VE281

Data Structures and Algorithms

Sorting II

Review

- Simple Sorting Algorithms and Heap Sort
- Merge Sort: A Divide-and-Conquer Approach
  - Time complexity: $\Theta(n \log n)$
- Quick Sort
  - Selecting the pivot
  - In-place partitioning the array

Outline

- Quick Sort
- Comparison Sort Summary and Time Complexity
- Non-Comparison Sort
  - Counting Sort and Bucket Sort
  - Radix Sort

In-Place Partitioning the Array

Time Complexity

1. Once pivot is chosen, swap pivot to the beginning of the array.
2. Start counters **`i=1`** and **`j=N-1`**.
3. Increment **`i`** until we find element **`A[i]>=pivot`**.
4. Decrement **`j`** until we find element **`A[j]<pivot`**.
5. If **`i<j`**, swap **`A[i]`** with **`A[j]`**. Go back to step 3.
6. Otherwise, swap the first element (pivot) with **`A[j]`**.

- Scan the entire array no more than twice.
- Time complexity is $\Theta(N)$, where N is the size of the array.

Quick Sort

Time Complexity

```
void quicksort(int *a, int left,
  int right) {
    int pivotat; // index of the pivot
    if(left >= right) return;
    pivotat = partition(a, left, right);    Θ(N)
    quicksort(a, left, pivotat-1);    T(LeftSz)
    quicksort(a, pivotat+1, right);    T(RightSz)
}
```

- Let $T(N)$ be the time needed to sort N elements.
  - $T(0) = c$, where c is a constant.
- Recursive relation:

$$T(N) = T(LeftSz) + T(RightSz) + \Theta(N)$$

  - $LeftSz + RightSz = N - 1$

Quick Sort

Worst Case Time Complexity

- Recursive relation:

$$T(N) = T(LeftSz) + T(RightSz) + \Theta(N)$$

- Worst case happens when each time the pivot is the smallest item or the largest item.
  - $T(N) = T(N-1) + T(0) + \Theta(N)$

$$\begin{aligned}
&= T(N-1) + T(0) + dN \\
&= T(N-2) + 2T(0) + d(N-1) + dN \\
&\quad \cdots \\
&= T(0) + NT(0) + d + 2d + \cdots + d(N-1) + dN \\
&= \Theta(N^2)
\end{aligned}$$

Quick Sort

Best Case Time Complexity

- Recursive realtaion:

$$T(N) = T(LeftSz) + T(RightSz) + \Theta(N)$$

- Best case happens when each time the pivot divides the array into two equal-sized ones.
  - $T(N) = T((N-1)/2) + T((N-1)/2) + \Theta(N)$
  - The recursive relation is similar to that of merge sort.
  - $T(N) = \Theta(N \log N)$

Quick Sort

Average Case Time Complexity

- Average case time complexity of quick sort can be proved to be $\Theta(N \log N)$.

Quick Sort
Characteristics

- In-place?
  - In-place partitioning.
  - Worst case needs $O(N)$ stack space.
  - Average case needs $O(\log N)$ stack space.
    - "Weekly" in-place.

- Not stable.

Quick Sort

Summary

- Like merge sort, quick sort is a divide-and-conquer algorithm.
- Merge sort: easy division, complex combination.
- Quick sort: complex division (partition with pivot step), easy combination.
- Insertion sort is faster than quick sort for small arrays.
  - Terminate quick sort when array size is below a threshold. Do insertion sort on subarrays.

