A-001

辞書順に並ぶ順列のランク付け操作とその逆操作に対する $O(n \log\log n)$ 領域を用いた線形時間アルゴリズムの簡単化

Simplified Linear-Time Algorithms for Lexicographic Ranking and Unranking Permutations with $O(n \log\log n)$ Space

明田川 卓† 三河 賢治‡
Takashi Aketagawa Kenji Mikawa

１．はじめに

本論文では，辞書順の順列に対するランクとアンランクを計算する線形時間アルゴリズムを提案する．順列のランクとは順列集合 S_n から $\{0,\dots,n!-1\}$ への全単射であり，その反対に，順列のアンランクとは $\{0,\dots,n!-1\}$ から S_n への全単射である．ランク，アンランクを計算することをそれぞれランク付け操作，ランク付けの逆操作とよぶ．ランク付けやその逆操作を扱う場合，四則演算やビット演算に要する時間計算量を $O(1)$ 時間とする計算モデル上で議論することが一般的である．本論文もこれまでの研究にならい，四則演算およびビット演算を定数時間で実現する計算モデル上で議論を進めていく．

辞書順に並べられた文字列長 n の順列に対するランク付けおよびその逆操作では，順列を構成する各要素の逆位（inversion）を用いて，$O(n^2)$ 時間で計算する素朴なアルゴリズムが知られている．また，二分探索法や結合ソート法を用いて，素朴なアルゴリズムの時間計算量を $O(n \log n)$ 時間に改良できることも知られている．最近，ビット演算を用いて，辞書順の順列のランク付けおよびその逆操作を線形時間で計算するアルゴリズムが発表された．Mares らは文献[1]で $O(n \log n)$ ビットのビットベクトルを用意し，ランク付けの逆操作は自然なビット演算で実現し，ランク付けは特殊な比較操作とビットベクトルの和を定数回の剰余算で求める方法を用い，初めて線形時間アルゴリズムを実現した．須藤らは文献[2]で $O(n \log\log n)$ ビットのビットベクトルを用意し，線形時間のランク付けおよびその逆操作を実現しながら領域計算量を $O(n \log\log n)$ ビットに改善した．須藤らが提案した手法は，集合 $X_n = \{0,\dots,n-1\}$ 上の置換の集合 S_n のランク付けおよびその逆操作をするために，X_n を $\lceil \log n \rceil$ 個ずつに分割して要素を管理するというアイデアに基づいている．一方，辞書順とはならない順列のランク付けおよびその逆操作をする問題は，Myrvold らによって時間計算量と領域計算量ともに $O(n)$ となる線形時間アルゴリズムが発表されている[3].

本論文は，Mares らと須藤らのアルゴリズムを改善し，よりシンプルに計算機に実装可能な線形時間アルゴリズムを提案する．須藤らのアルゴリズムでは，X_n を $\lceil \log n \rceil$ 個ずつの部分集合に分割し，部分集合に対応する各ランク値を埋め込んだビットベクトルをランク付けに利用している．ランク付けの逆操作も，ランク付けと同様に，アンランク値をビットベクトルに埋め込んでいる．これらの値をビットベクトルに埋め込む方法は，計算機への実装が容易である反面，ランク付け，およびその逆操作を複雑なものにしている．本論文では，ビットベクトルに埋め込む値を工夫して，ランク付けおよびその逆操作を行う．

解の全探索が必要となる巡回セールスマン問題や組合せパズルのような問題では，厳密解を得るために全ての順列を生成し，解の候補と順列を照合する場合が多い．これまでは，スーパーコンピュータのような大型計算機であっても計算機のリソース不足のため解の全探索を行うことが難しかったが，CPU の高速化や搭載メモリの大容量化にともない，現実的な計算時間で解の全探索を行うことができるようになってきた．それゆえに，近年，解の全探索を高速化するための手法の開発が活発に行われている．本研究の成果は，このような問題に対して，ランクに対応する順列パターンを高速に生成し，かつ，計算機のメモリ消費量の改善に貢献する．

２．定義

本論文では，集合 $X_n = \{0,\dots,n-1\}$ 上の順列πについて考える．ここで順列πとは n 組の数字の列で，順列πの i 番目の要素を $\pi[i]$ と表し，$\pi[i,\dots,j] \equiv (\pi[i],\dots,\pi[j])$ とすると，

$$\pi = (\pi[0],\dots,\pi[n-1]) \wedge$$
$$\forall \mathrm{i}(\pi[i] \in X_n) \wedge \forall \mathrm{i},\forall \mathrm{j},\mathrm{i} \neq \mathrm{j}(\pi[i] \neq \pi[j])$$

が成り立つπが順列である．つまり，集合 X_n の元を並び替えて，順序も考慮した n 組の数字の列が，順列である．ここで順列 $\pi = (\pi[0],\dots,\pi[n-1])$ をポジションベクトル（position vector）と呼ぶことにする．

