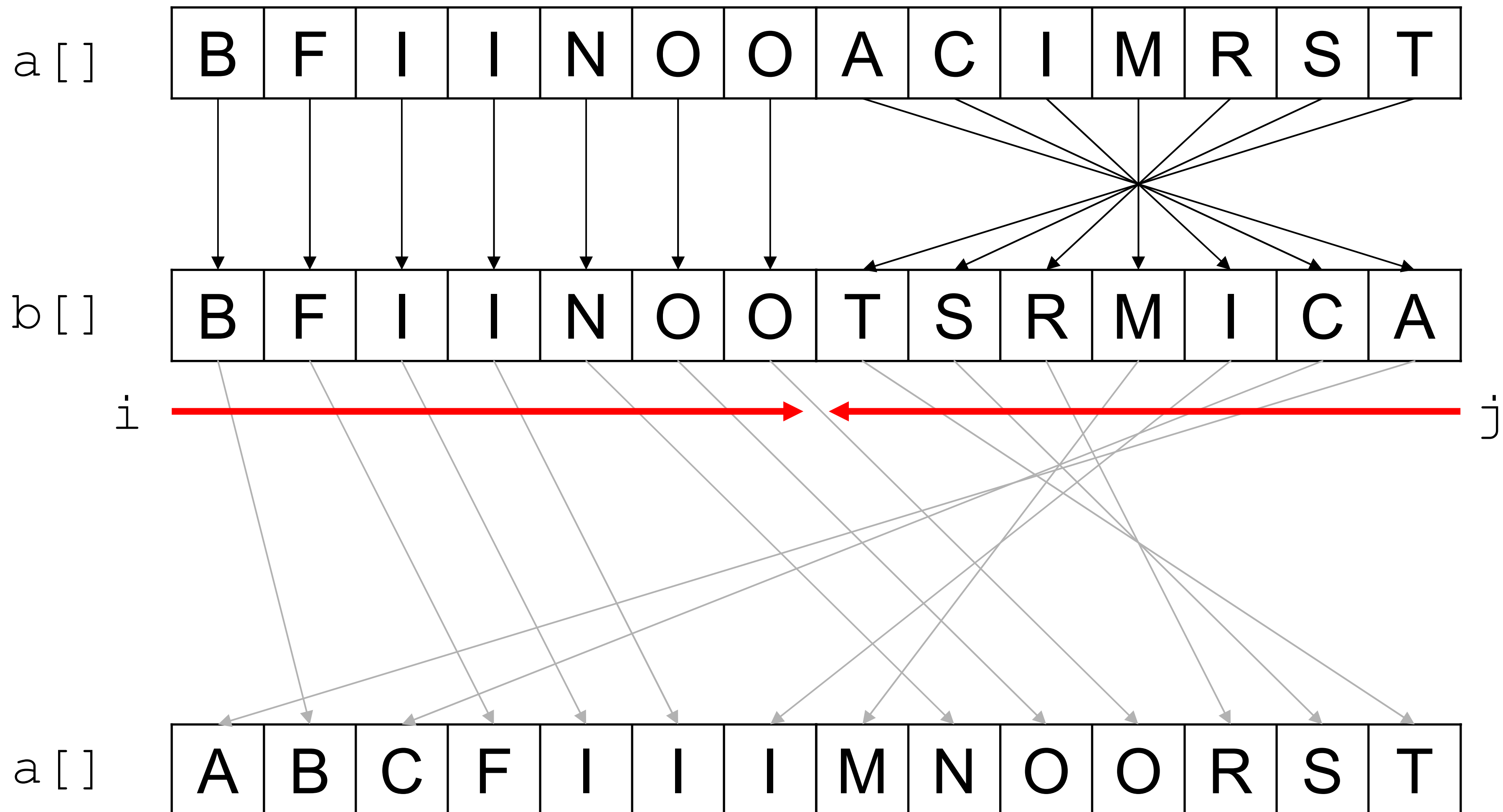
BFIINOOMRATICS
BFIINOOMRATICS
BFIINOOAMRTICS
BFIINOOAMRTCIS
BFIINOOAMRTCIS
BFIINOOAMRTCIS
BFIINOOACIMRST
ABCFIIIMNOORST

マージソートの実装


```
public void merge(int l, int r){
    if(l < r){
        int m = (r+l)/2;
        merge(l, m);                    // 左半分を merge sort
        merge(m+1, r);                  // 右半分を merge sort
        // 左半分と右半分を背中合わせにコピー
        for(int i = m+1; l < i; i--){        // 左半分はそのまま a から b へ
            b[i-1] = a[i-1];         }
        for(int j = m; j < r; j++){          // 右半分は逆順にコピー
            b[r+m-j] = a[j+1]; }
        // ソート
        int i = l;                      // 左端から
        int j = r;                      // 右端から
        for(int k = l; k <= r; k++){
            if(b[i] < b[j]){
                a[k] = b[i++];
            }else{
                a[k] = b[j--];
            }     // 右半分もしくは左半分を超えても正常に動作
        }
    }
}
```



動作例



ボトムアップ型マージソート


```
BIOINFORMATICS
BIIONFORMATICS
BIIOFNORMATICS
BIIOFNORMATICS
BIIOFNORAMTICS
BIIOFNORAMITCS
BIIOFNORAMITCS
```

```
BIIOFNORAMITCS
BIIOFNORAMITCS
BIIOFNORAIMTCS
BIIOFNORAIMTCS
BFIINOORAIMTCS
BFIINOORACIMST
ABCFIIIMNOORST
```

マージソートの性能

マージソートは N 個の要素を<u>最悪</u> N lg N 回の比較で整列

ボトムアップ型では約 N 回の比較が lg N 回行われる

トップダウン型では N 個の要素の比較回数を M_N とすれば
分割統治の漸化式　$M_N=2M_{N/2}+N$ より M_N は約 N lg N

16章　ハッシュ 法

ハッシュ関数

レコードを識別するキーに算術演算 （ハッシュ関数） をほどこし
表（ハッシュ表）のアドレスに変換する

ハッシュ関数の利点
異なるアドレスに変換された ２つのキーは異なる

衝突処理
同じアドレスに変換された ２つのキーが異なる可能性が残る

ハッシュ表を大きく取ることで衝突処理を回避
記憶領域と衝突頻度のトレードオフ

ハッシュ関数

もっとも伝統的な方法

$$h(k) = k \bmod M$$

M はハッシュ表のサイズで大きな素数であること

i 番目のアルファベットは、i を 5 ビットのコードで符号化

A	00001	K	01011	Y	11001
E	00101	L	01100	Z	11010

M=101　　キー "AKEY"

0000101011001011001 = 44217 (10進数)

44217 mod M = 80

M=32 だと 44217 mod M = 25

最後の文字の数になる　　"-----Y" の形は 25

ハッシュ関数の高速計算　mod の性質とホーナー法

整数へ変換した値		下位5ビットの整数表現	
V	86	22	$22*32^{10}+$
E	69	5	$5*32^{9}+$
R	82	18	$18*32^{8}+$
Y	89	25	$25*32^{7}+$
L	76	12	$12*32^{6}+$
O	79	15	$15*32^{5}+$
N	78	14	$14*32^{4}+$
G	71	7	$7*32^{3}+$
K	75	11	$11*32^{2}+$
E	69	5	$5*32^{1}+$
Y	89	25	25 mod M

大きな数の生成を避けたい

(((22*32+5)*32+18)*32+25).... mod M

=(((22*32+5 mod M)*32+18 mod M)*32+25 mod M)...

mod を加算、乗算前に実行しても同じ

ハッシュ関数の実装


```
public int hashFun(String s, int size){
      int answer = 0;
      for(int i = 0; i < s.length(); i++){
            answer =
            (answer*32+((int)s.charAt(i))%32)%size;
      }
            // ホーナー法で計算
            // 大文字アルファベットを 1-26 へ写像するため
            //  32 の余りを使っている
      return answer;
}
```

V E R Y L O N G K E Y

86 69 82 ...

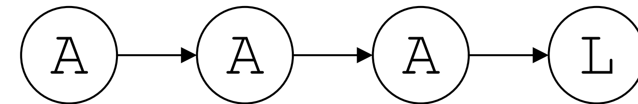
22 5 18 ...

