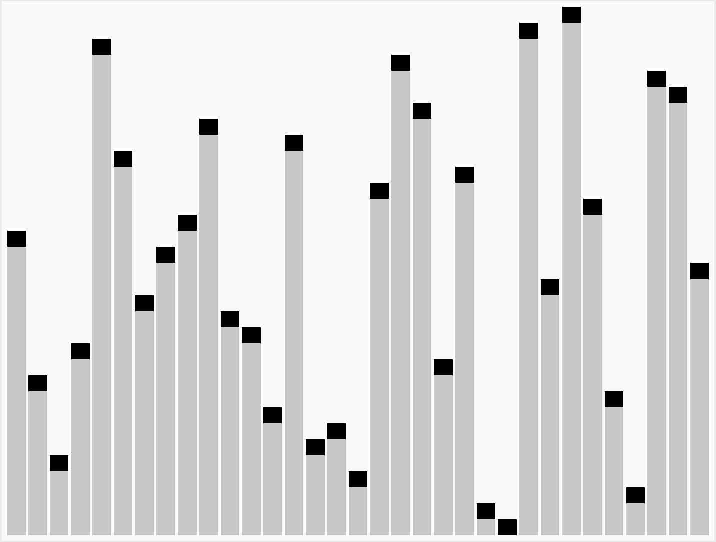
Introduction to Algorithms and Data Structures

Lecture 12: Sorting (3)
Quick sort, complexity of sort algorithms, and counting sort

Professor Ryuhei Uehara,
School of Information Science, JAIST, Japan.
uehara@jaist.ac.jp
http://www.jaist.ac.jp/~uehara

Tony Hoare
1934–

QUICK SORT



C.A.R. Hoare, “Algorithm 64: Quicksort”.
Communications of the ACM 4 (7): 321 (1961)

Quick sort

- Main property: On average, the fastest sort!
- Outline of quick sort:
  - Step 1: Choose an element x (which is called pivot)
  - Step 2: Move all elements ≦ x to left
    Move all elements ≧ x to right

≦x

≧x

  - Step 3: Sort left and right sequences <u>independently</u> and <u>recursively</u>
    - (When sequence is short enough, sort by any simple sorting)

Quick sort: Example

Step 1. Choose an element x

- Sort the following array by quick sort:

