We can prove that:

- **Comparison sorts**
  - The comparison between the input elements is based on their order.

**Interview common properties**

- Average-case: worst-case
- Average-case: best-case
- Average-case: random-case

**O(n log n)**

**Exampl**

- Every element **x** can be placed directly in its position in the sorted array, assuming that each input element is unique before moving into the new array.

**Counter sort**

**Example**

- Given an array of elements, the counter sort algorithm can be used to sort the elements in ascending order.

**Non-comparison sort algorithms**

- When the time complexity is considered, the **Quick sort** algorithm offers a more efficient solution compared to simple sorting algorithms.

**Notes:**

- Keep in mind that for some value cases, it might be necessary to use different sorting techniques.

**Textbook:**

Chapter 8, Sections 9.2 and 9.3

**CSE 310:**

Data Structures and Algorithms

**Instructor:**

For any question, feel free to ask.
Counting sort

Input: Array $A[1..n]$, $\text{length}(A) = n$
Output: Array $B[1..n]$
Temporary working storage: Array $C[1..n]$

Counting-Sort($A[1..n]$)
1. for $i$ from 1 to $n$ do $C[i] = 0$
2. for $j$ from 1 to $\text{length}(A)$ do $C[A[j]]++$
3. for $i$ from 1 to $n$ do $B[i] = C[i]$
4. for $i$ from 1 to $n$ do $B[i] = B[i] - C[i]$

Correctness:
Since $C[i]$ stores the number of elements less than or equal to $i$, we have $B[i] + C[i] = i$ for all $i$.

Running time:
The algorithm requires $O(n + \text{max}(A))$ time.

Counting sort is a stable sort, meaning that it preserves the relative order of equal elements.

Since some $x$ may not be different, need to decrement $C[A[j]]$ when placing an $A[j]$ into $B$. 


Counting sort: shaded

Exercise 9.2-1

Input: an array of numbers

Numbers with the same value appear in the same order as in A.

Counting sort: shaded

Exercise 9.2-1

Running time: depends on intermediate sorting algorithm

Correctness proof by induction on common partial sorted

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>88</td>
<td>88</td>
<td>88</td>
</tr>
<tr>
<td>B</td>
<td>77</td>
<td>77</td>
<td>77</td>
</tr>
<tr>
<td>C</td>
<td>66</td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>D</td>
<td>55</td>
<td>55</td>
<td>55</td>
</tr>
<tr>
<td>E</td>
<td>44</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>F</td>
<td>33</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>G</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>H</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

For i = 1 to n

Each of d[i] lower-order digits d[i-1…0] digit.

Input: A, array of elements

Example: sort records keyed by multiple fields

Header sort is stable

Digit sorting must be stable

A...and the numbers are sorted

1. A record from Z until last digit
2. A record to most least significant digit
3. A record to least most significant digit

With an stable sorting algorithm

Counting sort of d-digit Header sort

Counting sort
Sorting in Linear Time

Textbook, Chapter 9, Sections 9.2 and 9.3

CSCE310: Data Structures and Algorithms
www.cse.unl.edu/~choueiry/S01-310/

Berthe Y. Choueiry (Shu-we-ri)
Ferguson Hall, Room 104
choueiry@cse.unl.edu, Tel: (402)472-5444

Running time of Radixsort:

When each digit is in the range of 1 to k (k not too large)
Each Pass over n d-digit numbers is $\Theta(n+k)$
$d$ passes: $\Theta(d(n+k))$

Unlike Quicksort and Insertsort, Counting sort does not sort in place.

When space is at stake, use Quicksort.
Non-comparison sort algorithms

- Counting sort: assumes something about input $O(n)$, stable
- Radix sort, $\Theta(dn + kd)$
  When $d$ constant, $k = O(n) \Rightarrow$ linear time
- Bucket sort: assumes something about input $O(n)$

$O(n \lg n)$:
- Mergesort, heapsort: worst-case
- Quicksort: average-case

$\Omega(n \lg n)$:
- Mergesort, heapsort, quicksort

Interesting common property: sorted order is based only on comparisons between the input elements

$\rightarrow$ Comparison sorts algorithms

We can prove that: $\text{any comparison sort algorithm is in } \Omega(n \lg n)$  

(Section 1.1)
Counting sort

Input: Array $A[1 \ldots n]$, $\text{length}[A] = n$
Output: Array $B[1 \ldots n]$
Temporary working storage: Array $C[1 \ldots 2]$

```
COUNTING-SORT(A, B, k)
1   for i ← 1 to k
2     do $C[i] \leftarrow 0$
3   for j ← 1 to $\text{length}[A]$
4     do $C[A[j]] \leftarrow C[A[j]] + 1$
5     ▷ $C[i]$ now contains the number of elements equal to $i$.
6   for i ← 2 to k
7     do $C[i] \leftarrow C[i] + C[i - 1]$
8     ▷ $C[i]$ now contains the number of elements less than or equal to $i$.
9   for j ← $\text{length}[A]$ downto 1
10   do $B[C[A[j]]] \leftarrow A[j]$
11   $C[A[j]] \leftarrow C[A[j]] - 1$
```

