Design and Analysis of Algorithms

Instructor: Sharma Thankachan

Lecture 6: Sorting in Linear Time
About this lecture

• Sorting algorithms we studied so far
  - Insertion, Selection, Merge, Quicksort
  ➔ determine sorted order by comparison

• We will look at 3 new sorting algorithms
  - Counting Sort, Radix Sort, Bucket Sort
  ➔ assume some properties on the input, and determine the sorted order by distribution
Helping the Billionaire

• Your boss, Bill, is a billionaire
• Inside his BIG wallet, there are a lot of bills, say, $n$ bills
• Nine kinds of bills: $1, 5, 10, 20, 50, 100, 200, 500, 1000$
Helping the Billionaire

• He did not care about the ordering of the bills before.

• But then, he has taken the Algorithm course, and learnt that if things are sorted, we can search faster.

The horoscope says I should use only $500 notes today ... Do I have enough in the wallet?
A Proposal

• Create a bin for each kind of bill
• Look at his bill one by one, and place the bill in the corresponding bin
• Finally, collect bills in each bin, starting from $1-bin, $5-bin, ..., to $1000-bin
A Proposal

• In the previous algorithm, there is no comparison between the items ...
  • But we can still sort correctly... WHY?

• Each step looks at the value of an item, and distribute the item to the correct bin
  • So, in the end, when a bill is collected, its value must be larger than or equal to all bills collected before \( \Rightarrow \) sorted
Sorting by Distribution

• Previous algorithm sorts the bills based on distribution operations

• It works because:
  • we have information about the values of the input items ➔ we can create bins

• We will look at more algorithms which are based on the same distribution idea
Counting Sort
Counting Sort

• Input: Array $A[1..n]$ of $n$ integers, each has value from $[0,k]$

• Output: Sorted array of the $n$ integers

• Idea 1: Create $B[1..n]$ to store the output

• Idea 2: Process $A[1..n]$ from right to left
  • Use $k + 2$ counters:
    • One for “which element to process”
    • $k + 1$ for “where to place”
Counting Sort (Details)

Before Running

A

2 1 2 5 3 3 1 2

k+1 counters

c[0], c[1], c[2], c[3], c[4], c[5]

next element

B


**Counting Sort (Details)**

Step 1: Set $c[j]$ = location in $B$ for placing the next element if it has value $j$

$A$

```
2 1 2 5 3 3 1 2
```

$B$

```
c[0] = 0
c[1] = 2
c[2] = 5
c[3] = 7
c[4] = 7
c[5] = 8
```
Counting Sort (Details)

Step 2: Process next element of A and update corresponding counter

A

\[ 2 \quad 1 \quad 2 \quad 5 \quad 3 \quad 3 \quad 1 \quad 2 \]

B

\[ \text{c}[0] = 0 \quad \text{c}[1] = 2 \quad \text{c}[2] = 4 \quad \text{c}[3] = 7 \quad \text{c}[4] = 7 \quad \text{c}[5] = 8 \]
Counting Sort (Details)

Step 2: Process next element of A and update corresponding counter

A

2 1 2 5 3 3 1 2

c[2] = 4
c[5] = 8

c[0] = 0
c[1] = 1
c[3] = 7
c[4] = 7

B

1 2
Counting Sort (Details)

Step 2: Process next element of $A$ and update corresponding counter

$A$

2 1 2 5 3 3 1 2

next element

c[2] = 4

c[5] = 8

c[0] = 0

c[1] = 1

c[3] = 6

c[4] = 7

$B$

1 2 3
Counting Sort (Details)

Step 2: Process next element of A and update corresponding counter

A

2 1 2 5 3 3 1 2

next element

c[2] = 4

c[5] = 8

B

1 2 3 3

c[0] = 0
c[1] = 1
c[3] = 5
c[4] = 7
Counting Sort (Details)

Step 2: Process next element of A and update corresponding counter

\[ A = 2 \ 1 \ 2 \ 5 \ 3 \ 3 \ 1 \ 2 \]

\[ B = 1 \ \_ \ 2 \ 3 \ 3 \ \_ \ 5 \]

- \( c[0] = 0 \)
- \( c[1] = 1 \)
- \( c[2] = 4 \)
- \( c[3] = 5 \)
- \( c[4] = 7 \)
- \( c[5] = 7 \)
Counting Sort (Details)