衝突処理　分離連鎖法

key　i　hash(i mod M)　M=11

key	i	hash
A	1	1
S	19	8
E	5	5
A	1	1
R	18	7
C	3	3
H	8	8
I	9	9
N	14	3
G	7	7
E	5	5
X	24	2
A	1	1
M	13	2
P	16	5
L	12	1
E	5	5

hash	
0	
1	A A A L
2	M X
3	N C
4	
5	E E E P
6	
7	G R
8	H S
9	I
10	



各 hash に key が
いくつ挿入されるか
わからない場合

各 key をリストで繋ぐ

衝突処理　線形探索法

開番地法(open addressing)

N 個のレコードを M(>N)個の hash 表に格納し、
衝突処理に M-N 個の空き領域を利用

線形探索法 (linear probing)

hash した値が、異なるキーで使われている場合、
空いた場所が見つかるまで、ハッシュ表を順番に線形に探索

衝突処理　線形探索法

key	i	hash(i mod M)
A	1	1
S	19	0
E	5	5
A	1	1
R	18	18
C	3	3
H	8	8
I	9	9
N	14	14
G	7	7
E	5	5
X	24	5
A	1	1
M	13	13
P	16	16
L	12	12
E	5	5

N=17
M=19

0	S	1
1	A	0
2	A	3
3	C	5
4	A	12
5	E	2
6	E	10
7	G	9
8	H	6
9	I	7
10	X	11
11	E	16
12	L	15
13	M	13
14	N	8
15		
16	P	14
17		
18	R	4

hash が衝突した場合、
空いた場所まで移動する

T を探す場合
i=20
i mod 19=1

hash値 1〜14
まで探索して
無いことを確認

線形探索法の実装


```
public class Hash {
    private int M;          // ハッシュ用素数
    private int N;          // 入力ストリングの長さ
    private int L;          // ハッシュに使うストリングの長さ
    private String[] a;     // ハッシュ表  長さ L の部分列
    private int[] p;        // ハッシュ表  部分列が出現する位置
    public Hash(int size, int width, String s){
        M = size;
        N = s.length();
        L = width;
        a = new String[M];
        p = new int[M];
        for(int i = 0; i < M; i++){         // a, pos を空で初期化
            a[i] = "";
            p[i] = -1;
        }
        int i;
        for(i = 0; i+L-1 < N; i++){
            String t = s.substring(i,i+L); // i 番目から長さL の部分列を切り出す
            int k = hashFun(t, M);         // 切り出した配列が格納できる場所を探す
            while(0 < a[k].length()){      // a[k] が空でないならば先に進む
                k = (k+1) % M;
            }
            a[k] = t;
            p[k] = i;
        }
    } …
```

線形探索法で目的の文字をサーチ


```
public void search(String t){
     int k = hashFun(t, M);

     while(0 < a[k].length()){ // 空になるまで探す
          if(t.equals(a[k])){  // 同一か否か？
               System.out.print(" " + p[k]);
          }
          k = (k+1) % M;
     }
}
```

２重ハッシュ法

線形探索法の探索スピードを向上

hash が衝突した場合、空いた場所まで 1 つずつ移動するのではなく u 個おきに移動する

u と M は互いに素であるように選択して、最終的にはすべての空いた場所を巡ることができるように配慮

実際は u = 8-(k mod 8) のような簡単な関数で十分

２重ハッシュ法

key	i	hash1	hash2
A	1	1	7
S	19	0	5
E	5	5	3
A	1	1	7
R	18	18	6
C	3	3	5
H	8	8	8
I	9	9	7
N	14	14	2
G	7	7	1
E	5	5	3
X	24	5	8
A	1	1	7
M	13	13	3
P	16	16	8
L	12	12	4
E	5	5	3

hash1
k mod 19

hash2
8-(k mod 8)

0	S	
1	A	A
2		P
3	C	
4		
5	E	
6		M
7	G	
8		A
9	I	
10		
11		E
12	L	
13		X
14	N	
15		A
16		H
17		E
18	R	

索引

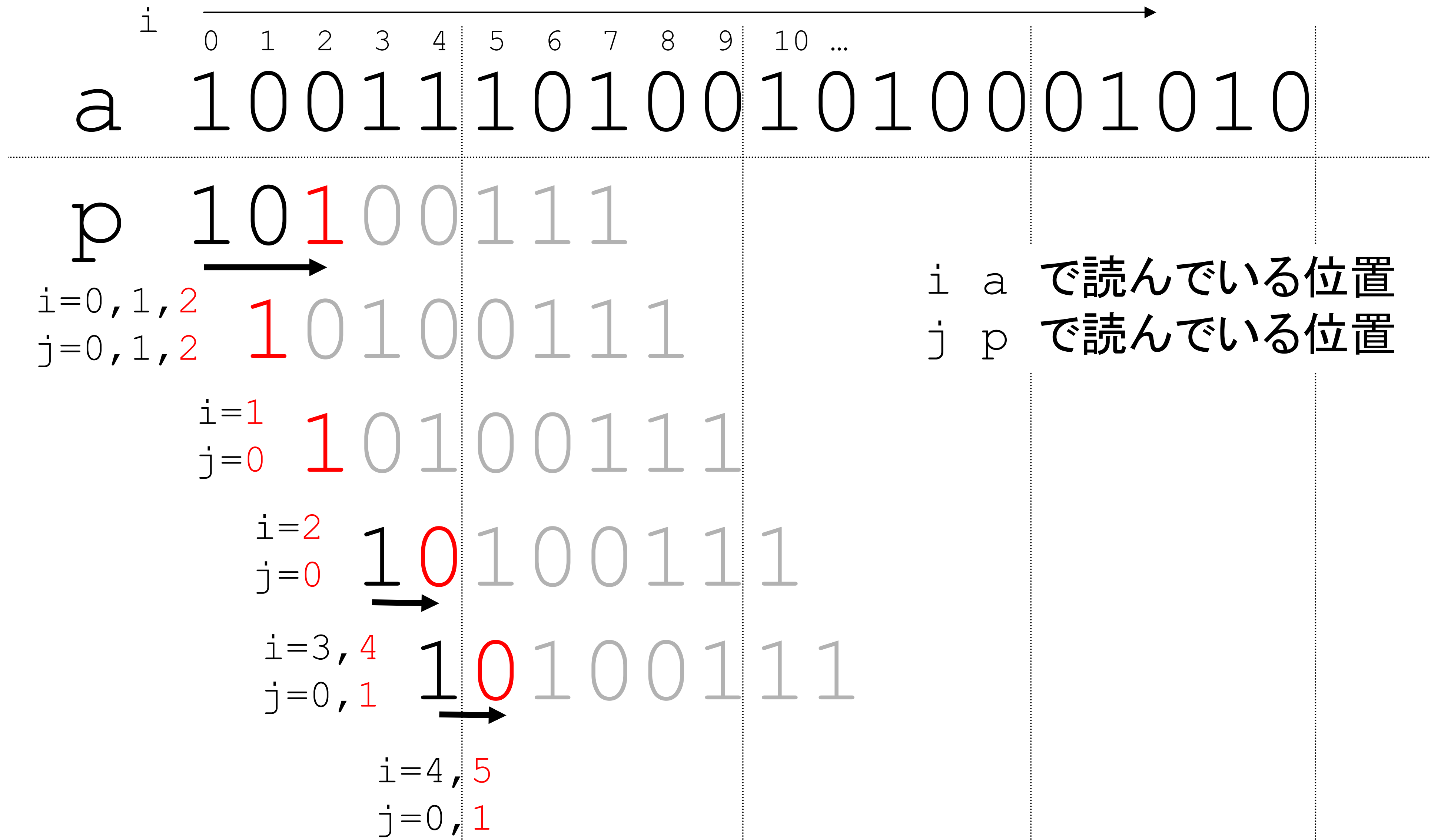
0 5 10 15 20
ATGCGAACTATGCCTCTCTATGACGCT

0	CT	7	10	TG	10	20		
1	CT	13	11	TG	18	21		
2	CT	15	12	CC	12	22	GA	4
3	TA	8	13	GC	22	23	AT	0
4	AA	5	14	CT	23	24	GC	2
5	TC	14	15			25	AT	9
6	AC	6	16	CG	3	26	GC	11
7	TA	16	17	CG	21	27	AT	17
8	AC	20	18			28	GA	19
9	TG	1	19					

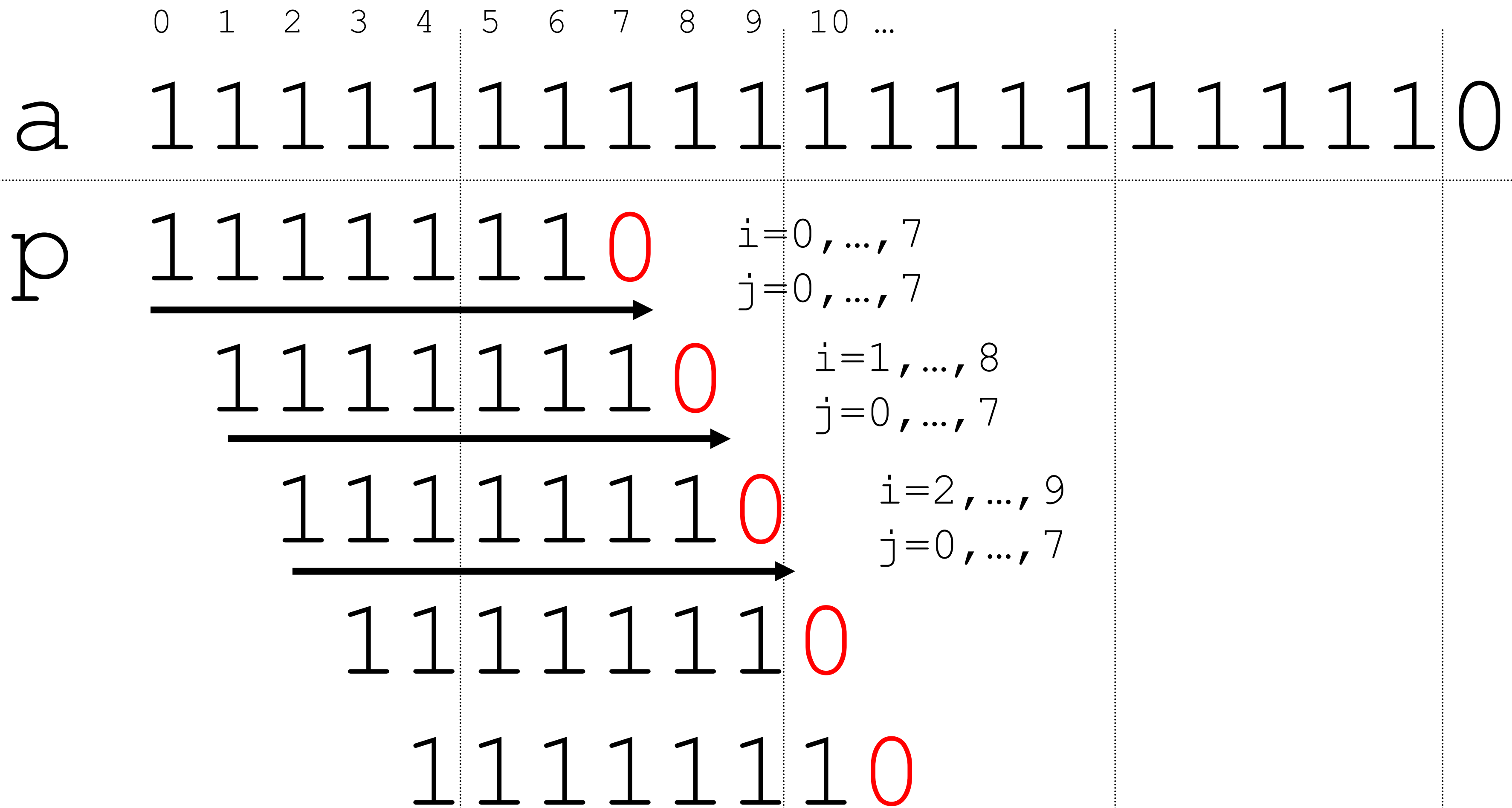
```
AT  A=1, T=20  1*32+20 mod 29 = 23
CT  C=3, T=20  3*32+20 mod 29 = 0
GA  G=7, A=1   7*32+1  mod 29 = 22
```

