データ構造と アルゴリズム I

第8回

小テスト問題

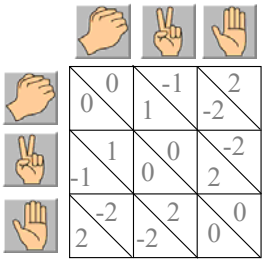

乱択の強さ

- ランダム／行き当たりばったりは悪いこと?
  - 敵／意地の悪い自然の選択に対抗するには有効
  - 最悪の場合（相手に読まれてしまう場合）を回避
- 例：ジャンケンで絶対に負けない方法
  - 確率1/3でグー，チョキ，パーを混ぜる
- ジャンケンで共に確率1/3でプレイする戦略の組合せはナッシュ均衡と呼ばれる
  - 相手がその戦略を取る限り，自分もその戦略を取ることが最適反応となっている

クイズ：ナッシュ均衡は?

- 階段で二人がじゃんけん
- グーで勝てば3歩，チョキで勝てば6歩，パーで勝てば6歩進める
- 先に上りきった方が勝ち
- 絶対に（確率的には）負けない戦略は?

ヒント：相手がこの戦略を用いていると，何を出しても期待利得は0



クイズ (解答)

- グーとチョキの確率は同じで2/5，パーの確率1/5がナッシュ均衡

クイズ：蚤殿じゃんけん

- 通常のじゃんけんに，新しい2つの手（蚤と殿）を加える
- 蚤はグー，チョキ，パーのいずれにも負けるが，殿には勝つ
- 殿はグー，チョキ，パーのいずれにも勝つが，蚤に負ける
- ナッシュ均衡（絶対に負けない戦略）は?

乱択の効果

- 直球が得意なバッター
- 直球に山を張って当たると8割打てる．外れると全然打てない
- 変化球に山を張って当たると3割打てる．外れても（直球なら）1割は打てる
- 最悪の場合を避けるなら，変化球に山を張るべきだが，最悪1割しか打てない

バッター＼ピッチャー	直	変
直	8	0
変	1	3

301

乱択を使うと?

- バッターはどちらに山を張るかを場合によって変化
- ピッチャーも適当に直球と変化球を混ぜる
- このような戦略を混合戦略と呼ぶ
- バッターが2割直球を待つと，ピッチャーがどんな割合で投げても打率は一定で 2割4分
- 得意技／伝家の宝刀は，たまに使う方が効果がある

	直	変
直	8	0
変	1	3

302

（単純化）ポーカー

- 以下のゲームを考える
- 後手はKingを持っている
- 先手にはAce, Queenの二枚から一枚がランダムに割り当てられる
- 先手は，1万円賭けるか，2万円賭けるかを選ぶ
- 先手が1万円賭けた場合は，直ちに勝負．カードが強い方が勝ち
- 先手が2万円賭けた場合，後手は1万円払って降りるか，2万円で勝負するかを選べる
- 互いに，どのようにプレイするのが最適?
- カードの強さだけなら，平均的には公平なはず
- 期待利得は0?

303

単純化ポーカーの戦略

- 先手はAceを引けば確実に勝てる
- Aなのに2万円賭けないのは明らかに損．
- 先手はQの時に，いくら賭けるべき? 予想：負けるのは分かっているから，1万にすべき?
- 後手は，先手が2万賭けた時に，勝負するか降りるかが問題
  - 予想：多分，先手はAを持っているから降りた方が無難?
- どちらの予想も誤り!
- 先手は，Qの時，確率1/3で2万賭ける，後手は，先手が2万賭けたら，確率1/3で降りる
- 先手の期待効用は1/3

	降	勝負
常に2	1	0
A: 2 Q: 1	0	1/2

304

8. 線形時間のソーティング

- 今までのソートアルゴリズム
  - 入力要素の比較のみに基づく
  - 比較ソート (comparison sort) と呼ぶ
  - 例：マージソート，ヒープソート，クイックソート
  - （最悪時の）計算量：$\Omega(n \lg n)$ --- 下界値
- この節でのソートアルゴリズム
  - 比較以外の演算を用いる
  - 例：計数ソート，基数ソート，バケットソート
  - 計算量：$O(n)$ (線形時間)

305

8.1 ソーティングの下界

- ソーティングの入力：$\langle a_1, a_2, \ldots, a_n \rangle$
- 比較ソートでは要素間の比較のみを用いてソートを行う
- 2つの要素 a_i, a_j が与えられたとき，それらの相対的な順序を決定するためにテストを行う
  - $a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, $a_i > a_j$ のみ
- これ以外の方法では要素のテストはできないと仮定する