Outline

- Quick Sort
- Comparison Sort Summary and Time Complexity
- Non-Comparison Sort
  - Counting Sort and Bucket Sort
  - Radix Sort

Comparison Sorts

Summary

	Worst Case Time	Average Case Time	In Place	Stable
Insertion	$O(N^2)$	$O(N^2)$	Yes	Yes
Selection	$O(N^2)$	$O(N^2)$	Yes	No
Bubble	$O(N^2)$	$O(N^2)$	Yes	Yes
Heap Sort	$O(N \log N)$	$O(N \log N)$	Yes	No
Merge Sort	$O(N \log N)$	$O(N \log N)$	No	Yes
Quick Sort	$O(N^2)$	$O(N \log N)$	Weakly	No

Comparison Sorts

Summary

- How can quick sort runs at $O(N \log N)$, while insertion sort runs at $O(N^2)$?
  - Insertion sort corrects one **reverse-ordered pair** at a time.
  - Quick sort moves elements far distances, correcting multiple reverse-ordered pairs at a time.

- Why is quick sort's worst-case $O(N^2)$ while merge sort has no such a problem?
  - The choice of pivot determines size of partitions in quick sort, whereas merge sort cuts array in half every time.

Comparison Sorts

Worst Case Time Complexity

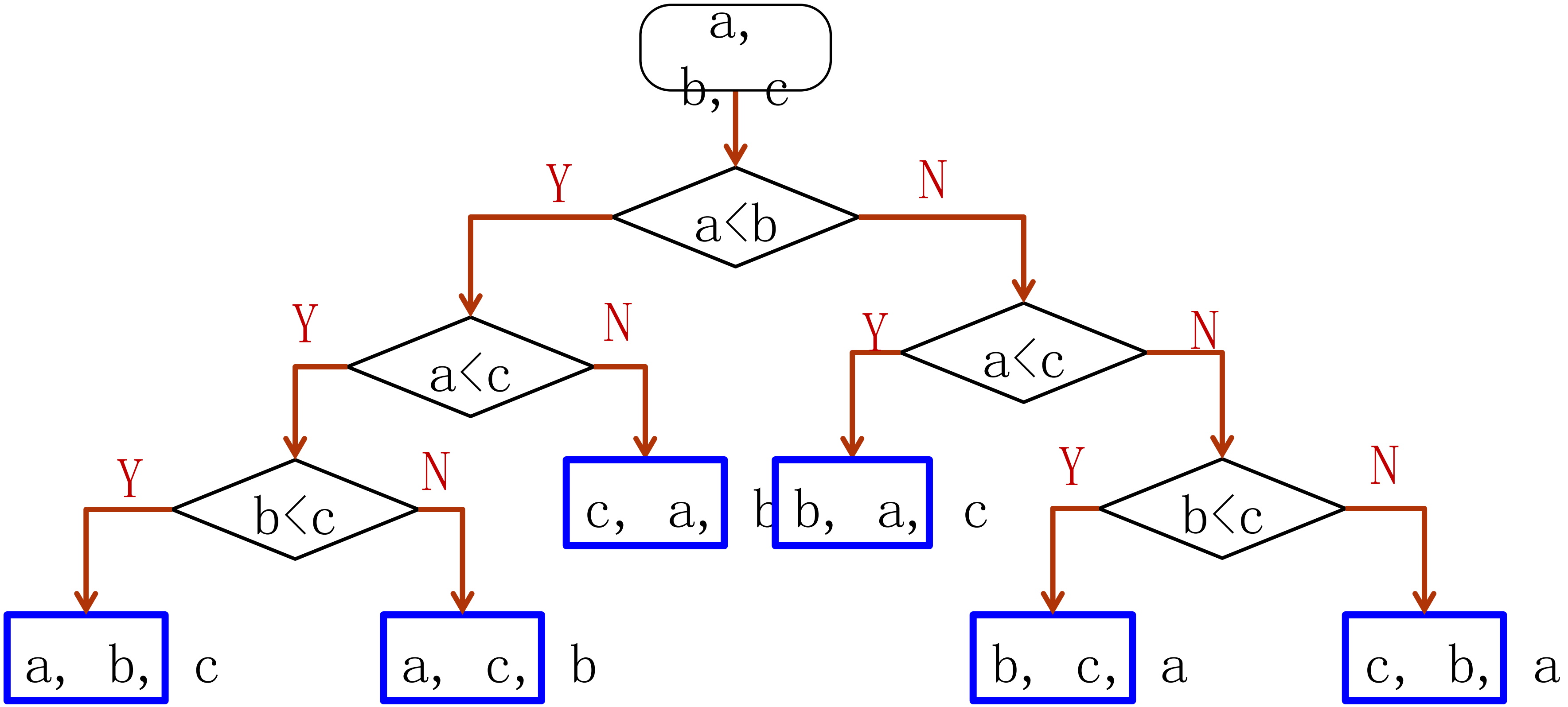
- For comparison sort, is $O(N \log N)$ the best we can do in the worst case?