順列のランク付けアルゴリズムを設計する上で，順列のランクを求めるための，順列の途中段階が必要になるかもしれない．そこで X_n の部分集合Aを考え（つまり $A \subseteq X_n$），A上の順列πを考えよう．以降しばらくこの章では，πはA上の順列を表す．また，A上の順列の集合を P_A と表す．P_A には集合Aの元を並び替えて作られる全ての順列が含まれる．集合Aの要素数を m とし，集合A上の順列π，π' に対し，πが π' よりも辞書順で前であることを $\pi < \pi'$ と表し，次のように定義する．

$$\pi < \pi' \Leftrightarrow_{def} \pi[0] < \pi'[0] \vee$$
$$(\pi[0] = \pi'[0] \wedge \pi[1,\dots,m-1] < \pi'[1,\dots,m-1])$$

集合A上の順列πの辞書順ランクを $R(\pi, A)$ と表し，次のように定義する．

$$R(\pi, A) \Leftrightarrow_{def} |\{\pi' \in P_A | \pi' < \pi\}|$$

すなわち，$R(\pi, A)$ は集合A上の順列の集合 P_A の中で，πが辞書順で何番目にあるかを示す．

例えば，$A = \{0,1,2\}$ のとき，集合A上の順列は全部で $3! = 6$ 通り存在するが，そのそれぞれに対して辞書順ランク $R(\pi, A)$ は次のようになる．

$$R((0,1,2),A) = 0 \qquad R((0,2,1),A) = 1$$
$$R((1,0,2),A) = 2 \qquad R((1,2,0),A) = 3$$
$$R((2,0,1),A) = 4 \qquad R((2,1,0),A) = 5$$

†新潟大学大学院自然科学研究科
‡新潟大学情報基盤センター

また，集合Aに対する，要素$x \in X_n$のランク$r(x,A)$を次のように定義する．

$$r(x,A) \Leftrightarrow_{def} |\{y \in A | y < x\}|$$

すなわち，$r(x,A)$は集合Aを昇順で並べたときに x が何番目の要素であるかを示す．また，r の逆関数も用意しておこう．集合Aを昇順で並べたときの i 番目の要素を x とすると，$r^{-1}(i,A) = x$となる．

再び集合$X_n = \{0,\dots,n-1\}$上の順列πについて考えよう．

$$(r(\pi[0],X_n), r(\pi[1],X_n \backslash \{\pi[0]\}), r(\pi[2],X_n \backslash \{\pi[0],\pi[1]\}), \dots, r(\pi[n-1],\{\pi[n-1]\}))$$

を順列πのインバーションベクトル（inversion vector）と言う．集合X_n上の順列πのインバーションベクトルは必ず最初はπ[0]，最後は 0 となる．この順列πのインバーションベクトルを，$(I[0],\dots,I[n-1])$とおこう．

例えば，$X_3 = \{0,1,2\}$のとき，集合X_3上の順列πのそれぞれに対して，インバーションベクトルは次のようになる．

$\pi = (0,1,2)$のとき$(0,0,0)$　　$\pi = (0,2,1)$のとき$(0,1,0)$
$\pi = (1,0,2)$のとき$(1,0,0)$　　$\pi = (1,2,0)$のとき$(1,1,0)$
$\pi = (2,0,1)$のとき$(2,0,0)$　　$\pi = (2,1,0)$のとき$(2,1,0)$

さらに，順列πの辞書順ランク$R(\pi,X_n)$について次式が成り立つことが知られている．

$$R(\pi,X_n) = \sum_{i=0}^{n-1} I[i] \cdot (n-i-1)! \quad (1)$$

1 章でも言及したように，四則演算にかかる時間を$O(1)$時間とみなすのが一般的である．よって(1)式により，順列πの辞書順ランク$R(\pi,X_n)$とインバーションベクトル$(I[0],\dots,I[n-1])$へは相互に線形時間で変換できる（図 2.1, 2.2）．しかし，インバーションベクトルからポジションベクトル$(\pi[0],\dots,\pi[n-1])$への変換やその逆変換は長い間線形時間では不可能とされてきた．次章以降では，Mares ら，須藤らのアルゴリズムを元に，実際に簡単化した順列の辞書順線形時間ランク付けとその逆操作のアルゴリズムを提案していく．

３． 順列の辞書順ランク付け

2 章では順列の辞書順ランク付けに関する様々な定義をし，また最後に順列の辞書順ランク付けとその逆操作において，インバーションベクトルからポジションベクトルへの変換とその逆変換がネックになっていることに言及した．したがって本章では順列の辞書順ランク付けに必要になる，ポジションベクトルからインバーションベクトルへの線形時間変換アルゴリズムを提案することになる．集合X_n上の順列πが与えられて，そのポジションベクトルが線形時間でインバーションベクトルに変換できれば，図 2.1 のアルゴリズムによりこれを辞書順ランク$R(\pi,X_n)$に変換でき，全体として線形時間で順列の辞書順ランク付けができることになる．