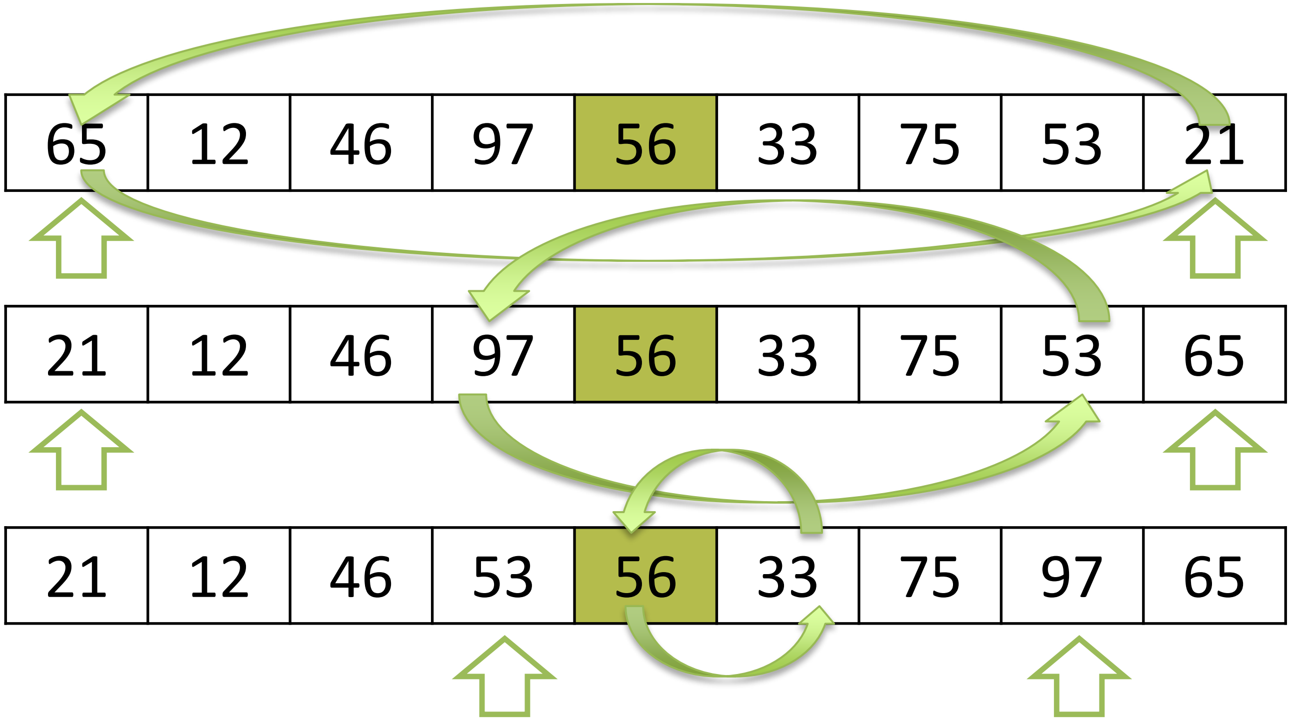
65	12	46	97	56	33	75	53	21

- Choose x=56, for example;

65	12	46	97	56	33	75	53	21

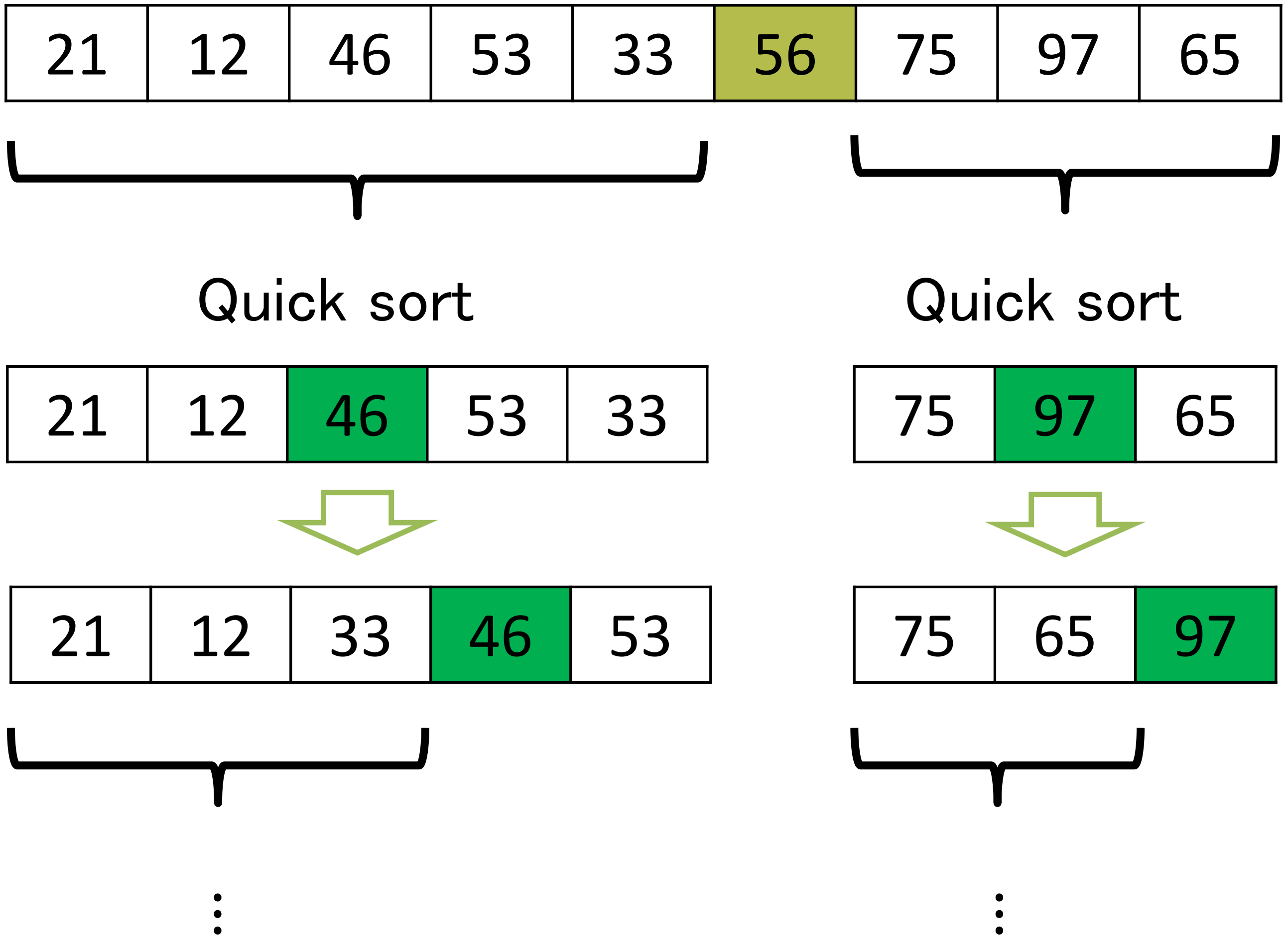
Quick sort: Example
Step 2. Move element w.r.t x:

- 
  ≦x ≧x
- Start from [l, r] = [0,n-1], move l and r,
  Swap a[l] and a[r] when a[l] >= x && a[r] < x



65	12	46	97	56	33	75	53	21
21	12	46	97	56	33	75	53	65
21	12	46	53	56	33	75	97	65

Quick sort: Example
Step 3. Sort left and right sequences recursively



21	12	46	53	33	56	75	97	65

Quick sort

21	12	46	53	33

21	12	33	46	53

⋮

Quick sort

75	97	65

75	65	97

⋮

Quick sort: Program

```
qsort(int a[], int left, int right){
  int i, j, x;
  if(right <= left) return;
  i = left; j = right; x = a[(i+j)/2];
  while(i<=j){
    while(a[i]<x) i=i+1;
      while(a[j]>x) j=j-1;
       if(i<=j){
      swap(&a[i], &a[j]); i=i+1; j=j-1;
    }
  }
  qsort(a, left, j); qsort(a, i, right);
}
```

Note: In MIT textbook, there is another implementation.

Report Problem 4

In page 2, we consider the following case;

- String data: lexicographical ordering
  e.g., aaa, aab, aba, abb, baa, bab, bbc, bcb

For any two binary strings s=s[1]…s[n] and t=t[1]…t[m], describe exact condition if and only if s<t (note that n≠m in general).

(Bonus; can you make a dictionary that has all binary strings in your lexicographical ordering, and any finite length word has finite index? How can you avoid the potential problem?)

A part of final report

- For the qsort, construct a bad input that gives the worst case.

Quick sort: Time complexity (1/~~3~~)

6

Worst case

- When the pivot x is the maximum or minimum element, we divide
  length n → length 1 + length n-1
- This repeats until the longer one becomes 2
- The number of comparisons; $\sum_{k=2}^{n} k \in \Theta(n^2)$

Almost as same as the bubble sort…

Analysis of QuickSort

– Sorting Problem

Input: An array a[n] of n data

Output: The array a[n] such that

a[1]<a[2]<…<a[n]

★To simplify, we assume that there are no pair i≠j with a[i]=a[j]

– In practical, QuickSort is said to be "the fastest sort"

- Representative algorithm based on divide-and-conquer
- If partition is well-done, it runs in $O(n \log n)$ time.
- If each partition is the worst case, it runs in $O(n^2)$ time.

…Can we analyze theoretically, and guarantee the running time?

Analysis of QuickSort

- Review of QuickSort
  - Call qsort(a,1,n)
  - If qsort(a, i, j) is called,
    - (Randomly) choose a pivot a[m]
    - Divide a[] into “former” and “latter” by a[m]. I.e., sort as
      a[i’]<a[m] for i≦ i’ < m, and
      a[j’]>a[m] for m< j’< j.
    - Return qsort(a, i, i’), a[m], qsort(a, j’, j) as the result

[C.F.]
We can always find the center in $O(j\text{-}i)$ time.

- Though they say that QuickSort is the fastest in a practical sense,,,
  - When a[m] is always the center of a[i]..a[j], we have
    $T(n) \leqq 2T(n/2) + (c+1)\, n$
    and hence $T(n) = O(n \log n)$.
  - When a[m] is always either a[i] or a[j], we have
    $T(n)\leqq T(1)+T(n\text{-}1)+(c+1)n$
    and hence $T(n) = O(n^2)$.

What about average case?

H_n is the harmonic number and H_n=O(log n) .

Analysis of QuickSort

- They say that QuickSort is the fastest in a practical sense,,,
  - Assumption: each item in a[i] ... a[j] is chosen uniformly at random.
    - Thus the kth largest value is chosen as the pivot with probability $1/(j\text{-}i+1)$

[Theorem] An upper bound of the expected value of the running time of QuickSort is $2n\,H(n)\sim 2n \log n$

It runs fast since few overhead.

