COMP2521 24T2
Sorting Algorithms (IV)
Non-Comparison-Based Sorting Algorithms

Sim Mautner
cs2521@cse.unsw.edu.au

$n \log n$ lower bound
radix sort

Slides adapted from those by Kevin Luxa 2521 24T1
All of the sorting algorithms so far have been \textit{comparison-based} sorts. It can be shown that these algorithms require $\Omega(n \log n)$ comparisons. That is, they require at least $kn \log n$ comparisons for some constant $k$. Why?
Suppose we need to sort 3 items.

Obviously, one comparison is not sufficient to sort them.
Suppose we need to sort 3 items.

Even two comparisons are not sufficient to sort them. Why?
If we have 3 items, there are $3! = 6$ ways to order them:

```
  ⬜ ⬜ ▲  ▲ ▬ ▬
  ▬ ▬ ⬜ ▲ ▲  ▬
  ▲ ▬ ▬ ⬜ ▲ ▲
```

Assuming items are unique, one of these permutations is in sorted order.
Suppose we performed the following comparisons:

\[
\begin{align*}
&\text{Red} < \text{Blue} \\
&\text{Blue} < \text{Green}
\end{align*}
\]

Four combinations of results are possible:

\[(\text{true, true}), (\text{true, false}), (\text{false, true}), (\text{false, false})\]
The two comparisons create four groups, and each permutation of items belongs to one of these groups.

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>⬤ &lt; ⬤</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⬤ &lt; ▲</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

- The first comparison separates the items into two groups based on the truth value of the comparison.
- The second comparison further separates the items within each group based on the second comparison's truth value.
Mathematically,

If we have 3 items, then there are $3! = 6$ ways to order them. In other words, 6 possible permutations.

But if we only perform 2 comparisons, then there are only $2^2 = 4$ groups, so at least one group will contain more than one permutation.

We need at least 3 comparisons, because this creates $2^3 = 8$ groups, so each permutation can belong in its own group.
If we have $n$ items, then there are $n!$ permutations.

If we perform $k$ comparisons, that creates up to $2^k$ groups.

So given $n$ items, we must perform enough comparisons $k$ such that $2^k \geq n!$
So given $n$ items, we must perform enough comparisons $k$ such that

$$2^k \geq n!$$

Taking the $\log_2$ of both sides gives

$$\log_2 2^k \geq \log_2 n!$$

Since $\log_2 2^k = k$, we get

$$k \geq \log_2 n!$$

Using Stirling’s approximation, we get

$$k \geq n \log_2 n - n \log_2 e + O(\log_2 n)$$

Removing lower-order terms gives

$$k = \Omega(n \log_2 n)$$
Therefore:

The theoretical lower bound on worst-case execution time for comparison-based sorts is $\Omega(n \log n)$. 
Non-Comparison-Based Sorting

If we aren’t limited to just comparing keys, we can achieve better than $O(n \log n)$ worst-case time.

Non-comparison-based sorting algorithms exploit specific properties of the data to sort it.
Radix sort is a non-comparison-based sorting algorithm.

It requires us to be able to decompose our keys into individual symbols (digits, characters, bits, etc.), for example:

- The key 372 is decomposed into (3, 7, 2)
- The key “sydney” is decomposed into (‘s’, ‘y’, ‘d’, ‘n’, ‘e’, ‘y’)

Formally, each key $k$ is decomposed into a tuple $(k_1, k_2, k_3, ..., k_m)$. 
Ideally, the range of possible symbols is reasonably small, for example:

- Numeric: 0-9
- Alphabetic: a-z

The number of possible symbols is known as the *radix*, and is denoted by $R$.

