Revised: 1/3/2020

Non-Comparative Sorting

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Counting Sort
  - Complexities
- Radix Sort
  - Complexities
- Bucket Sort
  - Complexities

Non-Comparative Sorting Algorithms

- **Non-comparative sorting algorithms** sort the contents of a sequence using the *characters* of values to be sorted.
  - Unlike comparative sorting algorithms, which base their sorting on making relational (<, >, etc.) comparisons.

Counting Sort

- The Counting Sort is a **non-comparative** sorting algorithm that uses a separate "counting" array for determining how many times an integer appears in an unsorted sequence.

- The counting array uses its own indexes to hold the totals of that corresponding value in the unsorted sequence.
  - For example, for the unsorted array {2, 3, 2, 1} a Counting Sort's counting array would contain {0, 1, 2, 1}
    - Zero 0's, One 1, Two 2's, One 3

{2, 3, 2, 1}

{0, 1, 2, 1}

0 1 2 3

Counting Sort

- The values in the counting array are summed linearly.
  - The counting array previously shown would become {0, 1, 3, 4}

{0, 1, 2, 1}

{0, 1, 2, 1}

{0, 1, **3**, 1}

{0, 1, 3, **4**}

Counting Sort

- The counting array (c) is then used to determine the placement of each unsorted element.

- Each value is retrieved from the original array.
  - 1 is subtracted from corresponding index of the counting array.
    - If 2 I retrieved from the original array, 1 is subtracted from index 2 of the counting array.

- The new value at the corresponding index is the index where the value is placed in the resulting array (r).

- The sorted array (r) is then copied over to the original array (a)

Counting Sort

a = {**2**, 3, 2, 1} c = {0, 1, 3, 4} r = {0, 0, 0, 0} *Get index 2 of c*	a = {**2**, 3, 2, 1} c = {0, 1, **2**, 4} r = {0, 0, 0, 0} *Subtract 1*	a = {**2**, 3, 2, 1} c = {0, 1, **2**, 4} r = {0, 0, **2**, 0} *Place value at that index in r*
a = {2, **3**, 2, 1} c = {0, 1, 2, 4} r = {0, 0, 2, 0} *Get index 3 of c*	a = {2, **3**, 2, 1} c = {0, 1, 2, **3**} r = {0, 0, 2, 0} *Subtract 1*	a = {2, **3**, 2, 1} c = {0, 1, 2, **3**} r = {0, 0, 2, **3**} *Place value at that index in r*

Counting Sort

a = {2, 3, **2**, 1}
c = {0, 1, 2, 3}
r = {0, 0, 2, 3}
Get index 2 of c

a = {2, 3, **2**, 1}
c = {0, 1, **1**, 3}
r = {0, 0, 2, 3}
Subtract 1

a = {2, 3, **2**, 1}
c = {0, 1, **1**, 3}
r = {0, **2**, 2, 3}
Place value at that index in r

a = {2, 3, 2, **1**}
c = {0, 1, 1, 3}
r = {0, 2, 2, 3}
Get index 1 of c

a = {2, 3, 2, **1**}
c = {0, **0**, 1, 3}
r = {0, 2, 2, 3}
Subtract 1

a = {2, 3, 2, **1**}
c = {0, **0**, 1, 3}
r = {**1**, 2, 2, 3}
Place value at that index in r

Counting Sort (C++ Function)

```
void countingSort(int a[], int length) {
    int result[length];
    int max = a[0];
    for(int i = 1; i < length; i++) {
        if(a[i] > max) {
            max = a[i];
        }
    }
    int c[max + 1];
    for(int i = 0; i < max+1; i++) {
        c[i] = 0;
    }
    for(int i = 0; i < length; i++) {
        int value = a[i];
        c[value] += 1;
    }
    for(int i = 1; i < max + 1; i++) {
        c[i] += c[i-1];
    }
    for(int i = 0; i < length; i++) {
        int temp = a[i];
        c[temp] -= 1;
        result[c[temp]] = temp;
    }
    for(int i = 0; i < length; i++) {
        a[i] = result[i];
    }

}
```