- Theorem: A sorting algorithm that is based on pairwise comparisons must use $\Omega(N \log N)$ operations to sort in the worst case.

- Proof: Consider the decision tree.

Decision Tree for 3 Items

- Input: an unsorted array of 3 items a, b, c.



Decision Tree and Theoretic Lower Bound

- Decision tree is a binary tree.
- The sorting result is at one of the leaves following the results of a sequence of pairwise comparisons.
- The number of pairwise comparisons in the worst case corresponds to the deepest leaf in the decision tree, or the height of the tree.
- The number of leaves in a decision tree for sorting N items is $N!$, i.e., the number of permutations on N items.
- Since a binary tree of height h has **at most** 2^h leaves, the height of the decision tree is **at least** $\lceil \log N! \rceil$.

Theoretic Lower Bound

$$
\begin{aligned}
\log(N!) &= \log N + \log(N-1) + \cdots + \log 1 \\
&\geq \log N + \log(N-1) + \cdots + \log(N/2) \\
&\geq \frac{N}{2}\log(N/2) \\
&= \Omega(N \log N)
\end{aligned}
$$

- Thus, the worst case time complexity for comparison sorts is $\Omega(N \log N)$.
- Any way to beat the theoretic lower bound?
  - Do not compare keys: Non-comparison sort.

Outline

- Quick Sort
- Comparison Sort Summary and Time Complexity
- Non-Comparison Sort
  - Counting Sort and Bucket Sort
  - Radix Sort

Counting Sort

A Simple Version

- Sort an array A of **integers** in the range $[0, k]$, where k is known.

1. Allocate an array **`count[k+1]`**.
2. Scan array A. For i**`=1`** to **`N`**, increment **`count[A[i]]`**.
3. Scan array **`count`**. For i**`=0`** to **`k`**, print **`i`** for **`count[i]`** times.

- Time complexity: $O(N + k)$.

- The algorithm can be converted to sort integers in some other known range $[a, b]$.
  - Minus each number by a, converting the range to $[0, b - a]$.

Counting Sort

A General Version

- In the previous version, we print **`i`** for **`count[i]`** times.
  - Simple but only works when sorting integer keys alone.
  - How to sort items when there is "additional" information with each key?
- A general version:

1. Allocate an array **`C[k+1]`**.
2. Scan array A. For **`i=1`** to **`N`**, increment **`C[A[i]]`**.
3. For **`i=1`** to **`k`**, **`C[i]=C[i-1]+C[i]`**
   - **`C[i]`** now contains number of items less than or equal to **`i`**.
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position

Counting Sort

Example

1. Allocate an array **C[k+1]**.
2. Scan array A. For **i=1** to **N**, increment **C[A[i]]**.
3. For **i=1** to **k**, **C[i]= C[i-1]+C[i]**
4. For **i=N** downto **1**, put **A[i]** in new position **C[A[i]]** and decrement

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

	0	1	2	3	4	5
C	2	2	4	7	7	8

Counting Sort

Example

1. Allocate an array **C[k+1]**.

2. Scan array A. For **i=1** to **N**, increment **C[A[i]]**.

3. For **i=1** to **k**, **C[i]= C[i-1]+C[i]**

4. For **i=N** downto **1**, put **A[i]** in new position **C[A[i]]** and decrement

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	2	4	7	7	8

1	2	3	4	5	6	7	8
						3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

Counting Sort

Example

1. Allocate an array **C[k+1]**.
2. Scan array A. For **i=1** to **N**, increment **C[A[i]]**.
3. For **i=1** to **k**, **C[i]=C[i-1]+C[i]**
4. For **i=N** downto **1**, put **A[i]** in new position **C[A[i]]** and decrement