Step 2: Process next element of A and update corresponding counter

A

2 1 2 5 3 3 1 2

next element


B

1 2 2 3 3 5

Step 2: Process next element of A and update corresponding counter

\( A \)

\[
\begin{array}{cccccc}
2 & 1 & 2 & 5 & 3 & 3 & 1 & 2 \\
\end{array}
\]

next element

\( c[2] = 3 \quad c[5] = 7 \)

\( B \)

\[
\begin{array}{cccccc}
1 & 1 & 2 & 2 & 3 & 3 & 5 \\
\end{array}
\]

\( c[0] = 0 \quad c[1] = 0 \quad c[3] = 5 \quad c[4] = 7 \)
Counting Sort (Details)

Step 2: Done when all elements of A are processed

Next element

c[0] = 0
c[1] = 0
c[2] = 3

c[3] = 5

c[4] = 7

c[5] = 7

A

2 1 2 5 3 3 1 2

B

1 1 2 2 2 3 3 5
Counting Sort (Step 1)

How can we perform Step 1 smartly?

1. Initialize $c[0]$, $c[1]$, ..., $c[k]$ to 0

2. /* First, set $c[j] = \# \text{ elements with value } j$ */
   For $x = 1,2,...,n$, increase $c[A[x]]$ by 1

3. /* Set $c[j] = \text{ location in } B \text{ to place next element whose value is } j$ (iteratively) */
   For $y = 1,2,...,k$, $c[y] = c[y-1] + c[y]$

Time for Step 1 = $O(n + k)$
Counting Sort (Step 2)

How can we perform Step 2?

/* Process A from right to left */

For \( x = n, n-1, \ldots, 2, 1 \)

\[
\begin{align*}
\text{Time for Step 2} & = O( n )
\end{align*}
\]
Conclusion:

- **Running time** = $O(n + k)$
  - if $k = O(n)$, time is (asymptotically) optimal
- **Counting sort is also stable**: elements with same value appear in same order in before and after sorting
Stable Sort

Before Sorting

After Sorting
Radix Sort
Radix Sort

- Input: Array $A[1..n]$ of $n$ integers, each has $d$ digits, and each digit has value from $[0,k]$
- Output: Sorted array of the $n$ integers
- Idea: Sort in $d$ rounds
  - At Round $j$, stable sort $A$ on digit $j$ (where rightmost digit = digit 1)

extra info on values
Radix Sort (Example Run)

Before Running

1904
2579
1874
6355
4432
8318
1304

4 digits
Radix Sort (Example Run)

Round 1: Stable sort digit 1

1 9 0 4
2 5 7 9
1 8 7 4
6 3 5 5
4 4 3 2
8 3 1 8
1 3 0 4

4 4 3 2
1 9 0 4
1 8 7 4
1 3 0 4
6 3 5 5
8 3 1 8
2 5 7 9
Radix Sort (Example Run)

Round 2: Stable sort digit 2

After Round 2, last 2 digits are sorted (why?)
Radix Sort (Example Run)

Round 3: Stable sort digit 3

1904 1304 8318 4432 6355 1874 2579

After Round 3, last 3 digits are sorted (why?)
Radix Sort (Example Run)

Round 4: Stable sort digit 4

After Round 4, last 4 digits are sorted (why?)
Radix Sort (Example Run)

Done when all digits are processed

1 3 0 4
1 8 7 4
1 9 0 4
2 5 7 9
4 4 3 2
6 3 5 5
8 3 1 8

The array is sorted (why?)
Radix Sort (Correctness)

Question:
“After $r$ rounds, last $r$ digits are sorted”
Why ??

Answer:
This can be proved by induction:
The statement is true for $r = 1$
Assume the statement is true for $r = k$
Then ...
Radix Sort (Correctness)

At Round $k+1$,

- if two numbers differ in digit “$k+1$“, their relative order [based on last $k+1$ digits] will be correct after sorting digit “$k+1$”
- if two numbers match in digit “$k+1$“, their relative order [based on last $k+1$ digits] will be correct after stable sorting digit “$k+1$” (why?)

$\Rightarrow$ Last “$k+1$” digits sorted after Round “$k+1$“
Conclusion:

• After $d$ rounds, last $d$ digits are sorted, so that the numbers in $A[1..n]$ are sorted.

• There are $d$ rounds of stable sort, each can be done in $O(n + k)$ time.