それではまず提案アルゴリズムで使用する変数について説明しよう．須藤らのアルゴリズムでは集合X_nを$\lceil \log n \rceil$個ずつに分けて管理していたが，本論文で提案するアルゴリズムでは，集合X_nを$\lfloor \log n \rfloor$個ずつに分けて管理する．わずかな差ではあり，領域計算量のオーダーには変化がないが，同じアルゴリズムならこちらの方がビットベクトルを節約できる．X_nを$\lfloor \log n \rfloor$個ずつの集合に分けたものを，$Y_{(n,j)}$とする．

例えば，n = 9のときは$X_n = \{0,\dots,8\}$，$\lfloor \log n \rfloor = 3$なので，$Y_{(n,0)} = \{0,1,2\}, Y_{(n,1)} = \{3,4,5\}, Y_{(n,2)} = \{6,7,8\}$となる．

X_nは，$\left\lceil \frac{n}{\lfloor \log n \rfloor} \right\rceil$個の$Y_{(n,j)}$に分けることができる．また，$Y_{(n,j)}$のインデックス k の元は，$\lfloor \log n \rfloor \cdot j + k$と表せる．これを$Y_{(n,j)}[k] = \lfloor \log n \rfloor \cdot j + k$とおこう．

2 章と同様に再び集合X_nの部分集合Aを考え，A上の順列πを考える．これに対して，1 つあたりビット長$\lceil \log(\lfloor \log n \rfloor + 1) \rceil$のビットベクトル$v_j[k]$に次の値を入れる．

$$v_j[k] = \left| \{ x \in A \wedge x \in Y_{(n,j)} | x \le Y_{(n,j)}[k] \} \right|$$

つまり，$v_j[k]$には$Y_{(n,j)}$のインデックス k 以下の元のうち，Aの元でもあるものの個数を入れる．

また，1 つあたりビット長$\lceil \log(n+2) \rceil$のビットベクトルa_jに次の値を入れる．

$$\begin{aligned} a_j &= v_j[\lfloor \log n \rfloor - 1] \\ &= \left| \{ x \in A \wedge x \in Y_{(n,j)} | x \le Y_{(n,j)}[\lfloor \log n \rfloor - 1] \} \right| \\ &= \left| \{ x \in A \wedge x \in Y_{(n,j)} \} \right| \end{aligned}$$

ここで$Y_{(n,j)}$とv_jの最大のインデックスは，$\lfloor \log n \rfloor - 1$となる．つまり，$a_j$には$v_j$の最大のインデックスと等しい値，集合 A と集合$Y_{(n,j)}$の共通元の個数を入れる．$v_j$と$a_j$のビット長は，提案アルゴリズムに最低限必要な長さとした．

提案アルゴリズムでは，集合 A の初期値をA = ∅とし，集合$X_n = \{0,\dots,n-1\}$上の順列πのインバーションベクトル$(I[0],\dots,I[n-1])$を求めるために，順列πのポジションベクトル$(\pi[0],\dots,\pi[n-1])$を，右から左へ，i = n − 1から i = 0に向かって見ていき，その順にI[i]を計算していく．その際，インバーションベクトルの 1 つの要素I[i]が計算し終わるごとに，現在見ていたポジションベクトルの 1 つの要素π[i]を集合 A の元に加え，新しくなった集合 A に応じて，ビットベクトルv_jとa_jを更新する．

```
入力：インバーションベクトル(I[0],…,I[n−1])
出力：辞書順ランク R(π,X_n)

rank ← 0
j ← 1
for(i = n − 1; i ≥ 0; i − −){
    rank ← rank + I[i] · j
    j ← j · (n − i)
}
return rank
```

図 2.1　インバーションベクトルから辞書順ランクへの変換アルゴリズム

```
入力：辞書順ランク R(π,X_n)
出力：インバーションベクトル(I[0],…,I[n−1])

rank ← R(π,X_n)
j ← 1
for(i = 1; i < n; i + +){j ← j · i}
for(i = 0; i < n; i + +){
    I[i] ← ⌊rank/j⌋
    rank ← rank mod j
    if(j ≠ n − 1){j ← j/(n − i − 1)}
}
return (I[0],…,I[n−1])
```

図 2.2　辞書順ランクからインバーションベクトルへの変換アルゴリズム

なお，A = ∅よりビットベクトルv_j，a_jの初期値は全て 0 である．

このとき，集合 A，ビットベクトルv_j，ビットベクトルa_jが正しく更新されているならば，次の補題が成り立つ．

補題 3.1　$\pi[i] = Y_{(n,t)}[tt]$とおくと，

$$\mathrm{I}[i] = v_t[tt] + \sum_{k=0}^{t-1} a_k$$

が成り立つ．

証明）

$\pi[i] = Y_{(n,t)}[tt]$とおいたので，インバーションベクトルとランク$\mathrm{r}(x, A)$の定義より，

$$\begin{aligned}\mathrm{I}[i] &= r(\pi[i], A)\\ &= \mathrm{r}\left(Y_{(n,t)}[tt], A\right)\\ &= \left|\{y \in A | y < Y_{(n,t)}[tt]\}\right|\\ &= \left|\{y \in A \wedge \left(y \in Y_{(n,0)} \vee \cdots \vee y \in Y_{(n,t)}\right) | y < Y_{(n,t)}[tt]\}\right|\end{aligned}$$