19章　文字列探索（完全マッチ）

brute-force algorithm



brute-force algorithm



最悪約 MN 回比較

brute-force algorithm


```
public static int bruteSearch(String a, String p) {
    int N = a.length();
    int M = p.length();
    if(M > N){              // パターン p の方が長いので完全にマッチするはずない
        return -1;
    }
    int i, j;
    for (i = 0, j = 0; i < N && j < M; ) {
        if (a.charAt(i) != p.charAt(j)) {
            i -= j - 1;     // マッチしなかったので前回の次の位置 (i-j)+1 から再開
            j = 0;          // パターンは、ふりだしに戻る
        } else {            // 1文字マッチしたら i, j ともに１つ進める
            i++;
            j++;
        }
    }
    if (j == M) {           // パターン p が最後まで完全にマッチした
        return i - M;       // パターン p のマッチが開始した位置を返す
    } else {
        return -1;          // p は完全にはマッチしなかった
    }
}
```

Knuth − Morris − Pratt

a 101010100111

p 10100111

再実行の場所は？

10100111 brute−force むだ

見落とし

10100111

10100111 むだなく 見落とし無く

Knuth – Morris – Pratt

青 一致 赤 不一致 灰 未走査

i

a 1010100111

p 10100111

j=4

10100111

j=2=next[4]

i は逆行させず固定
パターン p だけ を動かす
i に対応する j の値を
next[i]に入れる

a を見ないで p だけから計算

一致する接尾辞と接頭辞をさがす

文字列 $a_1, a_2, ..., a_{k-1}, a_k$
先頭a_1からはじまる連続した部分文字列 $a_1, ..., a_l\ (l \leqq k)$ を接頭辞（prefix）
最終文字a_kで終わる連続した部分文字列 $a_l, ..., a_k\ (1 \leqq l)$ を接尾辞（suffix）

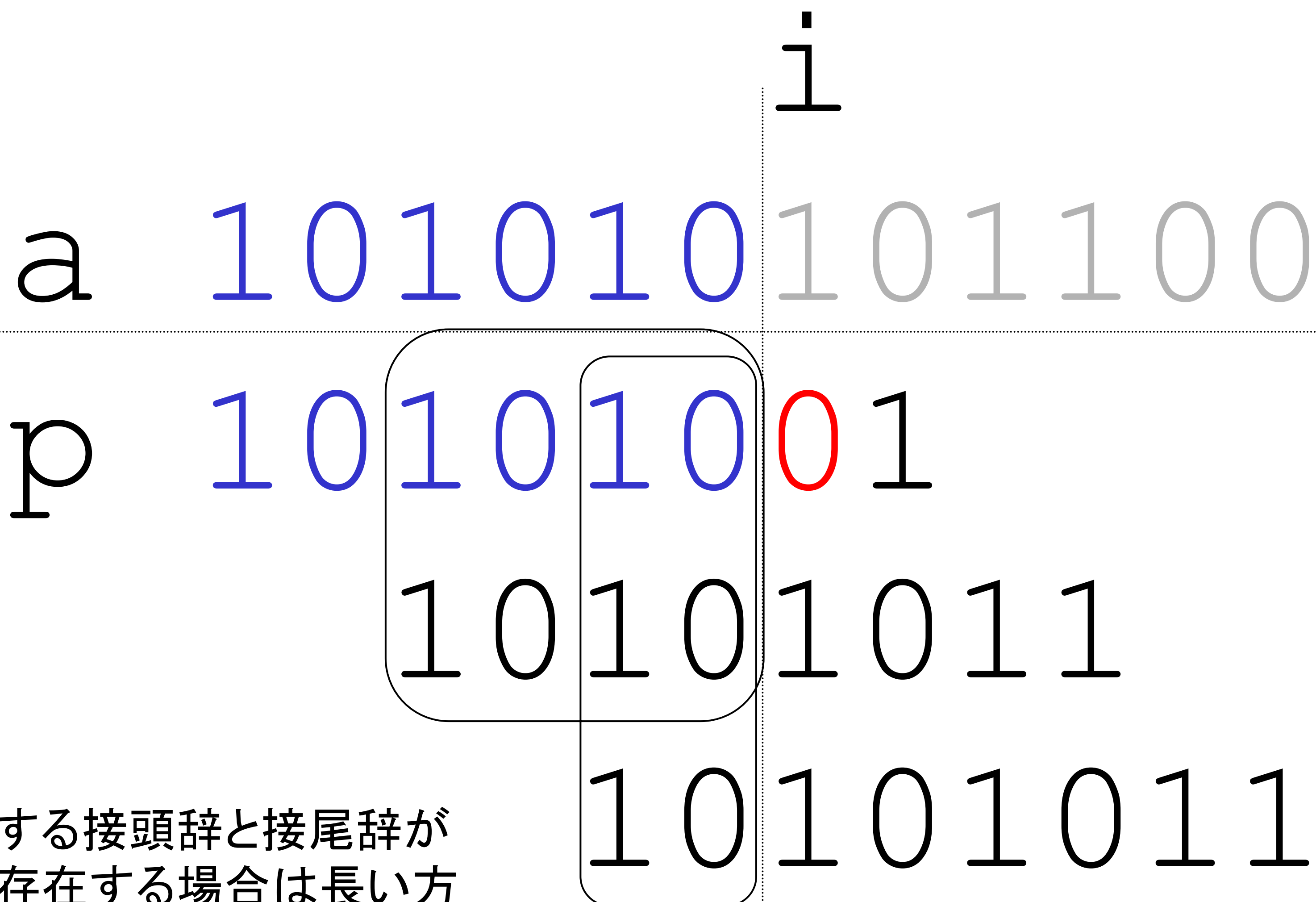
マッチした文字列　　接尾辞

1010
　　10100111

接頭辞　　パターン

Knuth-Morris-Pratt　のポイント
マッチした文字列より短い接尾辞と、パターンより短い接頭辞が一致するとき
その文字列がマッチするかどうかのチェックは省くことができる

一致する接尾辞と接頭辞をさがす



一致する接頭辞と接尾辞が複数存在する場合は長い方を選択

前のパターンとは異なるので注意

一致する接尾辞と接頭辞をさがす

i=2

1 0 1 0 0 1 1 1

10100111　next[2]=0

一致する接頭辞と接尾辞は空の文字列

i=3

1 0 1 0 0 1 1 1

10100111　next[3]=1

一致する接頭辞と接尾辞は1

一致する接尾辞と接頭辞をさがす


```
10100111
  10100111        next[4]=2
10100111
     10100111     next[5]=0
10100111
      10100111    next[6]=1
10100111
       10100111   next[7]=1
```

一致する接尾辞と接頭辞をさがす

	0	1	2	3	4	5	6	7
next	-1	0	0	1	2	0	1	1

```
    i=0
a   00100111
p   10100111
     10100111    next[0]=-1
```

```
     i=1
    11100111
p   10100111
     10100111    next[1]=0
```

一致する接頭辞と接尾辞は空の文字列なので便宜的にこのように値を設定

next[] の計算


```
                i    j
10100111        0    -1→next[0]
 10100111
10100111        1    0→next[1]
 10100111
   10100111          -1←next[0]
```

マッチしないので、マッチする文字が出るまでパターンをずらす

この場合は見つからず j=-1 で終了し、iとjを1増やす

```
10100111        2    0→next[2]
  10100111
```

マッチするので i,j を1増やす

next[] の計算


```
                  i   j
1010 0111         3   1→next[3]
  10100111

マッチするので i,j を1増やす

10100 111         4   2→next[4]
  10100111
    10100-            0←next[2]
      10100-          -1←next[0]
```

マッチしない場合、マッチする文字が出るまでパターンをずらす

この場合は見つからず j=-1 となり終了し、iとjを1増やす

next[] の計算


```
                        i      j
10100111                5      0→next[5]
     1010-
```

マッチするので i, j を1増やす

```
10100111                6      1→next[6]
     1010
      1010-                    0←next[1]
```

マッチしないので、マッチする文字が出るまでパターンをずらす
j=0 でマッチし、 i, j を1増やす

```
10100111                7      1→next[7]
      1010-
```