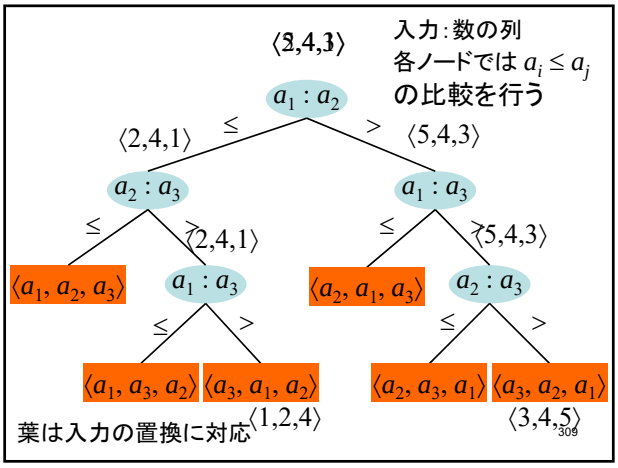
306

仮定

- すべての入力要素は異なると仮定する
  - $a_i = a_j$ という比較は行わないと仮定できる
- $a_i < a_j$, $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ は全て等価
- 全ての比較は $a_i \leq a_j$ という形と仮定できる

決定木モデル

- 比較ソートは決定木 (decision tree) とみなせる
- 決定木はソーティングアルゴリズムで実行される比較を表現している
- アルゴリズム中における制御，データの移動などの要因は無視する



決定木の高さと比較回数

- 決定木はソートアルゴリズム A から決まる
- 入力数列を与えると決定木の対応する葉が決まる
- 根から葉までのパスの長さ
  ＝Aを実行する際の比較回数
- 根から葉までのパス長の最大値
  ＝実行されるソートアルゴリズムの最悪比較回数
- 比較ソートでの最悪の比較回数は決定木の高さに対応

最悪時の下界

- 決定木の高さの下界＝任意の比較ソートアルゴリズムの実行時間の下界

定理 8.1 n 要素をソートするどんな決定木の高さも $\Omega(n \lg n)$

証明 n 要素をソートする高さ h の決定木を考える．

ソートを正しく行うには，n 要素の $n!$ 通りの置換全てが葉に現れなければならない．

高さ h の2分木の葉の数は高々 2^h．よって

$n! > 2^h$ ではソートできない．つまり $h \geq \lg(n!)$

$$h \geq \lg(n!)$$

Stirlingの公式より

$$n! > \left(\frac{n}{e}\right)^n$$

$$h \geq \lg\left(\frac{n}{e}\right)^n$$

$$= n\lg n - n\lg e$$

$$= \Omega(n\lg n)$$

系 8.2　ヒープソートとマージソートは漸近的に最適な比較ソートである

証明　これらの実行時間の上界 O(n lg n) は定理 8.1 の最悪時の下界 Ω(n lg n) と一致する.

スパゲティソート

- n個の数をソートする
- 各要素に対応するスパゲティを一本選び，数に対応する長さに切る
- スパゲティの束の片方を机に押し当てると，最も飛び出しているのが最大の要素
- 最大の要素を取り除き，以下繰り返し
- O(n)
- 一回の操作で最大値が得られる＝比較以外の操作が使える
- アナログ計算機，分子計算，量子計算

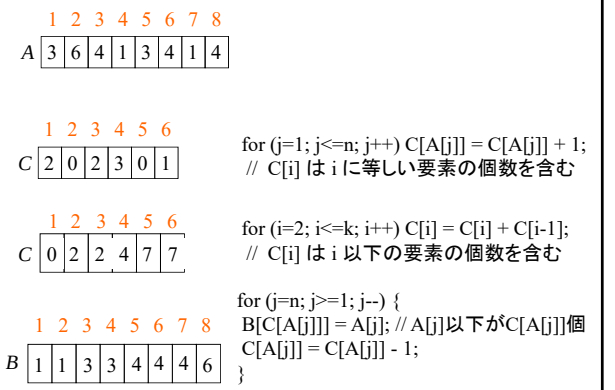
8.2 計数ソート (counting sort)

- n 個の各入力要素が 1 から k の範囲の整数であると仮定 (k: ある整数)
- k = O(n) のとき，計数ソートは O(n) 時間
- 基本的なアイデア
  - 各入力要素 x に対し，x より小さい要素の数を決定
  - その要素数から x の出力配列での位置が決まる
  - 例：x より小さい数が17個 ⇒ x の出力位置は18
  - 複数の要素が同じ値を持つ場合に対応する必要あり