$Y_{(n,j)}$は集合X_nを重複なく分割した集合なので，

$$\begin{aligned}\mathrm{I}[i] = &\left|\{y \in A \wedge y \in Y_{(n,0)} | y < Y_{(n,t)}[tt]\}\right| + \cdots +\\ &\left|\{y \in A \wedge y \in Y_{(n,t)} | y < Y_{(n,t)}[tt]\}\right|\end{aligned}$$

また，$\mathrm{j} < \mathrm{t}$のとき$Y_{(n,j)}$の元 y は必ず

$$\mathrm{y} \le Y_{(n,j)}[\lfloor \log n \rfloor - 1] < Y_{(n,t)}[tt]$$

を満たす．さらに，集合 A に$\pi[i] = Y_{(n,t)}[tt]$を加えるのは，$\mathrm{I}[i]$の計算後なので，$Y_{(n,t)}[tt] \notin A$である．以上により$\mathrm{I}[i]$の式を変形すると，

$$\begin{aligned}\mathrm{I}[i] = &\left|\{y \in A \wedge y \in Y_{(n,0)}\}\right| + \cdots +\\ &\left|\{y \in A \wedge y \in Y_{(n,t-1)}\}\right| +\\ &\left|\{y \in A \wedge y \in Y_{(n,t)} | y \le Y_{(n,t)}[tt]\}\right|\\ = &\ v_t[tt] + \textstyle\sum_{k=0}^{t-1} a_k\end{aligned}$$

□

提案アルゴリズムでは，補題 3.1 をもとに線形時間でインバーションベクトルを計算する．ここで，$v_t[tt]$はビット演算によりビットベクトルv_tから値を取り出し，ビットベクトルa_jの和は Mares らも用いていた次の性質により計算する[1].

性質 3.1　1 つあたりビット長 b のビットベクトル z の各フィールドの総和は，それが$2^b - 1$未満ならば，$\mathrm{z} \bmod (2^b - 1)$により計算できる．

この性質の証明は Mares らの論文を参照して欲しい．

それでは，ビットベクトルの更新方法について説明しよう．ポジションベクトルの 1 つの要素$\pi[i] = Y_{(n,t)}[tt]$を集合 A の元に加えると，ビットベクトルv_t，a_tを更新する必要性が生じる．

ビットベクトル$v_t[k]$は$\mathrm{k} \ge \mathrm{tt}$について 1 ずつ増えるので，ビットベクトル$v_t$は次の式により更新する．

$$v_t' = v_t + allone\_v \& \sim\left(\left(1 \ll (tt \cdot bitlength\_v)\right) - 1\right)$$

ここで，$allone_v$は全体のビット長がv_tと同じ長さであり，1 が$v_t[k]$のビット長と同じ間隔で現れるビットベクトルである（須藤らも同じようなビットベクトルを用意していた）．

&はビット単位の AND 演算であり，~はビット反転であり，≪はビットの左シフトである．また，$\mathrm{bitlength_}v = \lceil \log(\lfloor \log n \rfloor + 1) \rceil$である．

ビットベクトルaは次の式により更新する．

$$a' = a + \left(1 \ll (t \cdot bitlength\_a)\right)$$

ここで，$\mathrm{bitlength_}a = \lceil \log(n+2) \rceil$である．

以上の議論により，図 3.1 のアルゴリズムが導かれる．ここまでで提案アルゴリズムの説明が一通り終わったが，計算量の評価をしてみよう．

入力：ポジションベクトル$(\pi[0], \ldots, \pi[n-1])$
出力：インバーションベクトル$(I[0], \ldots, I[n-1])$

for$\left(j = 0; j < \left\lceil \frac{n}{\lfloor \log n \rfloor} \right\rceil; j++\right)\{v_j \leftarrow 0\}$
$a \leftarrow 0$
$\mathrm{b} \leftarrow \lfloor \log n \rfloor$
$\mathrm{bitlength\_}v \leftarrow \lceil \log(\lfloor \log n \rfloor + 1) \rceil$
$\mathrm{bitlength\_}a \leftarrow \lceil \log(n+2) \rceil$
$\mathrm{bitmask\_}v \leftarrow (1 \ll \mathrm{bitlength\_}v) - 1$
$\mathrm{bitmask\_}a \leftarrow (1 \ll \mathrm{bitlength\_}a) - 1$
$\mathrm{allone\_}v \leftarrow 0$
for$(j = 0; j < b; j++)$
　$\mathrm{allone\_}v \leftarrow \mathrm{allone\_}v \ll \mathrm{bitlength\_}v$
　$\mathrm{allone\_}v \leftarrow \mathrm{allone\_}v + 1$
}
for$(i = n-1; i \ge 0; i--)$
　$\mathrm{x} \leftarrow \pi[i]$
　$\mathrm{t} \leftarrow \lfloor \mathrm{x}/\mathrm{b} \rfloor$
　$\mathrm{tt} \leftarrow \mathrm{x} - \mathrm{t} \cdot \mathrm{b}$
　$\mathrm{v\_t\_tt} \leftarrow v_t \& \left(\mathrm{bitmask\_}v \ll (tt \cdot \mathrm{bitlength\_}v)\right)$
　$\mathrm{v\_t\_tt} \leftarrow \mathrm{v\_t\_tt} \gg (tt \cdot \mathrm{bitlength\_}v)$
　$a\_sum \leftarrow \left(a \,\&\left(\left(1 \ll (t \cdot \mathrm{bitlength\_}a)\right) - 1\right)\right)$
　　$mod\ \mathrm{bitmask\_}a$
　$\mathrm{I}[i] \leftarrow \mathrm{v\_t\_tt} + a\_sum$
　$v_t \leftarrow v_t + allone\_v \&$
　　$\sim\left(\left(1 \ll (tt \cdot bitlength\_v)\right) - 1\right)$
　$a \leftarrow a + \left(1 \ll (t \cdot bitlength\_a)\right)$
}
return $(I[0], \ldots, I[n-1])$