Determine max

Zero out the “c” array

Increment the element at index a[i] in c

Linearly sum the elements of c

Subtract 1 from c[a[i]] and
Put a[i] in result[c[a[i]]]

Copy each element from result to a

Counting Sort Time Complexity

- Where k is the length of the counting array.
  - Loops n-times = 4
  - Loops k-times = 2
  - 4n + 2k = **O(n + k)**
  - **Linear**

Counting Sort Space Complexity

- Total Space Complexity:
  - Original Array = n
  - Temporary Array = n
  - Counting array = k
  - n + n + k = 2n + k = **O(n + k)**
  - **Linear**

- Sometimes it is inefficient with the auxiliary space.
  - Array to sort: {4, 10, 3}
  - Counting array: {0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1}

- Sometimes it is efficient with the auxiliary space.
  - Array to sort: {1, 2, 0, 1, 2, 0, 1, 2, 2, 0, 1}
  - Counting array: {3, 4, 4}

Radix Sort

- The Radix Sort is another non-comparative sorting algorithm that is closely related to the Counting Sort algorithm.

- The Radix Sort sorts an sequence of numbers, going digit-by-digit of each value, starting with the least-significant digit to the most-significant digit.
  - The algorithm uses a modified Counting Sort to perform the actual sorting.

Radix Sort

Sorting by the 1's place

a = {45, 32, 7, 19}

c = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

a = {45, 35, 7, 19}

c = {0, 0, 0, 0, 0, 2, 0, 1, 0, 1}

c = {0, 0, 0, 0, 0, 2, 2, 3, 3, 4}

a = {35, 45, 7, 19}

Sorting by the 10's place

a = {35, 45, 7 19}

c = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

a = {35, 45, 07, 19}

c = {1, 1, 0, 1, 1, 0, 0, 1, 0, 1}

c = {1, 2, 2, 3, 4, 4, 4, 5, 5, 6}

a = {7, 19, 35, 45}

Radix Sort (C++ Function)

```
void radixSort(int a[], int length) {
    int max = getMax(a, length);
    for (int i = 1; max/i > 0; i *= 10) {
        countingSort(a, length, i);
    }
}
```

```
int getMax(int a[], int length) {
    int max = a[0];
    for (int i = 1; i < length; i++) {
        if (a[i] > max) {
            max = a[i];
        }
    }
    return max;
}
```

```
void countingSort(int a[], int length, int i) {
    int temp[length];
    int digitCount[10] = {0};

    for (int j = 0; j < length; j++) {
        digitCount[(a[j] / i) % 10]++;
    }
    for (int j = 1; j < 10; j++) {
        digitCount[j] += digitCount[j - 1];
    }

    for (int j = length - 1; j >= 0; j--) {
        int index = digitCount[(a[j] / i) % 10] - 1;
        temp[index] = a[j];
        digitCount[(a[j] / i) % 10]--;
    }

    for (int j = 0; j < length; j++) {
        a[j] = temp[j];
    }
}
```

Radix Sort (C++ Function)

```
void radixSort(int a[], int length) {
    int max = getMax(a, length);
    for (int i = 1; max/i > 0; i *= 10) {
        countingSort(a, length, i);
    }
}
```

Find the largest number; This will determine how many times the below loop repeats

Iterates from 1, to 10, to 100, and so on… based on the largest number

Passes the array, its length, and the current position to sort to a Counting Sort algorithm

Radix Sort (C++ Function)

```
void countingSort(int a[], int length, int i) {
    int temp[length];
    int digitCount[10] = {0};

    for (int j = 0; j < length; j++) {
        digitCount[(a[j] / i) % 10]++;
    }
    for (int j = 1; j < 10; j++) {
        digitCount[j] += digitCount[j - 1];
    }

    for (int j = length - 1; j >= 0; j--) {
        int index = digitCount[(a[j] / i) % 10] - 1;
        temp[index] = a[j];
        digitCount[(a[j] / i) % 10]--;
    }

    for (int j = 0; j < length; j++) {
        a[j] = temp[j];
    }
}
```

Counting array with a length of 10 (indexes 0-9)

Finds the digit at the i position of each number in a, and adds one at that index in the counting array

Linearly sums the values in the counting array

Subtracts 1 from the value in the counting array, for each value in a (based on the current digit/position it is sorting for). Puts the at the calculated index in a temporary array. Decrements the index by one.

