アルゴリズムとデータ構造
第10回: ソーティング(2)

担当:上原隆平 (uehara)

2015/05/15

counting sort

計数ソート

計数ソート (counting sort)

- 仮定: data[i]∈{1,…,k} for 1≦i≦n, k∈O(n)
- Θ(*n*)時間でソーティングを行うアルゴリズム
- 考え方: 要素xの位置を決める
  - xより小さい要素の個数がわかる➔
    その個数で示される場所にxを置く

3	7	4	1	2	5

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

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

1	2	3	4	5	7

計数ソート (counting sort)

- Q. 同じ値の要素がある時は?
- A. 3つの配列 a[], b[], c[]を利用
  (a: 入力データ，b: ソートされた列，c: カウンタ)
  - 値 a[i] を持つデータの個数を c[a[i]] でカウント
  - 0≦j≦k なる各 j について
    c’[j] := c[0] + … + c[j-1] + c[j] とすると
    c’[j]は値がj以下の入力データの個数
  - c’[]に従って各要素a[i]を配列b[]の正しい場所に順に蓄える

計数ソート: プログラム

```
CountingSort(a, b, k){
  for i=0 to k
     c[i] = 0;

  for j=0 to n-1
     c[ a[j] ] = c[ a[j] ] + 1;

  for i=1 to k
     c[i] = c[i] + c[i-1];

  for j=n-1 downto 0
    b[ c[a[j]]-1 ] = a[j];
    c[a[j]] = c[a[j]] - 1;
}
```

カウンタの値を初期化

値a[i]の個数をカウント

c[i]=i以下の値の個数

ソート列の格納

計数ソート: 0から6までの整数 (3,6,4,1,3,4,1,4)のソート

- ②を実行して
  c[]=(0,2,0,2,3,0,1)
- ③を実行して
  c[]=(0,2,2,4,7,7,8)

a[7]=4 => b[c[4]-1] = b[6], c[4]=6
a[6]=1 => b[c[1]-1] = b[1], c[1]=1
a[5]=4 => b[c[4]-1] = b[5], c[4]=5
a[4]=3 => b[c[3]-1] = b[3], c[3]=3
a[3]=1 => b[c[1]-1] = b[0], c[1]=0
a[2]=4 => b[c[4]-1] = b[4], c[4]=4
a[1]=6 => b[c[6]-1] = b[7], c[6]=7
a[0]=3 => b[c[3]-1] = b[2], c[3]=2

```
CountingSort(a, b, k){
  for i=0 to k
     c[i] = 0;

②for j=0 to n-1
     c[ a[j] ] = c[ a[j] ] + 1;

③for i=1 to k
     c[i] = c[i] + c[i-1];

  for j=n-1 to downto 0
    b[ c[a[j]]-1 ] = a[j];
    c[a[j]] = c[a[j]] - 1;
}
```

計数ソート: 0から6までの整数

値が同じ時ソート後の要素
の順序は元の順序を保存
➔ 安定であるという

- ②
  c[]
- ③を実行し
  c[]=(0,2,2,4,7,7,8)

a[7]=4 => b[c[4]-1] = b[6], c[4]=6
a[6]=1 => b[c[1]-1] = b[1], c[1]=1
a[5]=4 => b[c[4]-1] = b[5], c[4]=5
a[4]=3 => b[c[3]-1] = b[3], c[3]=3
a[3]=1 => b[c[1]-1] = b[0], c[1]=0
a[2]=4 => b[c[4]-1] = b[4], c[4]=4
a[1]=6 => b[c[6]-1] = b[7], c[6]=7
a[0]=3 => b[c[3]-1] = b[2], c[3]=2

```
②      to n-1
       c[ a[j] ] = c[ a[j] ] + 1;

③ for i=1 to k
       c[i] = c[i] + c[i-1];

   for j=n-1 to downto 0
     b[ c[a[j]]-1 ] = a[j];
     c[a[j]] = c[a[j]] - 1;
}
```