- Notation
  - » s_k is the kth largest item in a[1]...a[n].
  - » Define indicator variable X_{ij} as follows

$$X_{ij} = \begin{cases} 0 & s_i \text{ and } s_j \text{ are not compared in the algorithm} \\ 1 & s_i \text{ and } s_j \text{ are compared in the algorithm} \end{cases}$$

- Running time of QuickSort

~ the number of comparisons= $\sum_{i=1}^{n}\sum_{j>i} X_{ij}$

Analysis of QuickSort

[Theorem] An upper bound of the expected value of the running time of QuickSort is $2n\,H(n) \sim 2n \log n$

- The expected value of the running time of QuickSort=

$$E[\sum_{i=1}^{n}\sum_{j>i} X_{ij}] = \sum_{i=1}^{n}\sum_{j>i} E[X_{ij}]$$

（Linearity of expectation value）

- Define as “p_{ij} : probability that s_i and s_j are compared”,

$$E[X_{ij}] = p_{ij} \times 1 + (1 - p_{ij}) \times 0 = p_{ij}$$

  Thus consider the value of p_{ij}

- When s_i and s_j are compared??
  1. One of them is chosen as the pivot, and
  2. They are not yet separated by qsort up to there

     ⇔ Any element between s_i and s_j are not yet chosen as a pivot

Analysis of QuickSort

[Theorem] An upper bound of the expected value of the running time of QuickSort is $2n\,H(n) \sim 2n\log n$

- When s_i and s_j are compared?
  1. One of them is chosen as the pivot, and
  2. They are not yet separated by qsort up to there

     ⇔ Any element between s_i and s_j is not yet chosen as a pivot
  - The ordering of pivots in $s_i, s_{i+1}, s_{i+2}, \ldots, s_{j-1}, s_j$ is uniformly at random!
  - Thus s_i or s_j is the first pivot with probability $\dfrac{2}{j-i+1}$

Therefore, the expected time of the running time of QuickSort

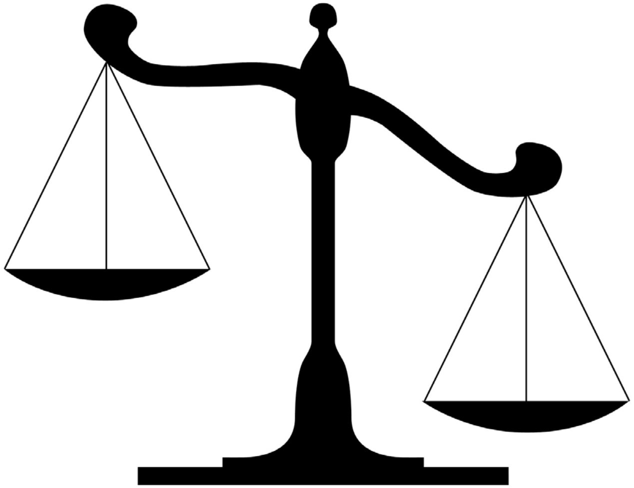
$$= E[\sum_{i=1}^{n}\sum_{j>i} X_{ij}] = \sum_{i=1}^{n}\sum_{j>i} E[X_{ij}] = \sum_{i=1}^{n}\sum_{j>i} p_{ij} = \sum_{i=1}^{n}\sum_{j>i}\frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n}\sum_{k=2}^{n-i+1}\frac{2}{k} \le 2\sum_{i=1}^{n}\sum_{k=1}^{n}\frac{1}{k} = 2nH(n)$$

COMPUTATIONAL COMPLEXITY OF THE SORTING PROBLEM

Sort on Comparison model

- Sort on comparison model: Sorting algorithms that only use the "ordering" of data
  - It only uses the property of "a > b, a = b, or a < b"; in other words, the value of variable is not used.