Copies all values from the temporary array to the actual array (replacing the original ordering)

Radix Sort Time Complexity

- Radix Sort loop repeats d times, where d is the number of digits in the largest value.
  - Counting Sort loops (all repeated d times)
    - Loops that repeat n-times = 3
    - Loops that repeat k-times (always <u>10</u> times) = 1
  - d * 3n + 10 = **O(n * d)**
  - **Linear**

Radix Sort Space Complexity

- Total Space Complexity
  - Length of the array to be sorted = n
  - k is the array length of counting sort's counting array, which will always be 10
  - n + 10 = **O(n)**
  - **Linear**

Bucket Sort

- The Bucket Sort is a sorting algorithm that:
  - Distributes the values of a sequence into containers or “buckets”
  - Sorts the buckets
  - Concatenated the buckets into the final, sorted result.

- Each bucket will contain the values in a certain range.
  - For example, if the range of values to be sorted is 0-100…
  - There will be a bucket for values 0-10, a bucket for values 11-20, a bucket for values 21-30, and so on.

Bucket Sort

- We'll apply the bucket sort algorithm on the following array:

{23, 16, 8, 42, 4, 15}

- First, we decide how many buckets we want.
  - In this example, we will use three.
- Next, we find the largest value in the sequence.
  - In this example, it is 42

Bucket Sort

- Now, we distribute the values into their buckets

- Before we do, let's calculate the range of each bucket (for reference)
  - Each bucket's range: $\frac{M+1}{N}$
    - M = Largest Value (42)
    - N = Number of Buckets (3)

$\frac{42+1}{3} = 14.33 \sim 15$ (<u>Round the result up</u>)

Bucket Sort

- Each bucket has a range of 15
  - Bucket 0: Will contain values 0 – 14
  - Bucket 1: Will contain values 15 – 29
  - Bucket 2: Will contain values 30 – 44

- We calculate the bucket number for a value with the following formula:

$$value\ * \frac{N}{M+1}$$

  - (Round the result down)