radix sort

基数ソート

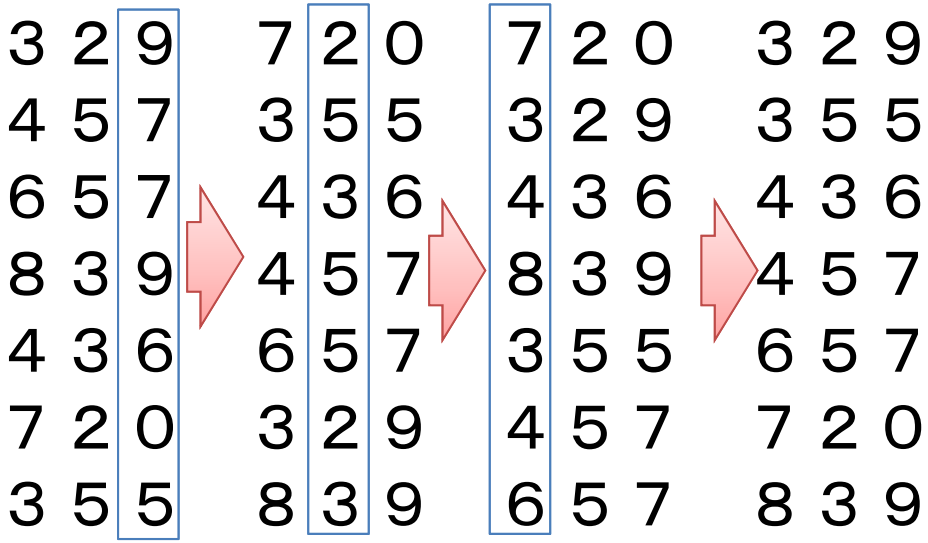
基数ソート (radix sort)

- 入力データがd桁の10進数の場合、
  各桁ごとにソートするならどちらがよいか
  - 上位桁から順にソートする
  - 下位桁から順にソートする

- A. 下位桁からソートする

基数ソート (radix sort)