Knuth – Morris – Pratt のアルゴリズム

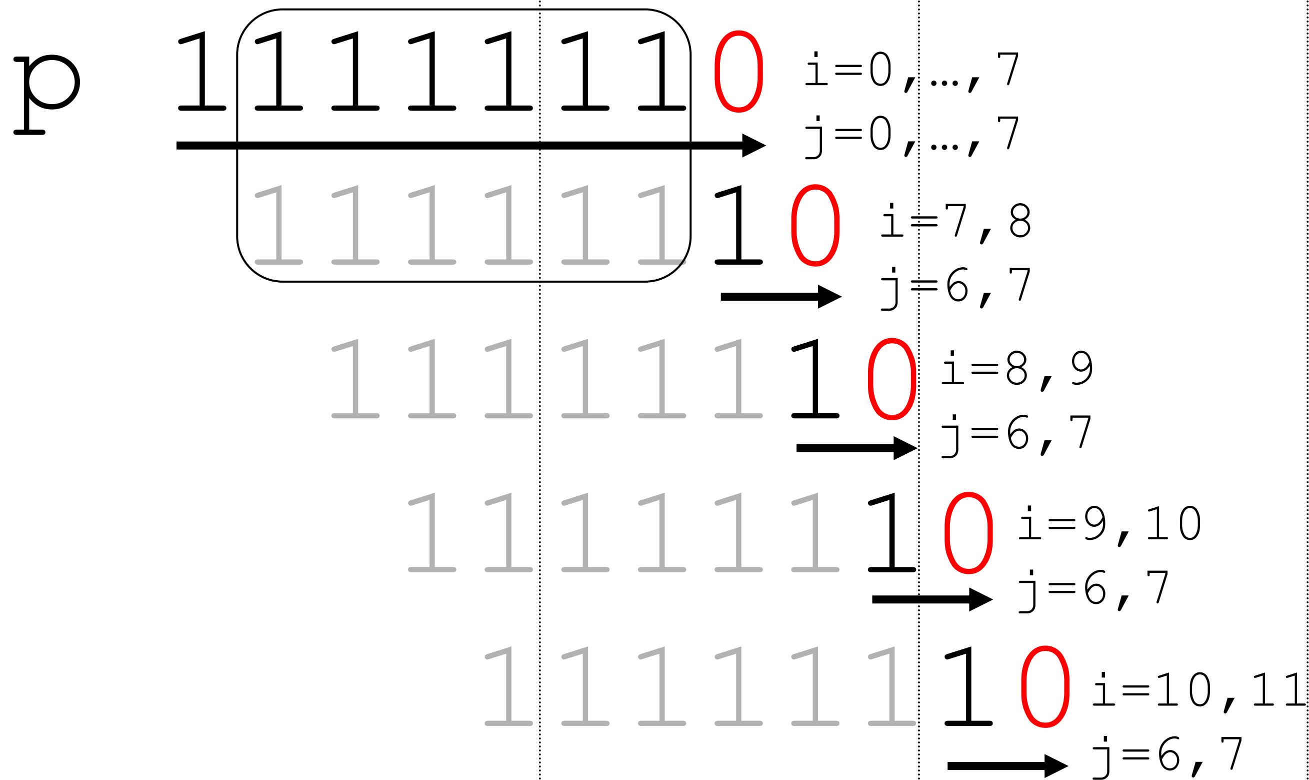

```
public static int kmpSearch(String a, String p){
        int N = a.length();
        int M = p.length();
        if(M > N){ return -1;}
        int i, j;
        // next を初期化
        int[] next = new int[M+1];
        next[0] = -1;
        for(i = 0, j = -1; i < M; i++, j++, next[i]=j){
                while( (0 <= j) && (p.charAt(i) != p.charAt(j)) ){
                        j = next[j];
                }
        }
        // マッチするか否かを探索
        for(i = 0, j = 0; (i < N) && (j < M); i++, j++){
                while( (0 <= j) && (a.charAt(i) != p.charAt(j)) ){
                        j = next[j];
                }
        }
        if(j == M)
                return i - M;  // 完全マッチを見つけたら先頭の位置を返す
        else
                return -1;
}
```

Knuth – Morris – Pratt


```
100111010010100001010100111
10100111
  10100111
   10100111
    10100111
     10100111
          10100111
```

j	0	1	2	3	4	5	6	7
next	-1	0	0	1	2	0	1	1

Knuth – Morris – Pratt での動作は？



Knuth - Morris - Pratt

性質 文字比較 (a.charAt(i)!=p.charAt(j))の実行回数は 2N 回以下

i++,j++ の実行回数(高々 N)と j=next[j] の実行回数の和

j++ は 1 ふえる
j=next[j]は 1 以上減り、j>=-1
j=next[j]はj++より実行回数が少ない

j++ は高々 N 回実行される. j=next[j] の実行回数は高々 N 回.

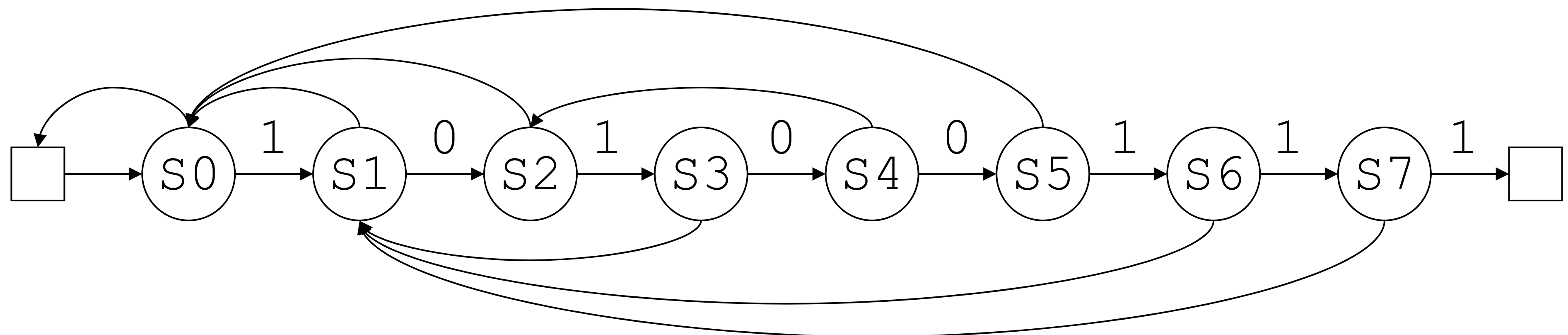
性質 initnext(p)の文字比較回数は 2M 回以下

証明は同様

Knuth – Morris – Pratt

	0	1	2	3	4	5	6	7
next	-1	0	0	1	2	0	1	1

有限状態機械として表現



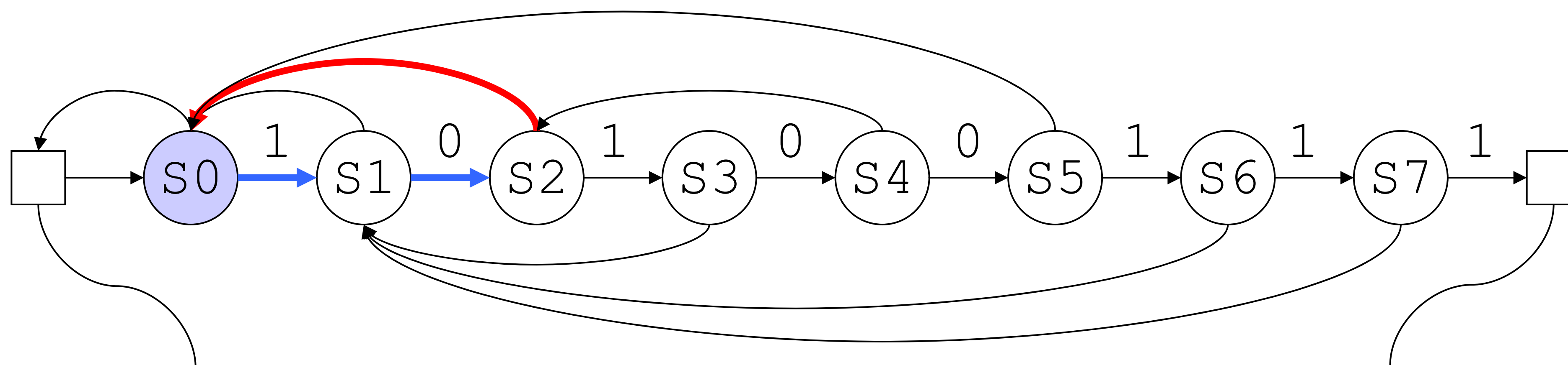
S3 は p[3] を a[i] と比較しようとしている状態
もどる辺が next[j] を表現

