VE281

Data Structures and Algorithms

Non-comparison Sort;
Linear-Time Selection Algorithm

Outline

- Non-comparison Sort
  - Counting Sort
  - Bucket Sort
  - Radix Sort

- Randomized selection algorithm

Review: General Counting Sort

- 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 **`C[A[i]]`** and decrement **`C[A[i]]`**.

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 **`C[A[i]]`**.

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 `C[A[i]]`.

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 `C[A[i]]`.

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 `C[A[i]]`.

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 **`C[A[i]]`**.

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 **`C[A[i]]`**.

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 **`C[A[i]]`**.

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 `C[A[i]]`.

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 `C[A[i]]`.

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

Done!

Is counting sort stable?

Yes!

Outline

- Non-comparison Sort
  - Counting Sort
  - Bucket Sort
  - Radix Sort

- Randomized selection algorithm

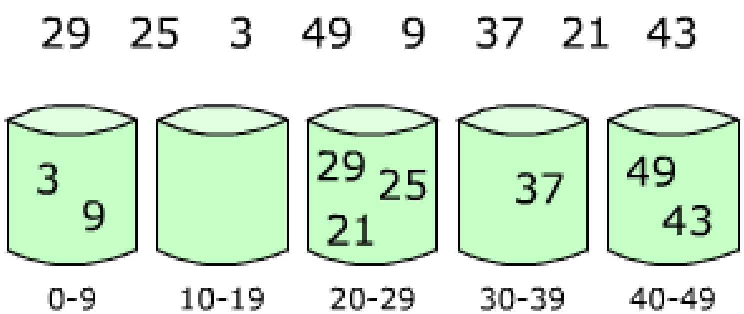
Bucket Sort

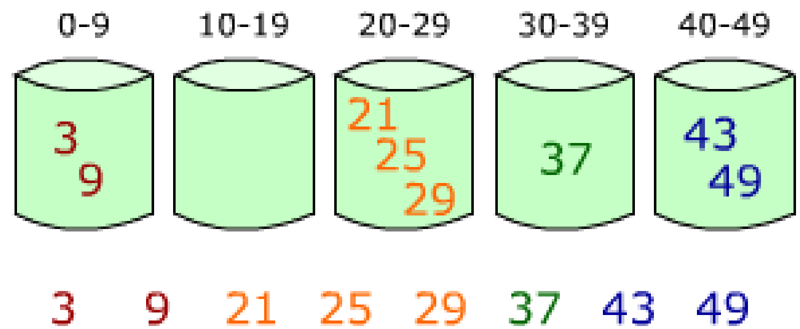
- Instead of simple integer, each key can be a complicated record, such as a real value.
- Then instead of incrementing the count of each bucket, **distribute** the records **by their keys** into appropriate buckets.
- Algorithm:

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 cN 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

- Non-comparison Sort
  - Counting Sort
  - Bucket Sort
  - Radix Sort
- Randomized selection algorithm

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: Correctness

- Claim: after bucket sorting the i-th LSB, the numbers are sorted according to their last i digits

- Proof by mathematical induction
- Inductive step
  - For two adjacent numbers if they are not in the same bucket, they are sorted according to their last i digits
  - If they are in the same bucket, they are also sorted due to stability of bucket sort

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 $O(kN)$.

Radix Sort

- Radix sort can be applied to sort keys that are 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).

Outline

- Non-comparison Sort
  - Counting Sort
  - Bucket Sort
  - Radix Sort

- Randomized selection algorithm

The Selection Problem

- Input: array A with n distinct numbers and a number i
  - "**Distinct**" for simplicity
- Output: i-th smallest element in the array
- Example: $A = (6, 3, 5, 4, 2), i = 3$
  - Should return 4
- Special cases
  - $i = 1$: the smallest item. Runtime: $O(n)$
  - $i = n$: the largest item. Runtime: $O(n)$
  - $i = n/2$: the median

Solution: Reduction to Sorting

- Step 1: Do merge sort
- Step 2: output the i-th element of the sorted array
- Time complexity is $O(n \log n)$