- 各桁ごとにソートするアルゴリズム



  - 各桁のソートには計数ソートを用いる
    - 計算量は？

これをどかすと答えがあります

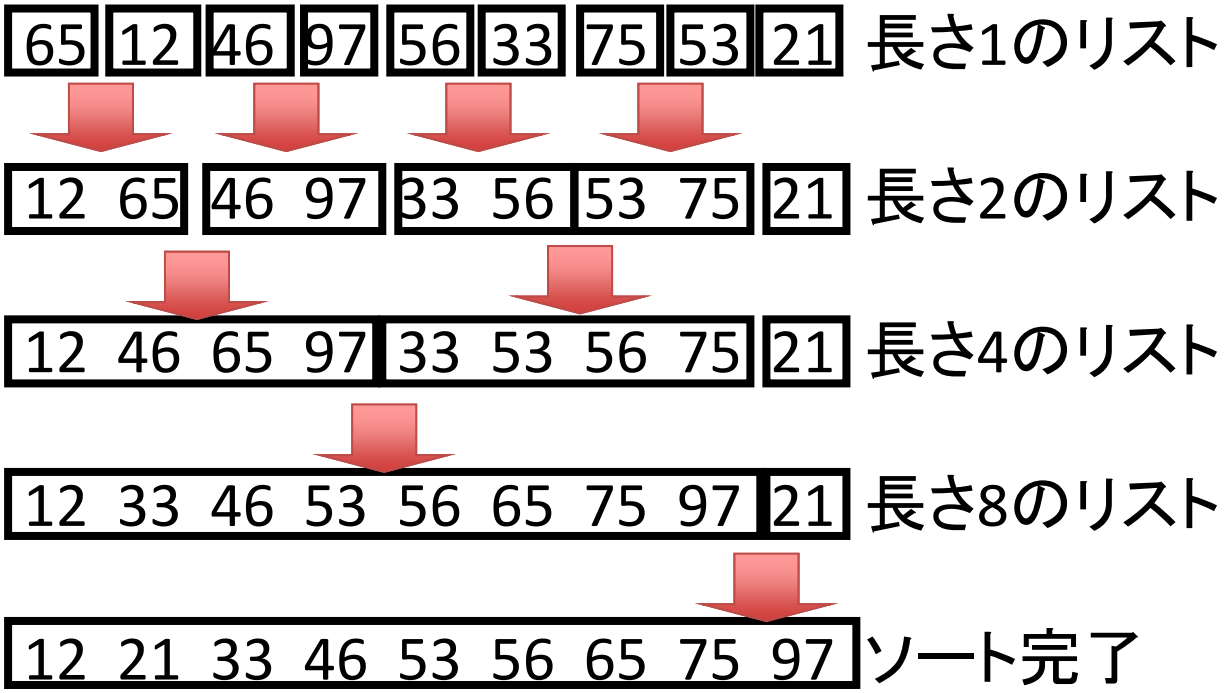
John von Neumann
1903−1957

merge sort

マージソート

マージソート (merge sort)

- 2個のソート済みの列をマージして1個のソート済みの列を得るアルゴリズム



- ソートするべき列の長さが1になるまで2分して2個ずつをマージするとソートができる

マージソートの実現: 再帰呼出の利用

- ソートすべき区間: [left, right]
- 中央 mid = (left + right)/2

- [left,right]➔[left,mid], [mid+1,right]
- それぞれの部分区間にマージソートを適用してソート済みの列を得、それらをマージする

マージソート: プログラムの概観

```
MergeSort(int left, int right){
    int mid;
    if(区間[left,right]が十分短い)
     他の方法でソート
    else{
      mid = (left+right)/2;
      MergeSort(left, mid);
      MergeSort(mid+1, right);
      [left, mid]と[mid+1, right]をマージする
    }
}
```

長さpとqの列のマージは$O(p+q)$で可能

マージソート: プログラムの概観 – マージ

[left, mid] と [mid+1, right] をマージする

$O(p+q)$

```
i=left; j=mid+1; k=left;
while(i<=mid && j<=right)
  if(a[i] <= a[j]) {
    b[k]=a[i]; k++; i++:
  } else {
    b[k]=a[j]; k++; j++;
  }
while(j<=right){ b[k]=a[j]; k++; j++; }
while(i<=mid){ b[k]=a[i]; k++; i++; }
for(i=left; i<=right; i++) a[i]=b[i];
```

一時的にソート済みの列をb[]に蓄える

a[]にb[]を書き戻す

マージソート: 計算時間の解析

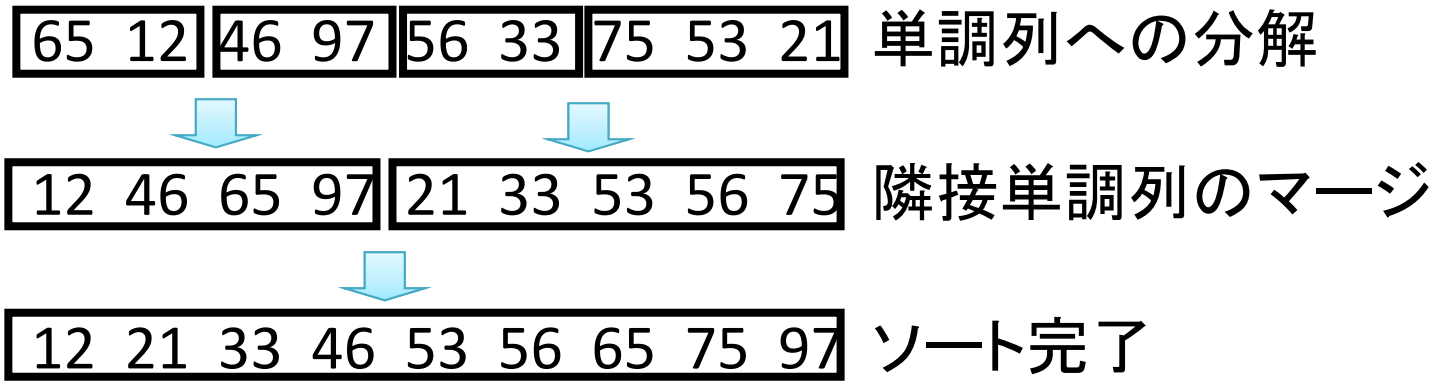
- $T(n)$: n データのマージソートに要する時間
  - $T(n) = 2T(n/2)$ + マージにかかる時間
    $= 2T(n/2) + cn + d$　(c, dは定数)
- 簡単のため $n = 2^k$ とすると