$\Rightarrow$ Running time = $O(d(n + k))$

• If $d=O(1)$ and $k=O(n)$, asymptotically optimal.
Bucket Sort
Bucket Sort

- Input: Array $A[1..n]$ of $n$ elements, each is drawn uniformly at random from the interval $[0,1)$
- Output: Sorted array of the $n$ elements
- Idea:
  Distribute elements into $n$ buckets, so that each bucket is likely to have fewer elements → easier to sort

extra info on values
Bucket Sort (Details)

Before Running

0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68

Step 1: Create \( n \) buckets

\( n = \#\text{buckets} = \#\text{elements} \)

Each bucket represents a subinterval of size \( \frac{1}{n} \)
Bucket Sort (Details)

Step 2: Distribute each element to correct bucket

If $\text{Bucket } j$ represents subinterval $[j/n, (j+1)/n)$, element with value $x$ should be in Bucket $b \times nc$
Bucket Sort (Details)

Step 3: Sort each bucket (by insertion sort)
Bucket Sort (Details)

Step 4: Collect elements from Bucket 0 to Bucket n-1

Sorted Output: 0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94
Bucket Sort (Running Time)

- Let $X = \#$ comparisons in all insertion sort
  
  Running time = $\Theta(n + X)$

  $\Rightarrow$ worst-case running time = $\Theta(n^2)$

- How about average running time?

Finding average of $X$ (i.e. #comparisons) gives average running time
Average Running Time

Let \( n_j \) = \# elements in Bucket \( j \)

\[ X \cdot c(n_0^2 + n_1^2 + \ldots + n_{n-1}^2) \]

So,

\[
E[X] \cdot E[c(n_0^2 + n_1^2 + \ldots + n_{n-1}^2)] \\
= c \ E[n_0^2 + n_1^2 + \ldots + n_{n-1}^2] \\
= c \ (E[n_0^2] + E[n_1^2] + \ldots + E[n_{n-1}^2]) \\
= cn \ E[n_0^2] \quad \text{(by symmetry)}
\]
Average Running Time

Textbook (pages 175-176) shows that

$$E[n_0^2] = 2 - (1/n)$$

$$\Rightarrow E[X] \cdot cn E[n_0^2] = 2cn - c$$

In other words, $E[X] = O(n)$

$$\Rightarrow \text{Average running time} = \Theta(n)$$
For Interested Classmates

The following is how we can show

\[ E[n_0^2] = 2 - (1/n) \]

Recall that \( n_0 \) = \# elements in Bucket 0

So, suppose we set

\[ Y_k = 1 \] if element \( k \) is in Bucket 0

\[ Y_k = 0 \] if element \( k \) not in Bucket 0

Then, \( n_0 = Y_1 + Y_2 + \ldots + Y_n \)
For Interested Classmates

Then,

\[ E[n_0^2] = E[(Y_1 + Y_2 + \ldots + Y_n)^2] \]

\[ = E[ Y_1^2 + Y_2^2 + \ldots + Y_n^2 + Y_1 Y_2 + Y_1 Y_3 + \ldots + Y_1 Y_n + Y_2 Y_1 + Y_2 Y_3 + \ldots + Y_2 Y_n + \ldots + Y_n Y_1 + Y_n Y_2 + \ldots + Y_n Y_{n-1} ] \]
\[= \mathbb{E}[Y_1^2] + \mathbb{E}[Y_2^2] + \ldots + \mathbb{E}[Y_n^2] + \mathbb{E}[Y_1Y_2] + \ldots + \mathbb{E}[Y_nY_{n-1}]\]

\[= n \mathbb{E}[Y_1^2] + n(n-1) \mathbb{E}[Y_1Y_2]\]

(by symmetry)

The value of \(Y_1^2\) is either 1 (when \(Y_1 = 1\)), or 0 (when \(Y_1 = 0\)).

The first case happens with \(1/n\) chance (when element 1 is in Bucket 0), so

\[\mathbb{E}[Y_1^2] = 1/n \times 1 + (1 - 1/n) \times 0 = 1/n\]
For $Y_1 Y_2$, it is either 1 (when $Y_1=1$ and $Y_2=1$),
or 0 (otherwise)
The first case happens with $1/n^2$ chance
(when both element 1 and element 2 are in Bucket 0), so
$$E[Y_1 Y_2] = 1/n^2 \cdot 1 + (1 - 1/n^2) \cdot 0 = 1/n^2$$

Thus,
$$E[n_0^2] = n E[Y_1^2] + n(n-1) E[Y_1 Y_2]$$
$$= n (1/n) + n(n-1) (1/n^2)$$
$$= 2 - 1/n$$