図 3.1　ポジションベクトルからインバーションベクトルへの変換の提案アルゴリズム

時間計算量は四則演算およびビット演算にかかる時間を$\mathrm{O}(1)$時間としているので，全体では$\mathrm{O}(n)$時間，つまり線形時間である．

領域計算量はビットベクトルaに必要とする領域は，$\lceil \log(n+2) \rceil \cdot \left\lceil \frac{n}{\lfloor \log n \rfloor} \right\rceil$ビットであり，n が非常に大きくなれば$\lceil \log(n+2) \rceil \fallingdotseq \lfloor \log n \rfloor$であり，ビットベクトル$a$については，$\mathrm{O}(n)$ビット，ビットベクトル$v$に必要とする領域は，$n \cdot \lceil \log(\lfloor \log n \rfloor + 1) \rceil$ビットであり，ビットベクトル$v$については$\mathrm{O}(n \log\log n)$ビットであり，全体としては領域計算量は$\mathrm{O}(n \log\log n)$ビットである．

したがって，次の定理が成り立つ．

定理 3.1　集合X_n上の順列πが与えられたとき，図 3.1 のアルゴリズムは，そのポジションベクトルをインバーションベクトルに，線形時間，領域計算量$\mathrm{O}(n \log\log n)$ビットで変換でき，引き続き図 2.1 のアルゴリズムを実行すれば，同じ計算量で順列πの辞書順ランク$\mathrm{R}(\pi, X_n)$を計算できる

4．順列の辞書順ランク付けの逆操作

3 章では順列の辞書順ランク付けにネックとなる，インバーションベクトルからポジションベクトルへの変換につ

いて，Mares らや須藤らの研究を参考にして，簡単化した線形時間，領域計算量$O(n \log\log n)$ビットのアルゴリズムを提案した．本章では，ランク付けの逆操作にネックとなる，ポジションベクトルからインバーションベクトルへの変換について，同様に Mares らや須藤らの研究を参考にして，ランク付けと同じ計算量であり，かつ簡単化したアルゴリズムを提案する．

ランク付けの逆操作においては，可能な限り，ランク付け操作のアルゴリズムを全く逆にたどったアルゴリズムであることが望ましい．そのため，本章で提案するアルゴリズムは，3 章で提案したアルゴリズムをベースにしているが，完全に逆の操作は行えないかもしれないので，その都度，別のビットベクトルを用意するなどして対応していく．

集合X_n上の順列πの辞書順ランク$R(\pi, X_n)$が与えられたとして，図 2.2 のアルゴリズムによりこれをインバーションベクトルに線形時間で変換でき，このインバーションベクトルをポジションベクトルπに線形時間で変換できれば，全体として線形時間で順列の辞書順ランク付けの逆操作ができることになる．

それではランク付けと同様に最初に提案アルゴリズムで使用する変数から説明を始めよう．ランク付けの逆操作もランク付け操作と同じく，集合X_nを$\lfloor \log n \rfloor$個ずつに分けて管理する．X_nを$\lfloor \log n \rfloor$個ずつの集合に分けたものを$Y_{(n,j)}$とする．X_nは$\left\lceil \frac{n}{\lfloor \log n \rfloor} \right\rceil$個の$Y_{(n,j)}$に分けることができる．$Y_{(n,j)}$の具体例は 3 章を参考にして欲しい．また，$Y_{(n,j)}$のインデックス k の元を$Y_{(n,j)}[k] = \lfloor \log n \rfloor \cdot j + k$とおく．2,3 章と同様に集合$X_n$の部分集合 A を考え，A 上の順列πを考える．

先程説明したように，ランク付けの逆操作においては可能な限りランク付け操作のアルゴリズムを全く逆にたどったアルゴリズムであることが望ましい．したがって，まずポジションベクトルの計算順序と集合 A の扱いを決めよう．