Computational complexity of sort on comparison model

- Upper bound: O(*n* log *n*)
  There exist sort algorithms that run in time proportional to *n* log *n* (e.g., merge sort, heap sort, …).
- Lower bound: Ω(*n* log *n*)
  For any comparison sort, there exists an input such that the algorithm runs in time proportional to *n* log *n.*

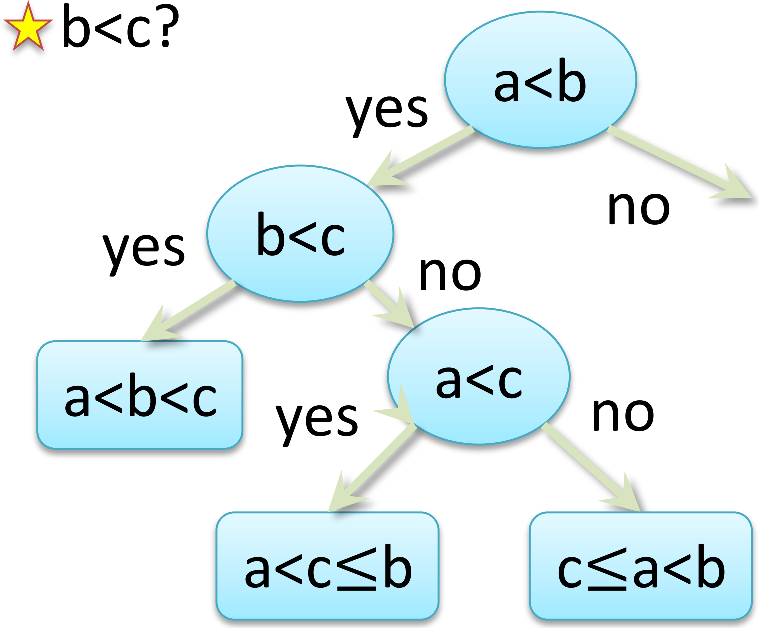
We consider the lower bound of comparison sorting.

Computational complexity of comparison sort: lower bound

- Simple example; sort 3 data a, b, c:

  First, compare (a,b), (b,c), or (c, a). Without loss of generality, we assume that (a,b) is compared; then the next pair is (b,c) or (c,a):

★$b<c$?



$a<b$

yes

no

$b<c$

yes

no

$a<b<c$

$a<c$

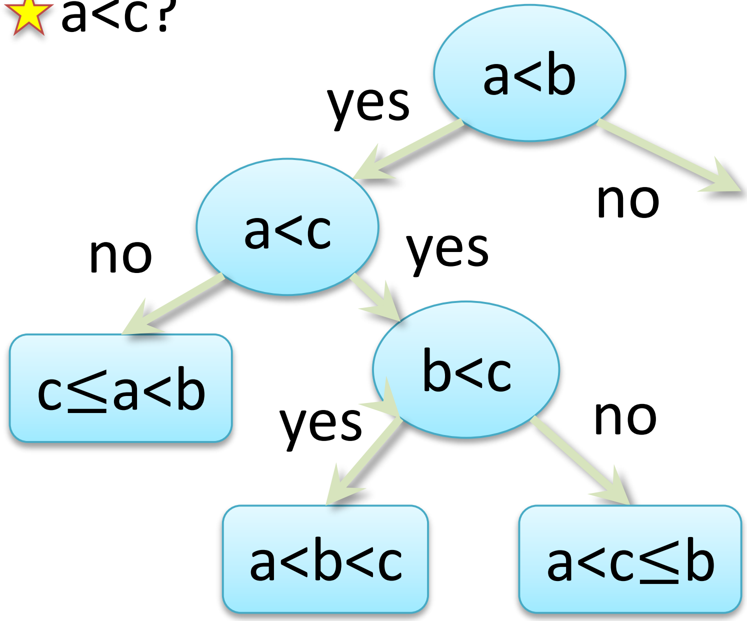
yes

no

$a<c\leq b$

$c\leq a<b$

★$a<c$?



$a<b$

yes

no

$a<c$

no

yes

$c\leq a<b$

$b<c$

yes

no

$a<b<c$

$a<c\leq b$

Computational complexity of comparison sort: lower bound

- What we know from sorting of {a, b, c}:
  - For any input, we obtain the solution at most 3 comparison operators.
  - There are some input that we have to compare at least 3 comparison operations.

= maximum length of a path from root to a leaf is 3, which gives us the lower bound.

When we build a decision tree such that "the longest path from root to a leaf is shortest," that length of the longest path gives us a lower bound of sorting problem.