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	2	4	6	7	8

1	2	3	4	5	6	7	8
	0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

Counting Sort

Example

1. Allocate an array **C[k+1]**.

2. Scan array A. For **i=1** to **N**, increment **C[A[i]]**.

3. For **i=1** to **k**, **C[i]= C[i-1]+C[i]**

4. For **i=N** downto **1**, put **A[i]** in new position **C[A[i]]** and decrement

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	1	2	4	6	7	8

1	2	3	4	5	6	7	8
	0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

Counting Sort

Example

1. Allocate an array **C[k+1]**.
2. Scan array A. For **i=1** to **N**, increment **C[A[i]]**.
3. For **i=1** to **k**, **C[i]= C[i-1]+C[i]**
4. For **i=N** downto **1**, put **A[i]** in new position **C[A[i]]** and decrement

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	1	2	4	5	7	8

1	2	3	4	5	6	7	8
	0		2		3	3	

	0	1	2	3	4	5
C	1	2	3	5	7	8

Counting Sort

Example

1. Allocate an array **C[k+1]**.
2. Scan array A. For **i=1** to **N**, increment **C[A[i]]**.
3. For **i=1** to **k**, **C[i]=C[i-1]+C[i]**
4. For **i=N** downto **1**, put **A[i]** in new position **C[A[i]]** and decrement

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	1	2	3	5	7	8

1	2	3	4	5	6	7	8
0	0		2		3	3	

	0	1	2	3	4	5
C	0	2	3	5	7	8

Counting Sort

Example

1. Allocate an array **C[k+1]**.
2. Scan array A. For **i=1** to **N**, increment **C[A[i]]**.
3. For **i=1** to **k**, **C[i]=C[i-1]+C[i]**
4. For **i=N** downto **1**, put **A[i]** in new position **C[A[i]]** and decrement

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	0	2	3	5	7	8

1	2	3	4	5	6	7	8
0	0		2	3	3	3	

	0	1	2	3	4	5
C	0	2	3	4	7	8

Counting Sort

Example

1. Allocate an array **C[k+1]**.
2. Scan array A. For **i=1** to **N**, increment **C[A[i]]**.
3. For **i=1** to **k**, **C[i]=C[i-1]+C[i]**
4. For **i=N** downto **1**, put **A[i]** in new position **C[A[i]]** and decrement

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	0	2	3	4	7	8

1	2	3	4	5	6	7	8
0	0		2	3	3	3	5

	0	1	2	3	4	5
C	0	2	3	4	7	7

Counting Sort

Example

1. Allocate an array **C[k+1]**.
2. Scan array A. For **i=1** to **N**, increment **C[A[i]]**.
3. For **i=1** to **k**, **C[i]=C[i-1]+C[i]**
4. For **i=N** downto **1**, put **A[i]** in new position **C[A[i]]** and decrement

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	0	2	3	4	7	7

1	2	3	4	5	6	7	8
0	0	2	2	3	3	3	5

	0	1	2	3	4	5
C	0	2	2	4	7	7

Is counting sort stable?

Yes!