Counting sort

Assumes that each of the $n$ input element is an integer in the range of 1 to $k$

When $k = O(n)$, counting sort is linear

Principle

- Determine for each input element $x$, the number of elements less than $x$
- Every element $x$ can be placed directly in its position in output array

Example

If $\exists 17$ elements less than $x$, $x$ must be in position 18

Slight modification for same value cases
Counting sort

\textbf{Counting-Sort}(A, B, k)
1 \hspace{1em} \textbf{for } i \leftarrow 1 \textbf{ to } k \\
2 \hspace{1em} \textbf{do } C[i] \leftarrow 0 \\
3 \hspace{1em} \textbf{for } j \leftarrow 1 \textbf{ to } \text{length}[A] \\
4 \hspace{2em} \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1 \\
5 \hspace{2em} \triangleright \ C[i] \text{ now contains the number of elements equal to } i. \\
6 \hspace{1em} \textbf{for } i \leftarrow 2 \textbf{ to } k \\
7 \hspace{2em} \textbf{do } C[i] \leftarrow C[i] + C[i - 1] \\
8 \hspace{2em} \triangleright \ C[i] \text{ now contains the number of elements less than or equal to } i. \\
9 \hspace{1em} \textbf{for } j \leftarrow \text{length}[A] \textbf{ downto } 1 \\
10 \hspace{2em} \textbf{do } B[C[A[j]]] \leftarrow A[j] \\
11 \hspace{2em} C[A[j]] \leftarrow C[A[j]] - 1

Lines 9–11: \(A[j]\) is place in correct position in \(B\)

correct final position for \(A[j]\) is \(C[A[j]]\)

Since some \(x\) may not be different,

need to decrement \(C[A[j]]\) when placing an \(A[j]\) into \(B\)

\[\]
**Counting sort**

```
COUNTING-SORT(A, B, k)
1 for i ← 1 to k
2 do C[i] ← 0
3 for j ← 1 to length[A]
4 do C[A[j]] ← C[A[j]] + 1
5 ▷ C[i] now contains the number of elements equal to i.
6 for i ← 2 to k
7 do C[i] ← C[i] + C[i - 1]
8 ▷ C[i] now contains the number of elements less than or equal to i.
9 for j ← length[A] downto 1
11 C[A[j]] ← C[A[j]] - 1
```

Lines 1–2: $O(k)$
Lines 3–4: $O(n)$
Lines 6–7: $O(k)$
Lines 9–11: $O(n)$

Counting sort: $O(k + n)$

Usually, used with $k = O(n)$, this in $O(n)$
Radix sort

Given numbers of $d$-digit, Radix-sort:
1. Starts with the least significant digit first
2. Sorts the numbers according this digit
   using a stable sorting algorithm
3. Moves to the next least-significant digit
4. Repeats from 2, until last digit $d$
5. .. and the numbers are sorted!

Digit sorting must be stable
Radix sort is stable

Example: sort records by dates (years, months, and days)
General use: sort records keyed by multiple fields

Counting sort: stable

Numbers with the same value appear in $B$ in same order as in $A$

Important in presence of satellite data

Exercise: 9.2-1
Try again in 9.2-3
**Running time:** of Radixsort

When each digit is in the range of 1 to $k$ ($k$ not too large)

Use Counting sort

Each pass over $n$ $d$-digit numbers is $\Theta(n + k)$

$d$-passes: $\Theta(dn + kd)$

When $d$ constant and $k = O(n)$, Radixsort is linear!!

Unlike Quicksort and Insertionsort, Countingsort does **not** sort in place

When space is at stake, use Quicksort

---

**Input:** $A$, array of $n$ elements, each of $d$ digits: 1 lowest-order digit, $d$ highest-order digit

For $i \leftarrow 1$ to $d$

*do* use a stable sort to sort array $A$ on digit $i$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>329</td>
<td>720</td>
<td>720</td>
<td>329</td>
</tr>
<tr>
<td>457</td>
<td>355</td>
<td>329</td>
<td>355</td>
</tr>
<tr>
<td>657</td>
<td>436</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td>839</td>
<td>457</td>
<td>839</td>
<td>457</td>
</tr>
<tr>
<td>436</td>
<td>657</td>
<td>355</td>
<td>657</td>
</tr>
<tr>
<td>720</td>
<td>329</td>
<td>457</td>
<td>720</td>
</tr>
<tr>
<td>355</td>
<td>839</td>
<td>657</td>
<td>839</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Correctness: proof by induction on column being sorted

**Running time:** depends on intermediate sorting algorithm

Exercise: 9.3-1