Knuth – Morris – Pratt

1001110100101000101001 11

10100111

10100111



初期状態
常に右側に遷移

停止状態
パターンを検出

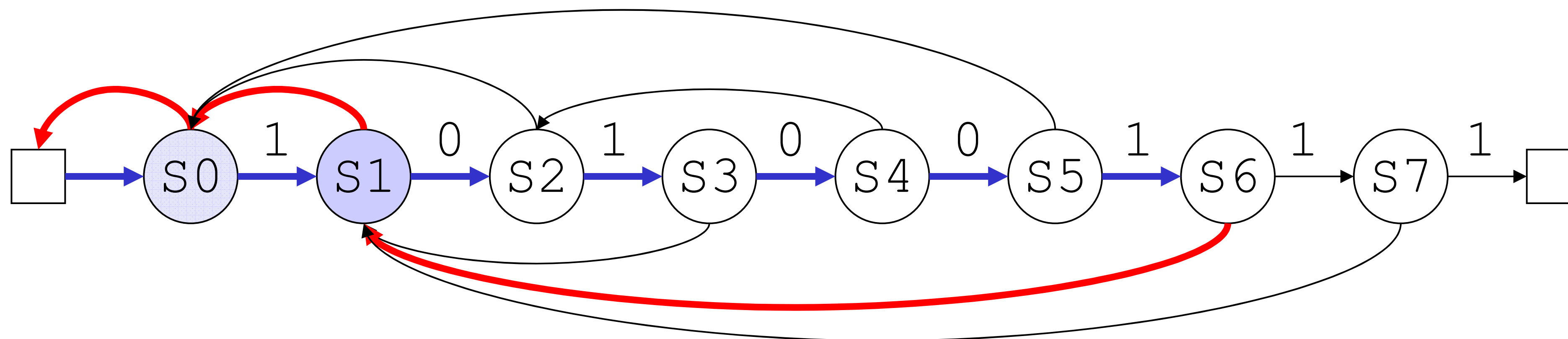
Knuth – Morris – Pratt

1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1 1 1

1 0 1 0 0 1 1 1

1 0 1 0 0 1 1 1

1 0 1 0 0 1 1 1

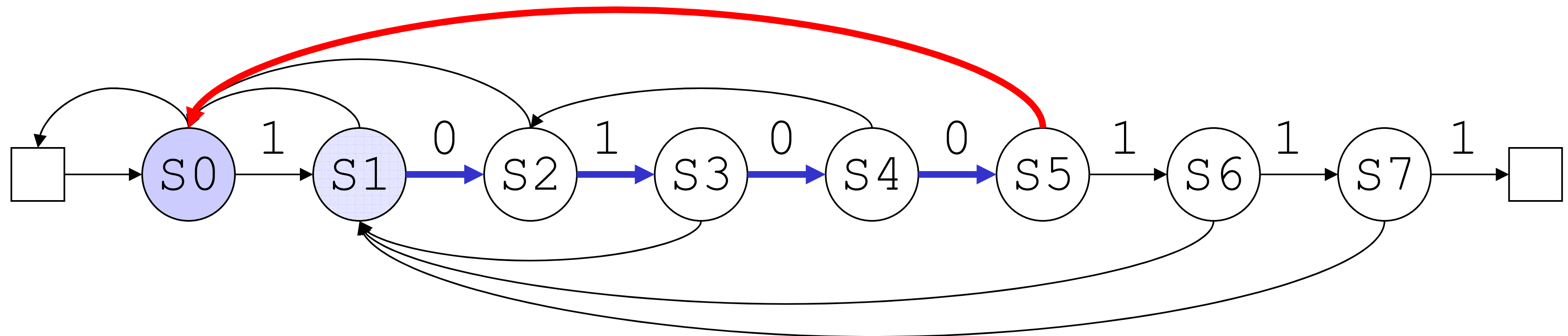


Knuth – Morris – Pratt

10011101001010001010 0111

1010 0111

10100111

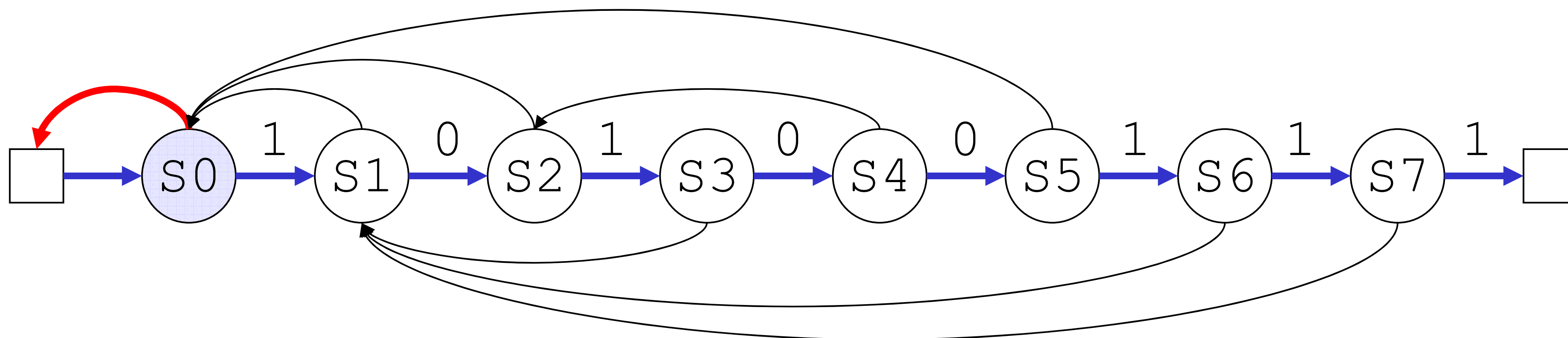


Knuth – Morris – Pratt

1001110100101000**10100111**

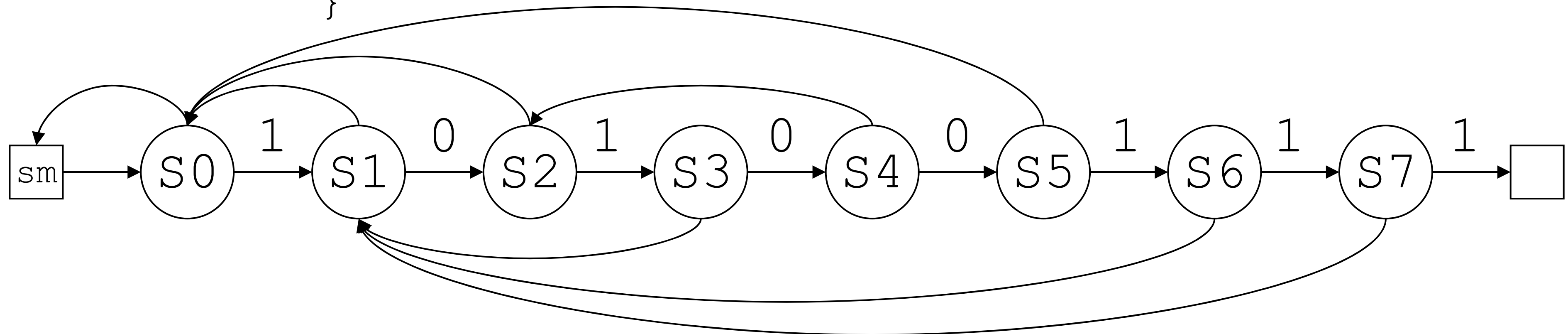
10100111

10100111



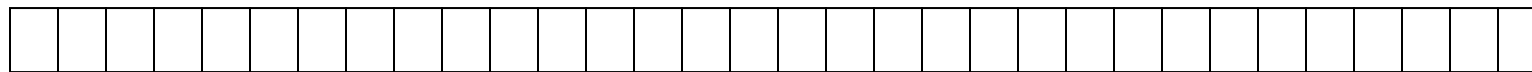
Knuth – Morris – Pratt

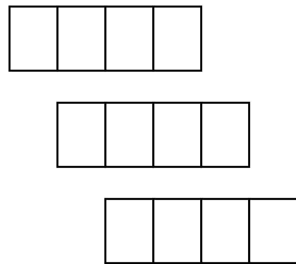

```
int kmpsearch(char *a){
   int i=-1;
sm:   i++;
s0:   if(a[i] != '1') goto sm; i++;
s1:   if(a[i] != '0') goto s0; i++;
s2:   if(a[i] != '1') goto s0; i++;
s3:   if(a[i] != '0') goto s1; i++;
s4:   if(a[i] != '0') goto s2; i++;
s5:   if(a[i] != '1') goto s0; i++;
s6:   if(a[i] != '1') goto s1; i++;
s7:   if(a[i] != '1') goto s1; i++;
   return i-8;
}
```



BLAST

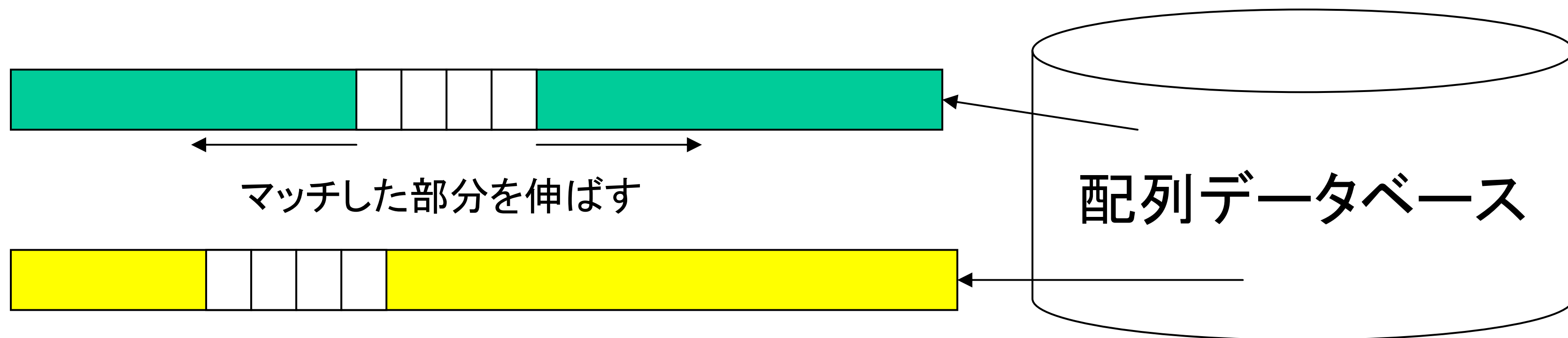
記号列





長さ l の部分文字列の集合 W を生成

W の元を部分にもつ配列を検索
(Knuth-Morris-Pratt, Aho-Corasick)