- Numeric: $R = 10$ (for base 10)
- Alphabetic: $R = 26$

If the keys have different lengths, pad them with a suitable symbol, for example:

- Numeric: 123, 015, 007
- Alphabetic: “abc”, “zz”, “t”
Radix Sort

Method:

- Perform stable sort on $k_m$
- Perform stable sort on $k_{m-1}$
- ...
- Perform stable sort on $k_1$

Example:

cat  ace  dog  cog  key  buy
ace  dog  cog  key  buy
stable sort on third char
stable sort on second char
stable sort on first char

ace  dog  cog  key  buy
stable sort on first char

ace  dog  cog  key  buy
stable sort on second char

ace  dog  cog  key  buy
stable sort on third char
radixSort(A):

**Input:** array A of keys where each key consists of m symbols from an "alphabet"

initialise R buckets // one for each symbol

for i from m down to 1:
    empty all buckets
    for each key in A:
        append key to bucket key[i]

clear A
for each bucket (in order):
    for each key in bucket:
        append key to A
Assume alphabet is {‘a’, ‘b’, ‘c’}, so $R = 3$.

We want to sort the array:

[“abc”, “cab”, “baa”, “a”, “ca”]

First, pad keys with blank characters:

[“abc”, “cab”, “baa”, “a␣␣”, “ca␣”]

Each key contains three characters, so $m = 3$. 
Radix Sort

Example

Array:

```
"abc"  "cab"  "baa"  "a"  "ca"
```

Buckets:

```
Δ   a   b   c
```
Radix Sort

Example

Array:

```
“abc”  “cab”  “baa”  “a”  “ca”
```

Buckets:

```
“a”  “baa”
“ca”

“b”  “cab”

“c”  “abc”
```
Radix Sort

Example

Array:

```
"a\_\_\_", "ca\_", "baa", "cab", "abc"
```

Buckets:

```
\_\
"a\_\_\_", "ca\_"

a
"baa"

b
"cab"

c
"abc"
```
Radix Sort

Example

Array:

“a” “ca” “baa” “cab” “abc”

Buckets:
Radix Sort

Example

Array:

```
“a
ca
baa
cab
abc”
```

Buckets:

```
“a
ca
baa
cab”
```

```
“b
abc”
```

```
c
```
Radix Sort

Example

Array:

```
"a\_\_\_"  "ca\_\_"  "baa"  "cab"  "abc"
```

Buckets:

```
\_
"a\_\_\_"

\_
"ca\_\_"
  "baa"
  "cab"

\_
"abc"

\_
"c"
```
Radix Sort

Example

Array:

“a” “ca” “baa” “cab” “abc”

Buckets:

□ □ □ □
Radix Sort

Example

Array:

```
"a_
ca_
"baa"
"cab"
"abc"
```

Buckets:

```
\_ 
a
"a_
"abc"

b
"baa"

\_ 
c
"ca_
"cab"
```
Radix Sort
Example

Array:

```
“aabc”  “abc”  “baa”  “ca”  “cab”
```

Buckets:

```
□
□
□
□
```

```
a
“aabc”
“abc”

b
“baa”

c
“ca”
“cab”
```
Radix Sort

Example

Array:

“a”  “abc”  “baa”  “ca”  “cab”

Buckets:

∅  a  b  c
Analysis:

- Array contains \( n \) keys
- Each key contains \( m \) symbols
- Radix sort uses \( R \) buckets
- A single stable sort runs in time \( O(n + R) \)
- Radix sort uses stable sort \( m \) times

Hence, time complexity for radix sort is \( O(m(n + R)) \).

- \( \approx O(mn) \), assuming \( R \) is small

Therefore, radix sort performs better than comparison-based sorting algorithms:

- When keys are short (i.e., \( m \) is small) and arrays are large (i.e., \( n \) is large)
Radix Sort

Properties

**Stable**
All sub Sorts performed are stable

**Non-adaptive**
Same steps performed, regardless of sortedness

**Not in-place**
Uses $O(R + n)$ additional space for buckets and storing keys in buckets
Other Non-Comparison-Based Sorts

- Bucket sort
- MSD Radix Sort
  - The version shown was LSD
- Key-indexed counting sort
- ...and others
https://forms.office.com/r/riGKCze1cQ