$$
\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + c2^k + d \\
&= 2(2T(2^{k-2}) + c2^{k-1} + d) + c2^k + d \\
&= 2^2T(2^{k-2}) + 2c2^k + (1+2)d \\
&= 2^2(2T(2^{k-3}) + c2^{k-2} + d) + 2c2^k + (1+2)d \\
&= 2^3T(2^{k-3}) + 3c2^k + (1+2+4)d \\
&\vdots \\
&= 2^iT(2^{k-i}) + ic2^k + (1+2+\ldots 2^{i-1})d \\
&= 2^kT(2^0) + kc2^k + (1+2+\ldots 2^{k-1})d \\
&= bn + cn\log n + (n-1)d \in O(n\log n)
\end{aligned}
$$

単調列マージソート

- 入力列を単調列に分割し、隣接列をマージ
  - cf., マージソートでは入力列に関係なく一定の長さの区間に分割した後隣接列をマージ

- 例題: 65 12 46 97 56 33 75 53 21 のソート



単調列マージソート: 計算時間

- 長さ p と q の 2単調列は O(p+q)でマージ可能
- 隣接する単調列をマージすると
  単調列の個数は約半分になる
  - 最初の単調列の個数を h とすると,
    $\log_2 h$ 回の再帰でソートが完了
- 一回の再帰で O(n)時間 → 全体で O(n log h)

- ソート済みのデータが入力された場合: $h = 1$
- 単調列の個数の最大値は n/2

The computational complexity of the sorting problem

ソート問題の計算複雑度

比較ソート

- データの大小関係のみを用いてソートするアルゴリズムを 比較ソート と呼ぶ
  - a > b, a = b, a < b という性質のみ用い，a や b がどんな数であるかは用いない

比較ソート問題の計算複雑度

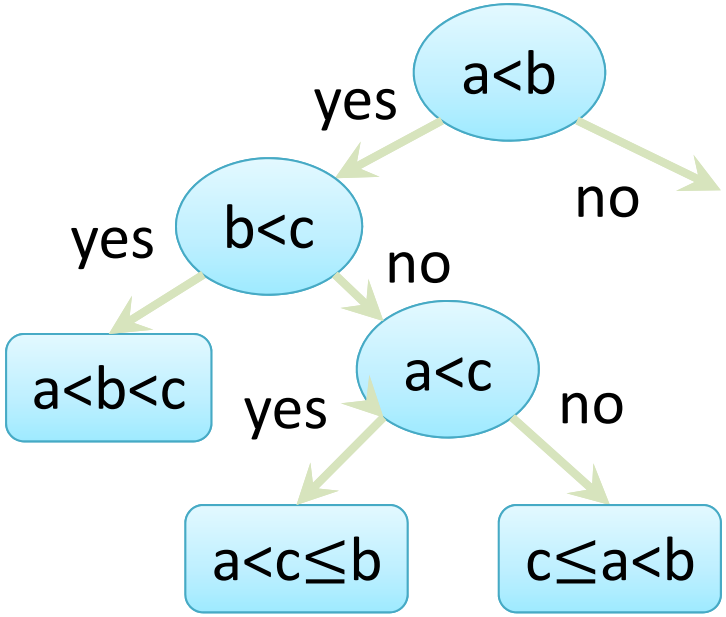
- 上界: $O(n \log n)$
  最悪の場合でも $n \log n$ に比例する時間で
  ソートする比較ソートアルゴリズムが存在
- 下界: $\Omega(n \log n)$
  どんな比較ソートアルゴリズムでも最悪時に
  $n \log n$ に比例する計算時間がかかってしまう
  入力例が存在する