Kunth-Morris-Pratt の逆走査版 (Good Suffix Heuristics)

10101111

10100111

10100111

パターンを後ろからマッチ

111 がパターン中に存在しない

一気に7文字ずらす

Boyer – Moore


```
       0
10110101
 10110101
```

最後から　1　文字目が不一致
ならばパターン全体を　1　ずらす

next[1]=1

```
      11
10110101
    10110101
```

最後から　2　文字目が初めて不一致
ならばパターン全体を　4　ずらす

next[2]=4

```
     001
10110101
       10110101
```

最後から　3　文字目が初めて不一致
ならばパターン全体を　7　ずらす

next[3]=7

Boyer – Moore


```
    1101
10110101
  10110101     next[4]=2
   00101
10110101
     10110101 next[5]=5
10110101
     10110101 next[6]=5
10110101
     10110101 next[7]=5
```

Boyer – Moore

不一致文字法：　パターンに出現しない文字を利用して、パターンを大幅にスキップ　(Bad Character Heuristics)

文字列 a

```
    i     i    i                      i   i
A STRING SEARCHING EXAMPLE CONSISTING OF
    j
STING     j
     STING    j
          STING
               STING
                    STING
                         STING    j
                              STING  j
                                 STING
```

移動する距離の表					
S	T	I	N	G	他
4	3	2	1	0	M

R は現れない
p を M 移動
i = i+5
j = M-1

S は現れる
p を 4 移動
i = i+4
j = M-1

T は現れる
パターンを3移動
i = i+3
j = M-1

パターン p　長さ M

Boyer – Moore


```
    i
SINGING STING
    j
STING
```

```
        i
SINGING STING
        j
STING
```

```
     i
SINGING STING
     j
  STING
```

i=i+2
j=5-1

既に調べた部分(長さ M-j)
は再度照合しないようにスキップ
(この場合 j=1 に注意)

```
          i
SINGING STING
          j
    STING
```

スキップする距離が
既に調べた長さ M-j を
下回る場合には
i を M-j だけ移動

移動する距離の表

S	T	I	N	G	他
4	3	2	1	0	5

Boyer – Moore


```
static final int SIZE = 256;

public static int mischarsearch(String p, String a){
          int[] skip = new int[SIZE+1];
          int M = p.length();

          // まずスキップする文字数を skip[]に計算
          for(int i=0; i<=SIZE; i++)
                    skip[i]=M;              // パターンの長さだけずらすように初期化
          for(int i=0; i < M-1; i++)
                    skip[p.charAt(i)] = M-1-i;
                    // 同じ文字が複数出現したら「一番右の文字」の位置までスキップするのに必要な長さを登録

          // 文字照合を開始
          int i, j;
          for(i=M-1, j=M-1; j >= 0; i--, j--)
                    while(a.charAt(i) != p.charAt(j)){
                              int t = skip[a.charAt(i)];
                                        // テキスト a の i番目の文字と一致するパターンの文字へスキップする距離
                              if(M-j > t)       // 既に照合した長さ M-j を t が下回るなら M-j だけスキップ
                                        i = i + M-j;
                              else
                                        i = i + t;
                              if(i >= a.length())            // パターンがみつからなかった
                                        return -1;
                              j = M-1;                       // 最右の文字から照合を再開
                    }
          return i+1;
                    // j が -1 になり for ループを抜ける. i も 1 だけ余計に減らされているので1増やす.
}
```

Boyer - Moore

Boyer-Moore　は
　Knuth-Morris-Pratt の逆走査法
　不一致文字法
を混合し、各時点でより大きなスキップ幅を選択する方法

Knuth-Morris-Pratt の逆走査法を使用するので
文字の比較回数は M+N に比例する

不一致文字法だけだと、M+N に比例するとは限らない

```
0000000000000000000000000000000000000
1000000
 1000000
```

Boyer − Moore

文字の種類が多いほどパターンに現れない文字の出現で大幅にスキップできる　⇒　0,1　の2進列に不向き

解決案

b個のビットでブロックとしてまとめ、ブロック化文字として扱う

長さ　M　のパターン　p　に　(M-b+1)　個のブロック化文字

長大なテキスト　a　に　2^b　個のブロック化文字があるとき

$$M-b+1 \ll 2^b$$

であれば不一致文字が増える

Rabin - Karp

a

p

長さ M のパターン p にたいして
テキスト a のすべての M 文字列 x が
p と一致するかをハッシュ関数 h で判定

h(p) = h(x) ?
h(k) = k mod q (q は大きな素数)

1つのパターン p をキーとして探索できればよい
ハッシュ表を保持しなくてよいメリット

ハッシュ関数の計算が重たい？

Rabin - Karp

文字は d 種類、 各文字 a[i] は $0 \leqq a[i] < d$ をみたす整数

$$x = a[i]d^{M-1} + a[i+1]d^{M-2} + ... + a[i+M-1]$$

a[i], a[i+1],..., a[i+M-1],a[i+M]

$$(x - a[i]d^{M-1})d + a[i+M]$$

- 差分だけを計算
- mod q は算術演算より先に実行して
  大きな数の計算を未然に防ぐ

Rabin – Karp


```
public static int rkSearch(String a, String p){
        int N = a.length();
        int M = p.length();
        if(M > N){ return -1;}
        int q = 33554393;      // 大きな素数 4バイトで表現可能
        int d = 32;     // 文字の種類は高々 D 個
        int dM = 1;     // d の M-1 乗 (M 乗でない) を dM に計算
        for(int i = 1; i < M; i++){  dM = (d * dM) % q; }
        int ha = 0;     // a のハッシュ値
        int hp = 0;     // p のハッシュ値
        for(int i = 0; i < M; i++){
                ha = ( ha * d + ((int)a.charAt(i) % d) ) % q;
                hp = ( hp * d + ((int)p.charAt(i) % d) ) % q;
        }
        int i;
        for(i = 0; hp != ha && i+M < N; i++){
                int tmp = (ha - ((int)a.charAt(i) % d) * dM + d * q) % q;
                        // d * q は (...) が正になるように余分に追加
                ha = (tmp * d + ((int)a.charAt(i+M) % d)) % q;
        }
        if(N <= i+M){  return -1;}     // マッチしなかった
        else{          return i; }
}
```

Rabin − Karp

異なる文字列のハッシュ値が一致する可能性あり

h(k) = k mod q の素数 q の値を非常に大きくすれば
この可能性を限りなく 0 にできる

Rabin−Karp の方法は M+N にほぼ比例する時間で動作
異なる文字列のハッシュ値が一致する可能性が高い場合、
最悪 O(MN) の計算時間

文字列探索
（不完全マッチ／アラインメント）

不完全なマッチ

- ゲノム、遺伝子配列には読み間違いがある
- Genome Shotgun Assembly では read を繋げてゆく

  生物情報科学実験２では、
  このソフトウエアを作成
- EST を収集したら類似性の高い配列をグループ化

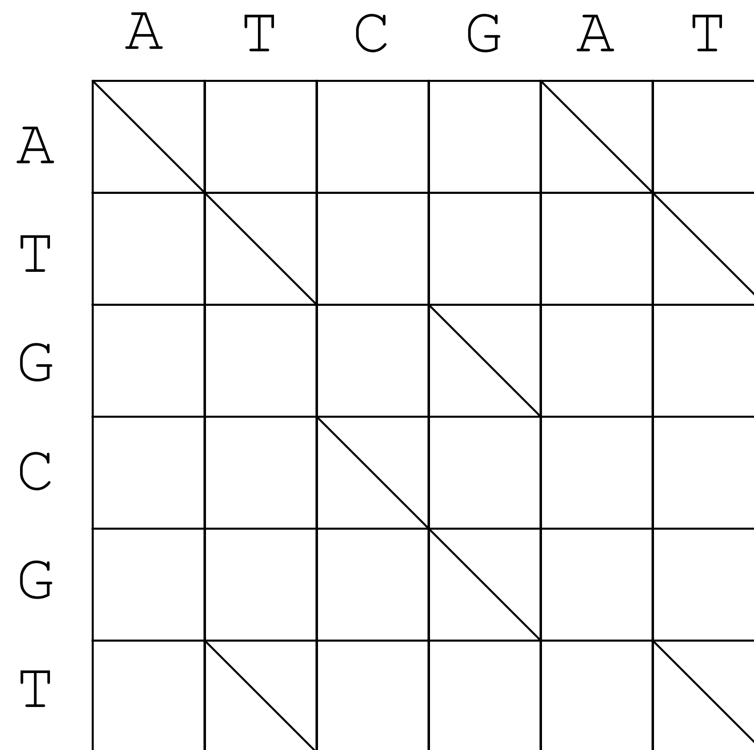
配列アラインメント

配列アラインメント

```
AT-CGAT        ATGCGATTAG
||  || |          ||  ||
ATGCG-T           CG-TT
```