Computational complexity of comparison sort: lower bound

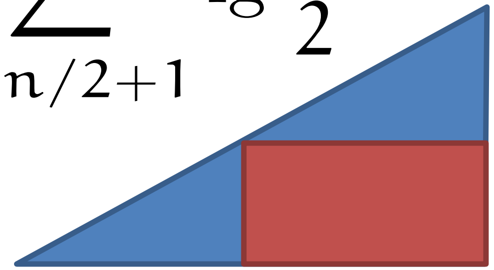
The case when *n* data are sorted

- Let k be the length of the longest path in an optimal decision tree T. Then,

  The number of leaves of T $\leqq 2^k$
- Since all possible permutations of *n* items should appear as leaves, $n! \leqq 2^k$
- By taking logarithm,

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

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



Non-comparison sort: Counting sort

- We need some assumption:

  $$\text{data}[i] \in \{1,\dots,k\} \text{ for } 1 \leqq i \leqq n,\ k \in O(n)$$

  (For example, scores of many students)
- Using values of data, it sorts in $\Theta(n)$ time.

Counting sort

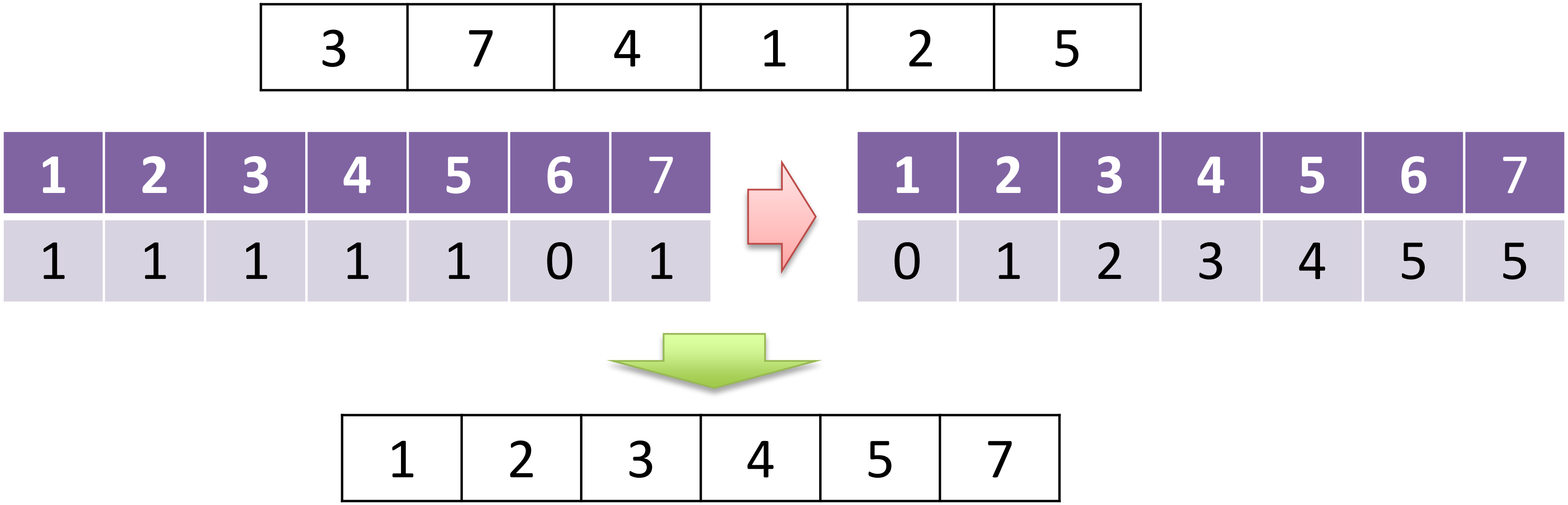
Input: $data[i] \in \{1,\dots,k\}$ for $1 \leqq i \leqq n$, $k \in O(n)$

Idea: Decide the position of element x

- Count the number of element less than x

➔That number indicates the position of x

Example:



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

Counting sort

Q. When array contains many data of same values?

A. Use 3 arrays a[], b[], c[] as follows;
(a[]: input, b[]: sorted data, c: counter)

- c[a[i]] counts the number of data equal to a[i]

- For each j with $0 \leqq j \leqq k$,
  let c’[j] := c[0] + … + c[j-1] + c[j], then
  c’[j] indicates the number of data whose value is less than j

- Copy a[i] to certain b[] according to the value of c’[]

Counting sort: program

```
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;
}
```

Initialize counter c[]

Count the number
of the value in a[i]

Compute c’[] from c[]
In an efficient way!

Copy a[] to b[]

Counting sort: Example
Sort integers (3,6,4,1,3,4,1,4)

- After (2);
  c[]=(0,2,0,2,3,0,1)
- After (3);
  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;

(2)for j=0 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;
}
```

Sort is said to be “stable” when two variables of the same value in order after sorting.

- After c[]=(0,2, 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

```
        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;
}
```