提案アルゴリズムではランク付け操作と全く逆で，集合 A の初期値を$A = X_n$とし，集合$X_n = \{0, \dots, n-1\}$上の順列 π（どのような順列かは計算するまで分からない）のポジションベクトル$(\pi[0], \dots, \pi[n-1])$を求めるために，順列πのインバーションベクトル$(I[0], \dots, I[n-1])$を，左から右へ，$i = 0$から$i = n-1$に向かって見ていき，その順にπ[i]を計算していく．その際，ポジションベクトルの 1 つの要素π[i]が計算し終わるごとに，このπ[i]を集合 A の元から取り除き，新たな集合 A とする．また，新しくなった集合 A に応じてこれから提案する各種ビットベクトルを更新する．

では，ランク付けの逆操作にはどのようなビットベクトルが必要だろうか．ランク付けの逆操作は，求めるπ[i]を$\pi[i] = Y_{(n,t)}[tt]$とおくと，集合X_n上の順列πのインバーションベクトルの 1 つの要素I[i]と集合 A が与えられたとき，この$\pi[i] = Y_{(n,t)}[tt]$を特定する問題ととらえることができる．π[i]を特定するには，t と tt という 2 つの値を特定することが必要になる．したがって，この 2 つの値を特定できるようなビットベクトルが必要だろう．

幸いにも t はランク付けと同じビットベクトルaを少し変形して，次の性質 4.1 を使うことで計算することができる[1]．$a'_j = \sum_{k=0}^{t-1} a_k$とし，$a'_j$を提案アルゴリズムで使うビットベクトルとする．今後本章では，このビットベクトルa'_jを単にビットベクトルa_jまたはビットベクトルaと呼ぶ．また，アルゴリズムの都合上，ビットベクトルa_jを 1 つあたりビット長$\lceil \log n \rceil + 1$のビットベクトルとする．ランク付けの逆操作はランク付け操作と違って，π[i]の計算前は$\pi[i] = Y_{(n,t)}[tt] \in A$であることに注意する．

補題 3.1 の証明の途中を使えば，

$$\begin{aligned} I[i] &= \left|\{y \in A | y < Y_{(n,t)}[tt]\}\right| \\ &< \left|\{y \in A \wedge y \in Y_{(n,0)}\}\right| + \cdots + \\ &\quad \left|\{y \in A \wedge y \in Y_{(n,t)}\}\right| \\ &= a_{t+1} \end{aligned}$$

つまり，$I[i] < a_{t+1}$であり，また$j < t+1$つまり$j \le t$のときは$a_j \le I[i]$である．すなわち，t とは$I[i]$を超えないa_jのうち，j が最大となるものである．この t を次の性質 4.1 により計算する[1].

性質 4.1　a_jを 1 つあたりビット長$b+1$のビットベクトルとし，各フィールドは2^b未満とする．このとき，$a_j \le x$であるフィールドの個数は，a_jと同じビット長のビットベクトルc_jを用意して，c_jの各フィールドに x を入れ，c_jの各フィールドの最上位ビットを 1 にセットすれば，$c - a$を計算することにより，$a_j > x$のときに限り$c - a$の各フィールドの最上位ビットが 0 になるので，$c - a$の各フィールドの最上位ビットの 1 の個数をビット演算により求めれば，$a_j \le x$であるフィールドの個数を計算できる．

性質 4.1 で$x = I[i]$のときの$c - a$の各フィールドの最上位ビットの 1 の個数から 1 を引いたものが t である．また，ビットベクトルa_jの初期値は，$A = X_n$より$a_j = j\lfloor \log n \rfloor$である．

t の計算方法は分かったので，残る課題は tt を計算することである．実は補題 3.1 を少し変形した次の補題 4.1 を用いることで，ランク付けで使用したビットベクトルv_jの情報の一部分を手に入れることができる．この情報は，tt を計算する上で役に立つ．

補題 4.1　$\pi[i] = Y_{(n,t)}[tt]$とおくと，

$$v_t[tt] = I[i] - a_t + 1$$

が成り立つ．

証明）

ランク付けの逆操作は，$\pi[i] = Y_{(n,t)}[tt] \in A$であることに注意して，補題 3.1 の証明途中の式を変形すると，

$$\begin{aligned} I[i] &= \left|\{y \in A \wedge y \in Y_{(n,0)}\}\right| + \cdots + \\ &\quad \left|\{y \in A \wedge y \in Y_{(n,t-1)}\}\right| + \\ &\quad \left|\{y \in A \wedge y \in Y_{(n,t)} | y < Y_{(n,t)}[tt]\}\right| \\ &= v_t[tt] + a_t - 1 \end{aligned}$$

（補題 3.1 と違ってπ[i]の分$v_t[tt]$が 1 つ増える）

ゆえに，$v_t[tt] = I[i] - a_t + 1$

□

提案アルゴリズムでは，補題 4.1 により$v_t[tt]$を計算し，$w_t[v_t[tt]] = tt$を常に満たすビットベクトル$w_j[k]$を用意することで，tt を計算する．また，アルゴリズムの都合上，ビットベクトル$w_j[k]$を 1 つあたりビット長$\lceil \log \lfloor \log n \rfloor \rceil$のビットベクトルとする．