全域的　　　　局所的

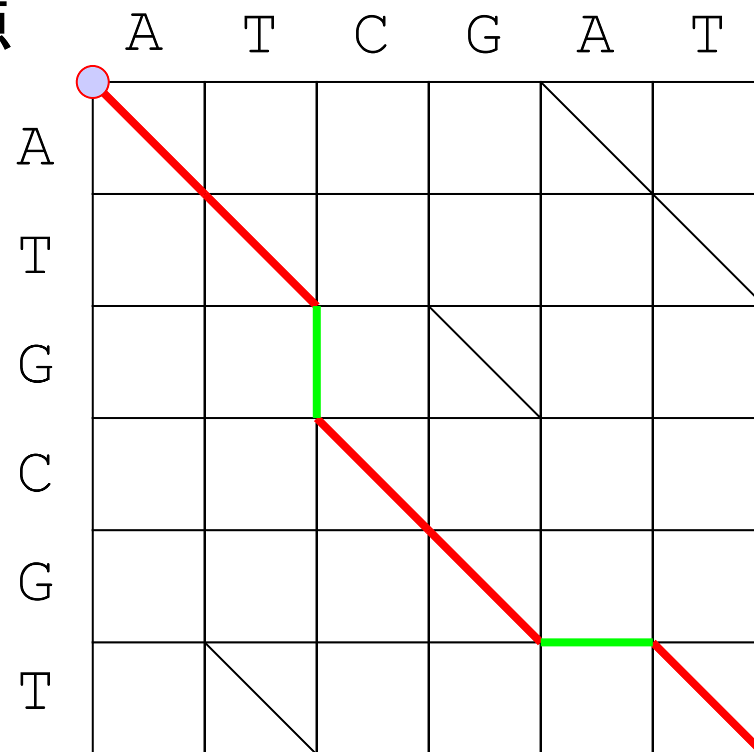
全域的アラインメント



Edit graph

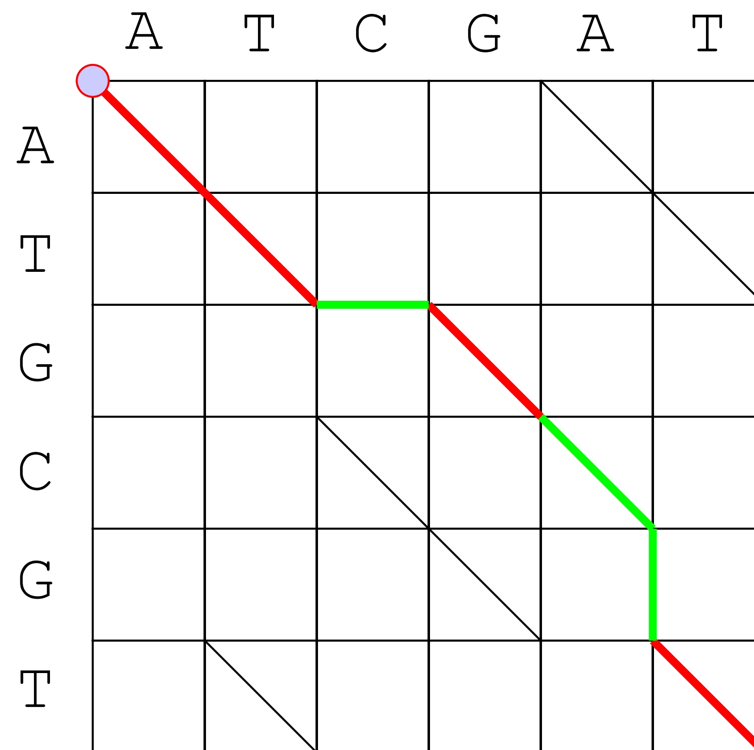
全域的アラインメント

出発点




```
AT-CGAT
|| || |
ATGCG-T
```

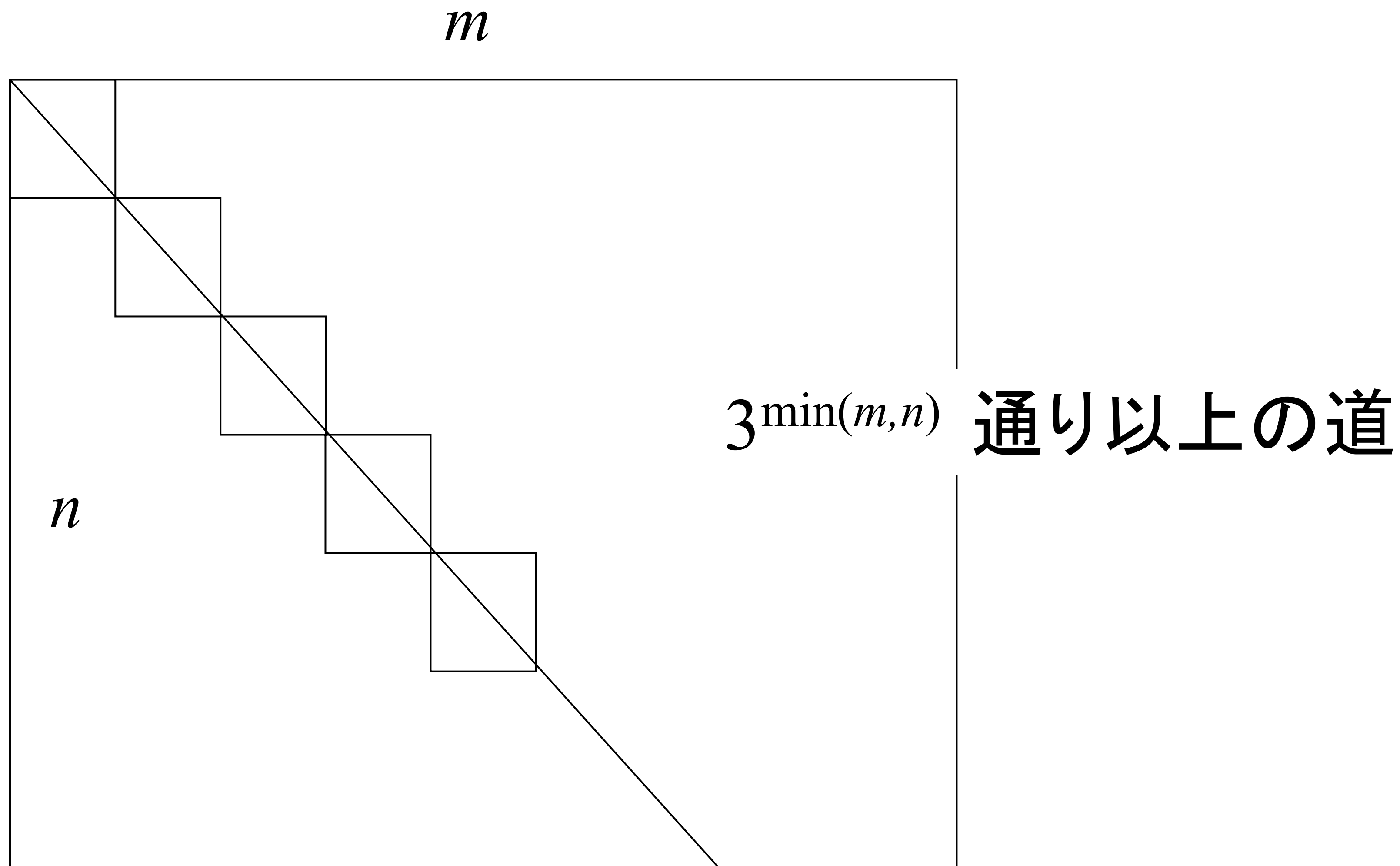
全域的アラインメント



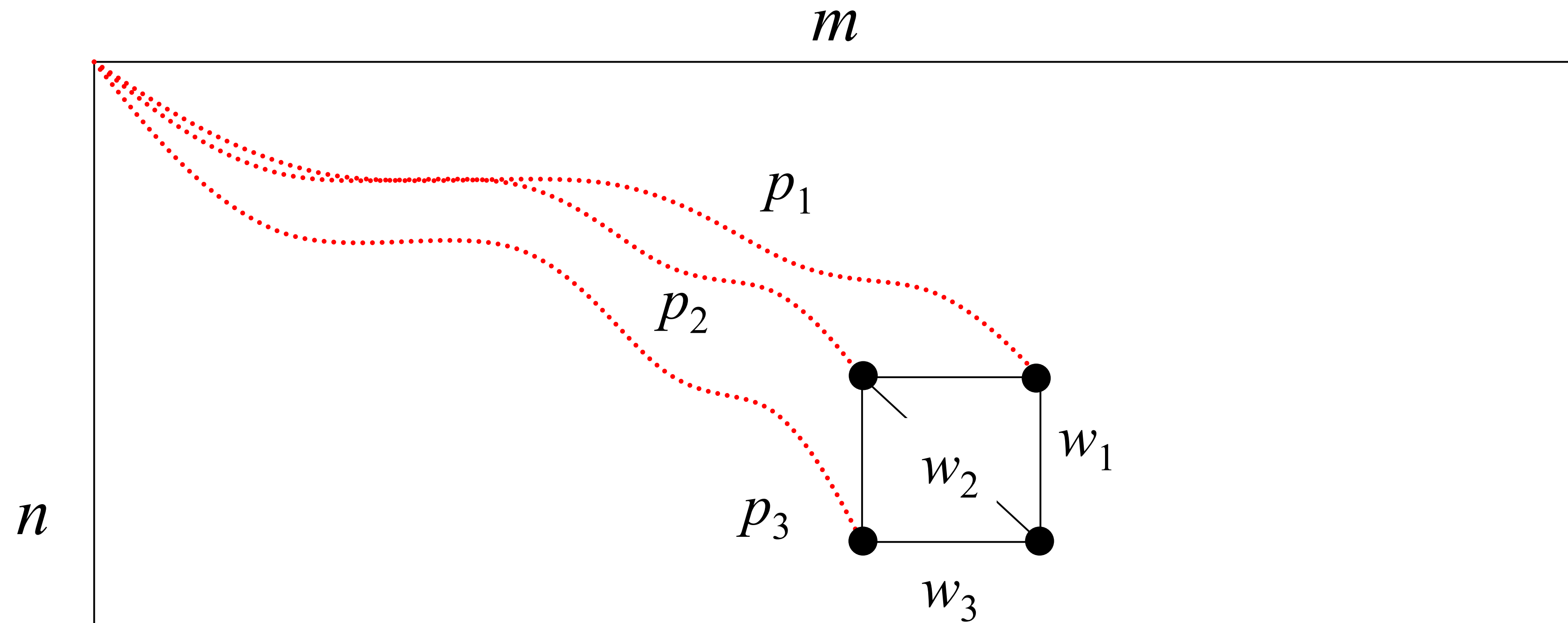
ATCGA-T

AT-GCGT

道の数は組合せが多い



動的計画法



重さの総和が最大の道は p_1+w_1, p_2+w_2, p_3+w_3 のどれか

重さ最大の道の値だけを記憶する

記憶する場所は格子点の数 $(m+1)(n+1)$ だけですむ

全域的アラインメント

$j \rightarrow$, $i \downarrow$

		y_1 A	y_2 T	y_3 C	y_4 G	y_5 A	y_6 T
x_1	A	1	0.5	0	-0.5	-1	-1.5
x_2	T	0.5	2	1.5	1	0.5	0
x_3	G	0	1.5	1.5	2.5	2	1.5
x_4	C	-0.5	1	2.5	2	2	1.5
x_5	G	-1	0.5	2	3.5	3	2.5
x_6	T	-1.5	0	1.5	3	3	4