Bucket Sort

- $23 \; * \; \frac{3}{42+1} = 1.6 \sim 1$

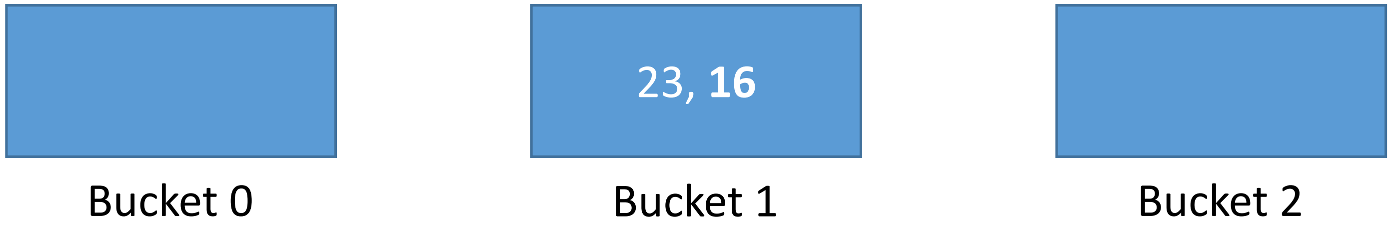
{**23**, 16, 8, 42, 4, 15}

- The value 23 would be placed in bucket 1

Bucket Sort

- $16 * \frac{3}{42+1} = 1.1 \sim 1$

{23, **16**, 8, 42, 4, 15}

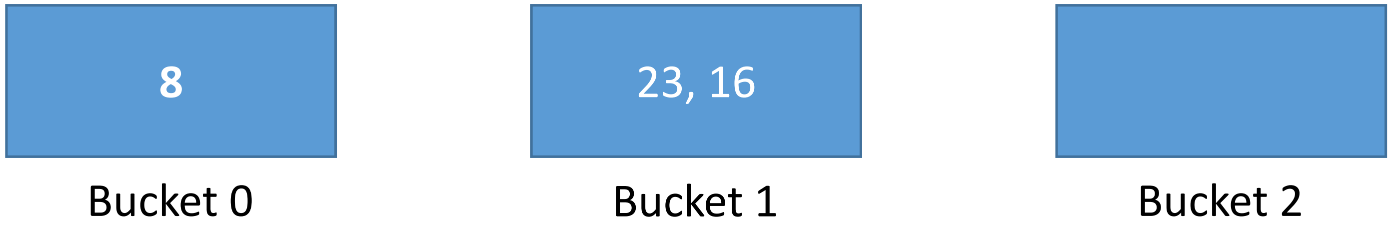


- The value 16 would be placed in bucket 1

Bucket Sort

- $8 * \frac{3}{42+1} = 0.5 \sim 0$

{23, 16, **8**, 42, 4, 15}



- The value 8 would be placed in bucket 0

Bucket Sort

- $42 * \frac{3}{42+1} = 2.9 \sim 2$

{23, 16, 8, **42**, 4, 15}

8	23, 16	42
Bucket 0	Bucket 1	Bucket 2

- The value 42 would be placed in bucket 2

Bucket Sort

- $4 * \frac{3}{42+1} = 0.3 \sim 0$

{23, 16, 8, 42, **4**, 15}

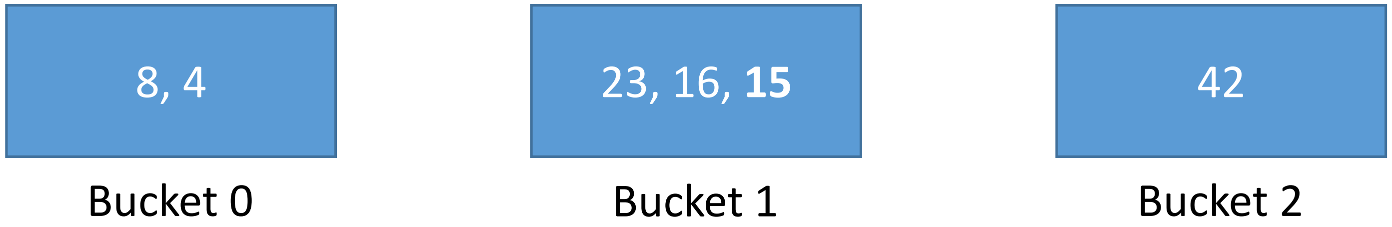
8, **4**	23, 16	42
Bucket 0	Bucket 1	Bucket 2

- The value 4 would be placed in bucket 0

Bucket Sort

- $15 * \frac{3}{42+1} = 1.04 \sim 1$

{23, 16, 8, 42, 4, **15**}



- The value 15 would be placed in bucket 1

Bucket Sort

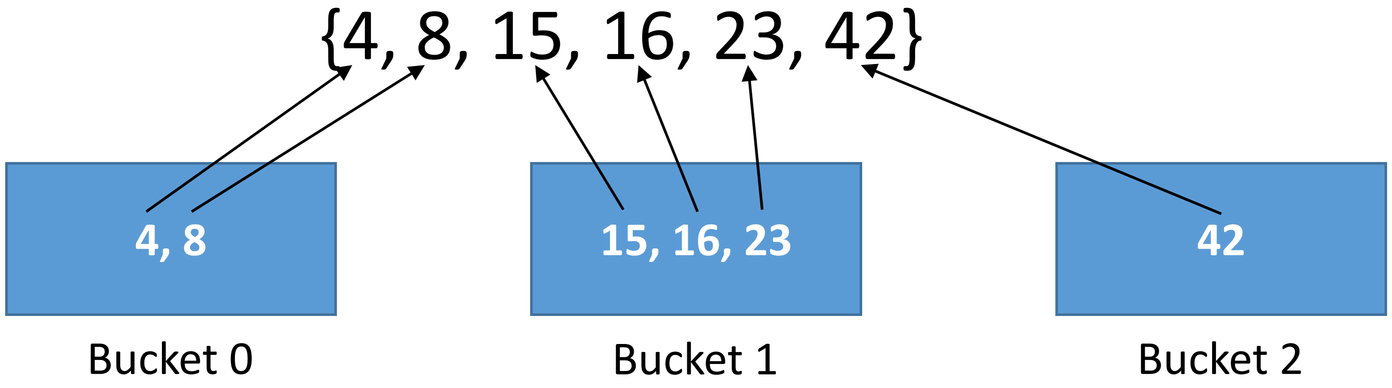
- Now, each bucket is sorted using a sorting algorithm, like insertion sort.