計数ソートのプログラム

- A[1..n], B[1..n]: 入出力配列
- C[1..k]: 補助的な配列

```
void COUNTING_SORT(int A[], int B[], int C[], int n, int k){
  int i,j;
  for (i=1; i<=k; i++) C[i] = 0;
  for (j=1; j<=n; j++) C[A[j]] = C[A[j]] + 1;
  //  C[i] は i に等しい要素の個数を含む

  for (i=2; i<=k; i++) C[i] = C[i] + C[i-1];
  //  C[i] は i 以下の要素の個数を含む

  for (j=n; j>=1; j--) {
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]] - 1;}}
```

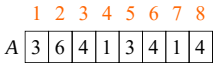



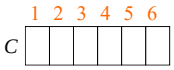

```
for (j=1; j<=n; j++) C[A[j]] = C[A[j]] + 1;
 //  C[i] は i に等しい要素の個数を含む

for (i=2; i<=k; i++) C[i] = C[i] + C[i-1];
 //  C[i] は i 以下の要素の個数を含む

for (j=n; j>=1; j--) {
 B[C[A[j]]] = A[j]; // A[j]以下がC[A[j]]個
 C[A[j]] = C[A[j]] - 1;
}
```


課題

- 以下のアルゴリズムは正しいか?

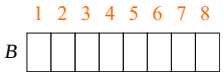
```
void COUNTING_SORT(int A[], int B[], int C[], int n, int k){
  int i,j;
  for (i=1; i<=k; i++) C[i] = 0;
  for (j=1; j<=n; j++) C[A[j]] = C[A[j]] + 1;
  //  C[i] は i に等しい要素の個数を含む

  for (i=2; i<=k; i++) C[i] = C[i] + C[i-1];
  //  C[i] は i 以下の要素の個数を含む

  for (j=1; j<=n; j++) { --- 後ろからではなく，前から並べる
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]] - 1;}}
```



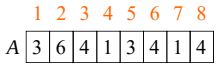


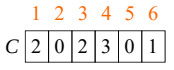

```
for (j=1; j<=n; j++) C[A[j]] = C[A[j]] + 1;
 // C[i] は i に等しい要素の個数を含む
```


```
for (i=2; i<=k; i++) C[i] = C[i] + C[i-1];
 // C[i] は i 以下の要素の個数を含む
```

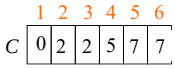



```
for (j=1; j<=n; j++) {
--- 後ろからではなく，前から並べる
 B[C[A[j]]] = A[j]; // A[j]以下がC[A[j]]個
 C[A[j]] = C[A[j]] - 1;
}
```

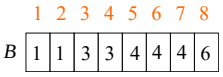






```
for (j=1; j<=n; j++) C[A[j]] = C[A[j]] + 1;
 // C[i] は i に等しい要素の個数を含む
```



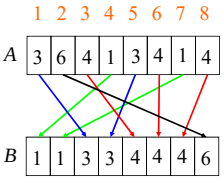

```
for (i=2; i<=k; i++) C[i] = C[i] + C[i-1];
 // C[i] は i 以下の要素の個数を含む
```




```
for (j=1; j<=n; j++) {
 B[C[A[j]]] = A[j]; // A[j]以下がC[A[j]]個
 C[A[j]] = C[A[j]] - 1;
}
```


安定なソート

- 同一の値は入力と出力で同じ順序になる
- 付属データをキーに従ってソートする場合に重要
- 計数ソートは安定なソートになっている



課題の答え

- ソーティング自体は正しく動く
- ただし安定でない（同じ値が元の順序と入れ替わってしまう）
- 元のアルゴリズムは後ろから配置することが安定にするためのポイント

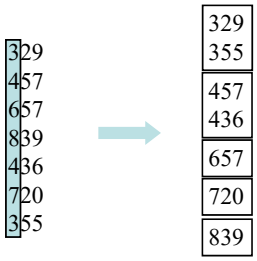
計数ソートの計算時間

- C の初期化: $O(k)$ 時間
- 頻度の計算: $O(n)$ 時間
- 累積頻度の計算: $O(k)$ 時間
- 出力配列の計算: $O(n)$ 時間

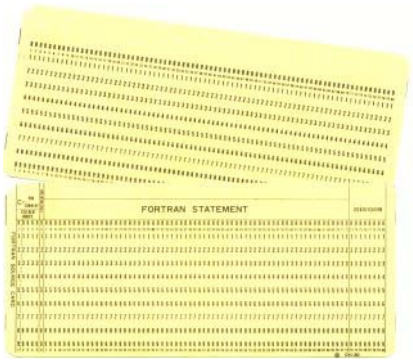
- 全体で $O(k + n)$ 時間
- $k = O(n)$ のとき，全体で $O(n)$ 時間
- 比較ソートの下限 $\Omega(n \lg n)$ を下回る