$i = 0 \quad \text{or} \quad j = 0: \quad Score(i,j) = -\infty$

ただし例外として　$Score(0,0) = 0$

$i > 0 \text{ and } j > 0:$

$$Score(i,j) = \max \begin{cases} Score(i-1, j-1) + match(x_i, y_j) \\ Score(i-1, j) - gap \\ Score(i, j-1) - gap \end{cases}$$

$gap = 0.5$

$$match(x,y) = \begin{cases} 1 & x = y \\ -0.5 & x \neq y \end{cases}$$

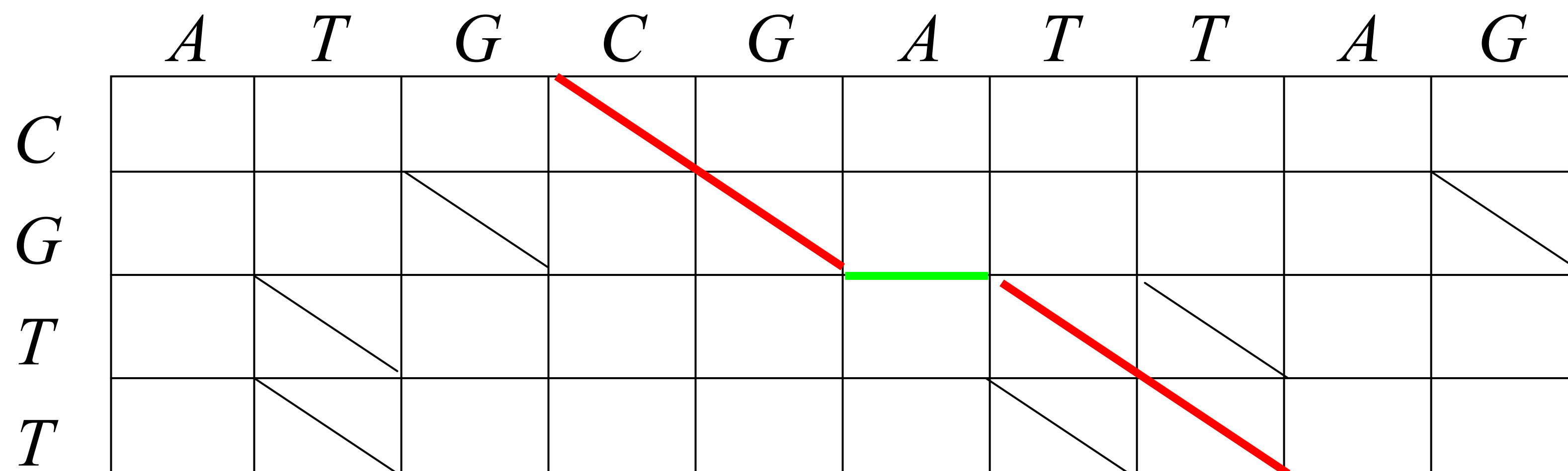
1	0.5
0.5	

全域的アラインメントの復元

	A	T	C	G	A	T
A	1	0.5	0	-0.5	-1	-1.5
T	0.5	2	1.5	1	0.5	0
G	0	1.5	1.5	2.5	2	1.5
C	-0.5	1	2.5	2	2	1.5
G	-1	0.5	2	3.5	3	2.5
T	-1.5	0	1.5	3	3	4

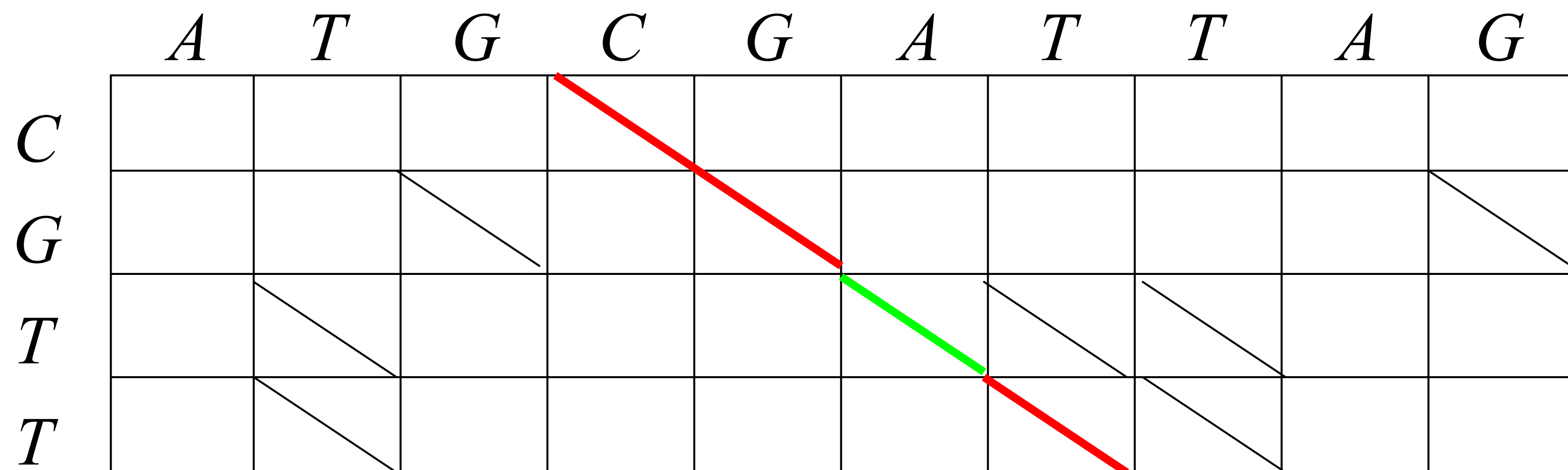
```
AT-CGAT
|| || |
ATGCG-T
```

局所的アラインメント




```
ATGCGATTAG
   || ||
   CG-TT
```

局所的アラインメント



ATGCGATTAG

CGTT

局所的アラインメント

		y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}
		A	T	G	C	G	A	T	T	A	G
x_1	C	0	0	0	1	0.5	0	0	0	0	0
x_2	G	0	0	1	0.5	2	1.5	1	0.5	0	1
x_3	T	0	1	0.5	0.5	1.5	1.5	2.5	2	1.5	1
x_4	T	0	1	0.5	0	1	1	2.5	3.5	2.5	2

$i = 0 \text{ or } j = 0: \quad Score(i, j) = 0$

$i > 0 \text{ and } j > 0:$

[Smith, Waterman 1981]

$$Score(i, j) = \max \begin{cases} 0 \\ Score(i-1, j-1) + match(x_i, y_j) \\ Score(i-1, j) - gap \\ Score(i, j-1) - gap \end{cases}$$

$$gap = 0.5$$

$$match(x, y) = \begin{cases} 1 & x = y \\ -0.5 & x \neq y \end{cases}$$

局所的アラインメントの復元

	A	T	G	C	G	A	T	T	A	G
C	0	0	0	1	0.5	0	0	0	0	0
G	0	0	1	0.5	2	1.5	1	0.5	0	1
T	0	1	0.5	0.5	1.5	1.5	2.5	2	1.5	1
T	0	1	0.5	0	1	1	2.5	3.5	2.5	2

```
ATGCGATTAG
   ||  ||
   CG-TT
```

動的計画法によるアラインメント計算時間的限界

- ２つの配列長を m と n
- $(m+1)*(n+1)$ 個の値を計算

- たとえば

$m = 200{,}000,\ \ n = 2{,}500,\ \ 220$ 秒

(Fujitsu PrimePower 1000, clock rate of 675MHz
ちょっと古いスペックですが)

計算量

計算量

決定不能問題 （ゲーデル、チューリング 1930年代）
プログラムの停止性の判定 定理の自動証明

$\Omega\left(2^{2^{\cdot^{\cdot^{2^N}}}}\right)$

$O(2^N)$ NP困難問題 NP完全問題 （クック、カープ 1972年）
最適なマルチプルアラインメント （近似解CLUSTAL W）
最適なクラスタリング 最適なクラス分類（決定木など）
近似解法 探索法が試みられている

? ?

$O(N^k)$

$O(N^2)$ 2つのN塩基（残基）配列のアラインメント（Smith-Waterman）
最短経路

$O(N \lg N)$ N個の数のソーティング(merge sort, heap sort)

$O(N)$ 文字列の探索、 N 個の数の最大値