ビットベクトル$w_j[k]$の初期値は，$k = v_t[tt]$のときの tt の値より，

$$k = v_t[tt] = \left|\{x \in A \wedge x \in Y_{(n,t)} | x \le Y_{(n,t)}[tt]\}\right|$$

A の初期値は$A = X_n$のため，$x \le Y_{(n,t)}[tt]$のどの値をとっても$x \in A$を満たす．よって，

$$k = \left|\{x \in Y_{(n,t)} | x \le Y_{(n,t)}[tt]\}\right| = tt + 1$$

ゆえに$tt = k - 1$であり，ビットベクトル$w_j[k]$の初期値は，$w_j[k] = k - 1$である．ただし，$w_j[0] = 0$とする．

t と tt が計算できれば，$\pi[i] = t\lfloor \log n \rfloor + tt$の式よりπ[i]を計算することができる．

それでは，ビットベクトルの更新方法について説明しよう．$\pi[i] = Y_{(n,t)}[tt]$を集合 A の元から取り除くと，ビットベクトルa_j，w_jを更新しなければならない．

ビットベクトルa_jはj > tについて 1 ずつ減るのでランク付けのv_jと同様な次の式により更新する．

$$a' = a - allone\_a\&\sim\left(\left(1 \ll \left((t+1)\cdot bitlength\_a\right)\right) - 1\right)$$

ここで，$allone_a$は全体のビット長がaと同じ長さであり，1 がa_jのビット長と同じ間隔で現れるビットベクトルである．また，$bitlength_a = \lceil \log n \rceil + 1$である．その他の記号は，ランク付けと同じ記号を用いた．

ビットベクトル$w_t[k]$については，$k = v_t[tt]$となる tt の値が存在しなくなり，$k > v_t[tt]$については$v_t[tt]$の値が 1 ずつ減り，tt の値には変化がない．よってビットベクトル$w_t[k]$については，$w_t[k]$を削除し，ビットを右に詰める処理が必要となる．この処理は，プログラムで書けば 3 行ほどであり，もしくは式をまとめれば 1 行で書ける処理である．

以上の議論により，図 4.1 のアルゴリズムが導かれる．ここまでで提案アルゴリズムの説明が一通り終わったが，ランク付けと同様に計算量の評価をしてみよう．

時間計算量はランク付けと同様に四則演算およびビット演算にかかる時間を$O(1)$時間としているので，全体では$O(n)$時間，すなわち線形時間である．

領域計算量はビットベクトルaに必要とする領域は，$(\lceil \log n \rceil + 1)\cdot\left\lceil \frac{n}{\lfloor \log n \rfloor} \right\rceil$ビットであり，n が非常に大きくなれば$(\lceil \log n \rceil + 1) \fallingdotseq \lfloor \log n \rfloor$であり，ビットベクトル$a$については，$O(n)$ビット，ビットベクトル w に必要とする領域は，実装によってはビットベクトルw_jの最下位ビットを無視できるので，$n \cdot \lceil \log \lfloor \log n \rfloor \rceil$ビットであり，ビットベクトル w については$O(n \log\log n)$ビットであり，全体としては領域計算量は$O(n \log\log n)$ビットである．

したがって，次の定理が成り立つ．

定理 4.1　集合X_n上の順列πの辞書順ランク$R(\pi, X_n)$が与えられたとき，図 2.2 のアルゴリズムを実行すれば，これを線形時間でインバーションベクトルに変換でき，引き続き図 4.1 のアルゴリズムを実行すれば，線形時間，領域計算量$O(n \log\log n)$ビットでこのインバーションベクトルをポジションベクトルに変換でき，順列πを求めることができる．

５． 実験

Intel Pentium 4，2.8GHz の CPU を搭載した PC に，CentOS 5.8 をインストールした 32 ビット環境上で我々の提案アルゴリズム，Mares らのアルゴリズム，須藤らのアルゴリズムの実装を行い，ランク付け操作およびその逆操作の実行時間を測定した．ランク付け操作およびその逆操作 1 回分は 1μs 未満で終わってしまうため，1000 万回の操作にかかる時間を 1000 で割ったもの，つまり 1 万回の操作にかかる時間を測定した．ランク付け操作にかかる時間を測定するには，毎回ランダムな順列を生成することが必要となり，無視できない時間となる．そのため，毎回 0 から$n! - 1$までのランダムな整数を生成し，まずランク付けの逆操作を先に測定した．次に，ランク付けの逆操作で生成した順列に対してランク付け操作を行い，ランク付け操作およびその逆操作の合計時間を測定した．ランク付け操作の実行時間はこの 2 つの差分から予測した．結果は表 5.1，5.2 となった．ただし，Mares らのアルゴリズムに関しては，32 ビット環境であるため，n = 8までしか測定していない．