- Can we do better?
  - This essentially asks whether selection is **fundamentally easier** than sorting
  - Answer: Yes!
  - We will show an $O(n)$ time randomized algorithm by modifying quick sort
  - Also will show an $O(n)$ time deterministic algorithm (However, not as practical as the randomized algorithm)

Recall: Partitioning in Quick Sort

- Pick a pivot
- Put all elements < pivot to the left of pivot
- Put all elements ≥ pivot to the right of pivot
- Move pivot to its correct place on the array

6	2	8	5	11	10	4	1	9	7	3

pivot

4	2	3	5	1	6	10	11	9	7	8

Basic Idea

- Suppose we are looking for 6^{th} smallest item in an array of length 12. We do partition.
  - Suppose the pivot is at position 4. Then we only need to focus on the sub-array right of the pivot and look for the 2^{nd} item in the array
  - Suppose the pivot is at position 8. Then we only need to focus on the sub-array left of the pivot and look for the 6^{th} item in the array
  - In both case, recurse!

Randomized Selection

```
Rselect(int A[], int n, int i) {
// find i-th smallest item of array A of size n
  if(n == 1) return A[1];
  Choose pivot p from A uniformly at random;
  Partition A using pivot p;
  Let j be the index of p;
  if(j == i) return p;
  if(j > i) return Rselect(1st part of A, j-1, i);
  else return Rselect(2nd part of A, n-j, i-j);
}
```

Runtime of Rselect

- Depends on the quality of the chosen pivots

- Consider $i = n/2$. What is a worst pivot sequence and what is the worst case runtime?
  - $\Theta(n^2)$
- Best case for an arbitrary i: the pivot is always the median
  - The pivot gives "balanced" split
  - Recurrence: $T(n) \leq T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = O(n)$
- Average case?

Average Case Runtime of Rselect

- Theorem: for every input array of length n, the average runtime of Rselect is $O(n)$
  - Holds for every input data (no assumption on data)
  - "Average" is over random pivot choices made by the algorithm

Average Case Runtime Analysis

- Note: Rselect uses $\leq cn$ operations outside of recursive call (from partitioning)
- Observation: the length of the array the algorithm works on decreases
- Definition: We say Rselect is in phase j if current array size is between $(\frac{3}{4})^{j+1}n$ and $(\frac{3}{4})^{j}n$
- X_j denote the number of recursive calls in phase j
- $runtime \leq \sum_j X_j \cdot c \cdot (\frac{3}{4})^{j}\, n$

Average Case Runtime Analysis

- If Rselect chooses a pivot so that the left sub-array's size is am, where $a \in [\frac{1}{4}, \frac{3}{4}]$ and m is the old length, then the current phase ends
  - Because new sub-array length is at most 75% of the old length
  - **"Good pivot"**
- What is the probability of $a \in [\frac{1}{4}, \frac{3}{4}]$ (i.e., good pivot)?
  - Answer: 0.5
- Claim: $E[X_j] \leq$ Expected number of times you need to flip a fair coin to get a "head"
  - Heads: good pivot; tails: bad pivot

Coin Flipping Analysis

- Let N be the number of coin flips until you get heads
  - N is a geometric random variable: $P(N = k) = \frac{1}{2^k}, k = 1,2, \dots$

#flips when 1st is head

#flips when 1st is tail

- $E[N] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (1 + E[N]) \Rightarrow E[N] = 2$

Prob. 1st flip is head

Prob. 1st flip is tail

Therefore, $E[X_j] \leq E[N] = 2$

Average Case Runtime Analysis

$$E[runtime] \leq E\left[\sum_j X_j \cdot c \cdot \left(\frac{3}{4}\right)^j n\right]$$

$$= cn \sum_j \left(\frac{3}{4}\right)^j E[X_j] \quad \leq 2cn \sum_j \left(\frac{3}{4}\right)^j \leq 2cn \frac{1}{1 - \frac{3}{4}}$$

$$= 8cn = O(n)$$