8.3 基数ソート (radix sort)

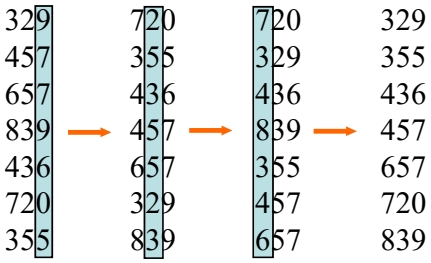
- 数の集合をある桁に従って分割する機械がある
- この機械を用いて d 桁の数 n 個をソートしたい



パンチカード



- まず最上位桁でソート (分割) し，それぞれを再帰的にソートすることを考える
- 大量の部分問題が生成される
  - 大量の作業領域が必要
- 解：最下位桁からソートする⇒基数ソート
  - まず最下位桁に従ってソート
  - 次に下から2桁目に従ってソート
  - *d* 桁目まで繰り返す



基数ソートでは各桁のソートは安定であることが要求される

基数ソートの擬似コード

RADIX_SORT(A,d)

1. for (i=1; i<=d; i++)
2. 安定ソートを用いて i 桁目に関して配列をソート

基数ソートの実行時間

- 各桁が 1 から *k* の範囲として計数ソートを使用
- *d* パスあるので $\Theta(d(n+k))$ 時間
- *d* が定数で $k = O(n)$ ならば，全体で $O(n)$ 時間

9. 中央値と順序統計量

- *n* 要素の集合の *i* 番目の順序統計量 (order statistic): その集合で *i* 番目に小さい要素
- 最小値 (minimum): $i = 1$
- 最大値 (maximum): $i = n$
- 中央値 (median):
  - *n* が奇数のとき $i = (n+1)/2$
  - *n* が偶数のとき $i = \lfloor (n+1)/2 \rfloor$ と $i = \lceil (n+1)/2 \rceil$ に統一

選択問題 (selection problem)

- 入力: n 個の異なる数の集合 A, 整数 i ($1 \le i \le n$)
- 出力: A 中のちょうど $i-1$ 個の他の要素より大きい要素 $x \in A$
- 選択問題は O(n lg n) 時間で解ける
  - A の要素をソートする
  - 出力配列の i 番目の要素を返す
- 選択問題はソートを使わずに O(n) 時間で解ける

331

9.1 最大と最小

- n 要素の集合の最小値を決定するために必要な比較回数を考える
- 上限: $n-1$ 回

```
data MINUMUM(int n, int A[])
{
   int i;
   int MIN;
   MIN = A[1];
   for (i=2; i<=n; i++) {
      if (A[i] < MIN) MIN = A[i];
   }
   return MIN;
}
```

332

比較回数の下限

命題: $n-2$ 回の比較では最小値は決定できない

- 要素間のトーナメントによって最小値を決定する
- 1回の比較はトーナメントの試合に相当する
- 値の小さいほうが試合に勝つ
- 最小値 (優勝者) が決まる⇔それ以外が負ける
- $n-1$ 個が負けるには $n-1$ 試合必要

アルゴリズムMINIMUMは比較回数の点で最適

333

9.2 平均線形時間の選択法

- 順序統計量は $\Theta(n)$ 時間で求まるが複雑
- まず，平均的に O(n) 時間のアルゴリズムを考える
- アイデア: クイックソートと同様にPARTITIONを使うが，順序統計量を求めるのに不要な部分のソートはサボる

```
data RANDOMIZED_SELECT(int A[], int p, int r, int i)
// A[p..r] の中で i 番目に小さい要素を返す
{
   int q, k;
   if (p == r) return A[p];
   q = RANDOMIZED_PARTITION(A,p,r); // q: 分割の位置
   k = q – p + 1;                   // k: 左部分の要素数
   if (i <= k) return RANDOMIZED_SELECT(A,p,q,i);
   else        return RANDOMIZED_SELECT(A,q+1,r,i–k);
}
```

334

- q = RANDOMIZED_PARTITION(A,p,r);の実行後
  - 2つの空でない部分配列 $A[p..q]$, $A[q+1..r]$ に分かれる
  - 左側の各要素は右側よりも小さい
  - $k = q - p + 1$; は左側の要素数
- $i \le k$ ならば i 番目の要素は左側の i 番目
- $i > k$ ならば i 番目の要素は右側の $i - k$ 番目
- 部分配列のサイズが 1 になるまで繰り返す

335

平均実行時間の上界

- 詳細は省略，基本的には，運の悪い分割 (i番目が含まれている方にn-1個の要素がある) ことが確率1/nでしか生じないことから導かれる

336