入力：インバーションベクトル$(I[0], \dots, I[n-1])$
出力：ポジションベクトル$(\pi[0], \dots, \pi[n-1])$

```
for(j = 0; j < ⌈n/⌊log n⌋⌉; j++){w_j ← 0}
a ← 0
b ← ⌊log n⌋
bitlength_w ← ⌈log⌊log n⌋⌉
bitlength_a ← ⌈log n⌉ + 1
bitmask_w ← (1 ≪ bitlength_w) − 1
bitmask_a ← (1 ≪ bitlength_a) − 1
allone_a ← 0
for(j = 0; j < ⌈n/⌊log n⌋⌉; j++){
    allone_a ← allone_a ≪ bitlength_a
    allone_a ← allone_a + 1
}
for(j = ⌈n/⌊log n⌋⌉ − 1; j ≥ 0; j−−){
    a ← a ≪ bitlength_a
    a ← a + i · b
}
for(j = n − 1; j ≥ 0; j−−){
    t ← ⌊j/b⌋
    tt ← j − t · b
    w_t ← w_t ≪ bitlength_w
    w_t ← w_t + tt
}
for(j = 0; j < ⌈n/⌊log n⌋⌉; j++){
    w_t ← w_t ≪ bitlength_w }
for(i = 0; i < n; i++){
    x ← I[i]
    c ← (allone_a · x) +
        (allone_a ≪ (bitlength_a − 1))
    t ← ( ( ((c − a) ≫ (bitlength_a − 1))
        &allone_a ) mod bitmask_a ) − 1
    a_t ← a &(bitmask_a ≪ (t · bitlength_a))
    a_t ← a_t ≫ (t · bitlength_a)
    v_t_tt ← x − a_t + 1
    tt ← w_t&(bitmask_w ≪ (v_t_tt · bitlength_w))
    tt ← tt ≫ (v_t_tt · bitlength_w)
    π[i] ← t · b + tt
    a ← a − allone_a&
        ~((1 ≪ ((t + 1) · bitlength_a)) − 1)
    right ← w_t&((1 ≪ (v_t_tt · bitlength_w)) − 1)
    left ← (w_t ≫ bitlength_w)&
        ~((1 ≪ (v_t_tt · bitlength_w)) − 1)
    w_t ← left + right
}
return (π[0], ..., π[n − 1])
```

図 4.1　インバーションベクトルからポジションベクトルへの変換の提案アルゴリズム

表 5.1　1 万回のランク付け操作にかかる時間（単位 ms）

n	Mares らの手法	須藤らの手法	提案手法
4	3	4	4
5	3	5	4
6	5	5	6
7	6	6	5
8	5	6	6
9		7	7
10		7	9
11		7	9
12		8	8

表 5.2　1 万回のランク付け逆操作にかかる時間（単位 ms）

n	Mares らの手法	須藤らの手法	提案手法
4	5	8	9
5	6	9	11
6	6	10	11
7	7	11	14
8	9	13	15
9		15	17
10		16	18
11		17	20
12		19	22

まず，ランク付け操作は 3 アルゴリズムともあまり変わらない実行時間となった．次に，ランク付けの逆操作では Mares らのアルゴリズムがもっとも高速であった．これは Mares らのアルゴリズムは集合X_nの値を全てビットベクトルに格納しているため，領域計算量はかかるが，ランク付けの逆操作は単純なビットベクトルの削除操作で実現しているからだと思われる．また，ランク付けの逆操作では，提案アルゴリズムは須藤らのアルゴリズムより少し遅かった．これは提案アルゴリズムはビットベクトルの初期値の計算に少し時間がかかるためだと思われる．我々のアルゴリズムではビットベクトルの初期値は同じ n に対して毎回同じ値をとるため，この値を記憶するなどすれば，須藤らのアルゴリズムと近い実行時間になると思われる．実験結果をまとめると，我々のアルゴリズムは，領域計算量は須藤らと同じく Mares らよりも改善しており，ランク付け操作およびその逆操作にかかる実行時間は須藤らと大差なく，かつ実装がシンプルなアルゴリズムと言える．

６．まとめ

本論文では，Mares らと須藤らの線形時間ランク付けおよびその逆操作のアルゴリズムを簡単化して，厳密に$O(n \log\log n)$領域を用いて$O(n)$時間で辞書順のランク付けおよびその逆操作のアルゴリズムを提案した．本アルゴリズムは簡単な計算式を用いてランク付けを行い，その逆操作はなるべくランク付け操作のアルゴリズムを全く逆にたどった操作であるようにアルゴリズムを設計した．また，提案アルゴリズムは 5 章で示したように，既存のアルゴリズムと比べて特別遅いというわけでもなかった．今後の研究課題としては，領域計算量は本当にこれで限界であるか考えることや，シミュレーションの上でより高速なアルゴリズムを考えることや，今回扱わなかった辞書順以外の順序や n-m 順列やその他の組合せ構造について考えることなどが挙げられる．

参考文献

[1] Mares M., Straka M., “Linear-Time Ranking of Permutations”, ESA 2007, LNCS 4698 (2007)
[2] 須藤, 篠原, “置換のランク付けに対する $O(n\log\log n)$ ビット領域の線形時間アルゴリズム”, 数理解析研究所講究録, No.1649 (2009).
[3] Myrvold W., Ruskey F., “Ranking and Unranking Permutations in Linear Time”, Inf. Process. Lett., Vol.79 (2000)