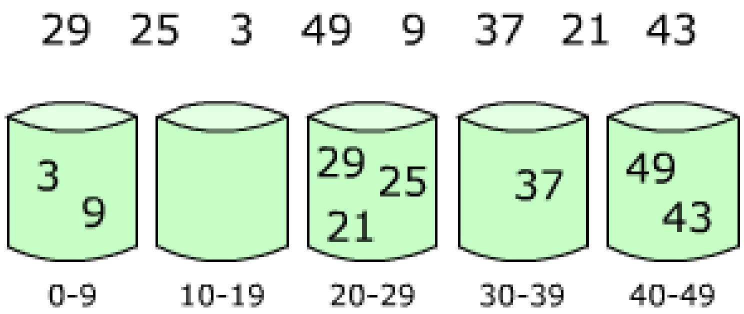
Bucket Sort

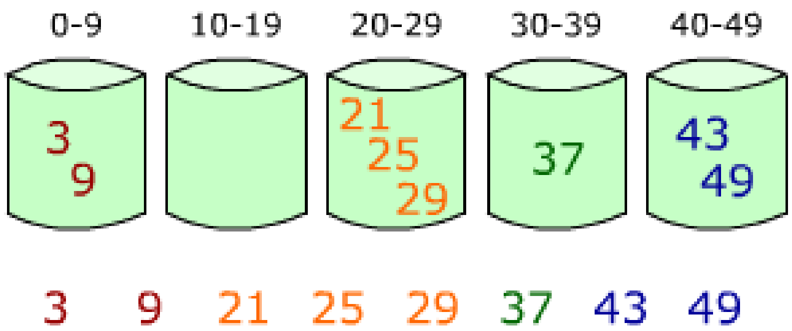
- Instead of simple integer, each key can be a complicated record, such as a real value.
- Then instead of incrementing the count of each bucket, distribute the records by their keys into appropriate buckets.
- Algorithm:

1. Set up an array of initially empty "buckets".
2. Scatter: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. Gather: Visit the buckets in order and put all elements back into the original array.

Bucket Sort

- Example





- Time complexity
  - Suppose we are sorting N items and we divide the entire range into N buckets.
  - Assume that the items are uniformly distributed in the entire range.
  - The average case time complexity is $O(N)$.

Outline

- Quick Sort
- Comparison Sort Summary and Time Complexity
- Non-Comparison Sort
  - Counting Sort and Bucket Sort
  - Radix Sort

Radix Sort

- **Radix sort** sorts integers by looking at one digit at a time.
- Procedure: Given an array of integers, from the least significant bit (LSB) to the most significant bit (LSB), repeatedly do **stable** bucket sort according to the current bit.

- For sorting base-b numbers, bucket sort needs b buckets.
  - For example, for sorting decimal numbers, bucket sort needs 10 buckets.

Radix Sort

Example

- Sort 815, 906, 127, 913, 098, 632, 278.
- Bucket sort 815, 906, 127, 913, 098, 632, 278 according to the least significant bit:

0	1	2	3	4	5	6	7	8	9
		632	913		815	906	127	098 278	

- Bucket sort 632, 913, 815, 906, 127, 098, 278 according to the second bit.

Radix Sort

Example

- Bucket sort 632, 913, 815, 906, 127, 098, 278 according to the second bit.

0	1	2	3	4	5	6	7	8	9
906	913 815	127	632				278		098

- Bucket sort 906, 913, 815, 127, 632, 278, 098 according to the most significant bit.

Radix Sort

Example

- Bucket sort 906, 913, 815, 127, 632, 278, 098 according to the most significant bit.

0	1	2	3	4	5	6	7	8	9
098	127	278				632		815	906 913

- The final sorted order is: 098, 127, 278, 632, 815, 906, 913.

Radix Sort

- Radix sort can be applied to sort keys that is built on positional notation.
  - Positional notation: all positions uses the same set of symbols, but different positions have different weight.
  - Decimal representation and binary representation are examples of positional notation.
  - Strings can also be viewed as a type of positional notation. Thus, radix sort can be used to sort strings.
- We can also apply radix sort to sort records that contain multiple keys.
  - For example, sort records (year, month, day)

Radix Sort

Time Complexity

- Let k be the maximum number of digits in the keys and N be the number of keys.
- We need to repeat bucket sort k times.
  - Time complexity for the bucket sort is $O(N)$.
- The total time complexity is $O(kN)$.