ソーティングを「入力データ上での比較を繰り返して
最後に正しい順序で出力する」こととして下界を考える

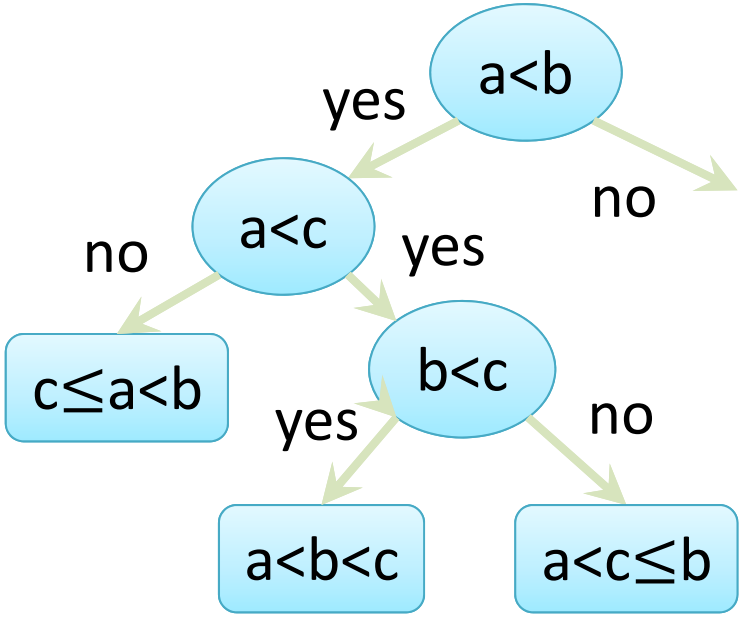
比較ソート問題の計算複雑度: 下界

- 3個のデータa, b, cをソートする場合:
  最初に(a,b), (b,c), or (c, a)を比較する
  - 先に (a,b) を比較した場合は2回めは (b,c) or (c,a)

★b<cを比較



★a<cを比較



比較ソート問題の計算複雑度: 下界

- {a, b, c}の整列問題からわかること
  - どのような入力例に対しても高々三回以内の比較で答えを得る
  - 少なくとも三回の比較を行わないと答えを得られない場合がある

    =

    根から葉に至るパスの長さの最大値が3 (下界)

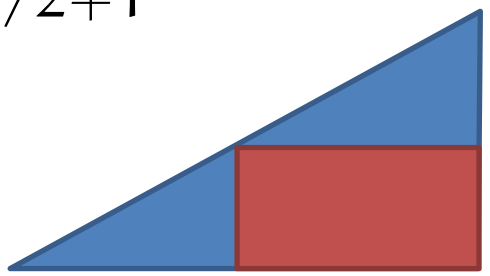
根から葉に至るパスの長さの最大値が最小になるように決定木を構成したとき，最長パスの長さが問題の下界を与える

比較ソート問題の計算複雑度: 下界

- n 個のデータをソートする場合
  - 最適な決定木の最長パスの長さを k をすると，この決定木の葉の個数 $\leqq 2^k$
  - n 個の要素の順列が全て決定木の葉として現れないとならないから，$n! \leqq 2^k$
  - 両辺の対数をとると，

$$k = \lg 2^k \geq \lg n! = \sum_{i=1}^{n} \lg i \geq \sum_{i=n/2+1}^{n} \lg \frac{n}{2}$$

$$= \frac{n}{2} \lg \frac{n}{2} \in \Omega(n \log n)$$



ミニ演習

- 前回・今回と学んだソートアルゴリズムのうち，
  - 安定なソートはどれか？
  - 比較ソートでないものはどれか？

- 証明せよ: $\frac{n}{2} \lg \frac{n}{2} \in \Theta(n \log n)$

  - 定義: $\Theta(f(n)) = \{g(n) \mid \exists c_1, c_2 > 0, \exists n_0, \forall n \geq n_0, c_1 f(n) \leq g(n) \leq c_2 f(n)\}$