{23, 16, 8, 42, 4, 15}

4, 8	15, 16, 23	42
Bucket 0	Bucket 1	Bucket 2

Bucket Sort

- Finally, each value is placed back in the original array, beginning with the first bucket.



Bucket Sort

- We know the insertion sort will make, at most, $\sum_{i=1}^{n-1} \boldsymbol{i}$ comparisons.
  - We'll change this sequence so it is in reverse order, which will make the algorithm to the most comparisons

{42, 23, 16, 15, 8, 4}

- To sort this sequence, the insertion sort alone will make…
  - $\sum_{i=1}^{6-1} \boldsymbol{i}$ = 1 + 2 + 3 + 4 + 5 = 15 comparisons

Bucket Sort

- Now, we'll perform the bucket sort process on the same sequence.
  - The buckets and ordering of values would actually remain the same.

{42, 23, 16, 15, 8, 4}

8, 4	23, 16, 15	42
Bucket 0	Bucket 1	Bucket 2

Bucket Sort

- An insertion sort is performed on each bucket.

{42, 23, 16, 15, 8, 4}

Bucket 0	Bucket 1	Bucket 2
8, 4	23, 16, 15	42
$\sum_{i=1}^{2-1} i$ = 1 comparison	$\sum_{i=1}^{3-1} i$ = 3 comparisons	$\sum_{i=1}^{1-1} i$ = 0 comparisons

- **4 Comparisons**
  - 15 comparisons made by using the insertion sort alone.

Bucket Sort

```
void bucketSort(int a[], int length, int numBuckets) {
    vector<int> buckets[numBuckets];
    int max = a[0];
    for(int i = 1; i < length; i++) {
        if(a[i] > max) {
            max = a[i];
        }
    }

    for(int i = 0; i < length; i++) {
        int bIndex = (int)(a[i] * numBuckets / (max+1));
        buckets[bIndex].push_back(a[i]);
    }
    for(int i = 0; i < numBuckets; i++) {
        insertionSort(buckets[i], buckets[i].size());
    }
    int index = 0;
    for (int i = 0; i < numBuckets; i++) {
        while(!buckets[i].empty()) {
            a[index++] = *(buckets[i].begin());
            buckets[i].erase(buckets[i].begin());
        }
    }
}
```

An array of vectors
(The vectors will be our “buckets”)

Determine the max/largest value

Place each value into their correct bucket

Sort each bucket

Put each value from the buckets into the array

Bucket Sort Time Complexity

- Ignoring the sorting process for a moment:
  - Determine the max = O(n)
  - Put each value in the correct bucket = O(n)
  - Putting the sorted bucket values into the array = O(n)
    - It might look polynomial at first, but every value in each bucket is used once, and there will always be n values spread across the buckets.

- We will sort however many buckets we have.
- With an insertion sort, it will be performed b times, where b is the number of buckets:
  - Performing the sort for each bucket: **$O(b*n^2)$ (Polynomial)**
  - (It is only polynomial here because it uses the insertion sort algorithm)

Bucket Sort Time Complexity

- As long as the values in the sequence are well distributed, the bucket sort should perform well.
- For example:

  {7, 5, 2, 8, 3, 99, 6}

- If broken up into, say, 10 buckets (0-10, 11-20… 91-100)
  - All but one value will be in the first bucket.
- This sequence would not be a good one to sort with the bucket sort.

Bucket Sort Space Complexity

- Total Space Complexity
  - Length of the array to be sorted = n
  - Total size of all buckets combined = n
  - n + n = 2n = **O(n)**
  - **Linear**