Quick Sort

• Another divide-and-conquer algorithm.

• Pick any **pivot** element v.
• Partition the array into elements
  • $x \leq v$ and $x \geq v$.
• Recursively sort the partitions, then concatenate them.

| 34 | 8  | 64 | 2  | 51 | 32 | 21 | 1  |
Quick Sort

• Another divide-and-conquer algorithm.

• Pick any **pivot** element v.
• Partition the array into elements
  • $x \leq v$ and $x \geq v$.
• Recursively sort the partitions, then concatenate them.

```
34  8  64  2  51  32  21  1
```

```
21
```

$2 \text{ v}$
Quick Sort

• Another divide-and-conquer algorithm.

• Pick any **pivot** element v.
• Partition the array into elements
  • \( x \leq v \) and \( x \geq v \).
• Recursively sort the partitions, then concatenate them.

\[
\begin{array}{cccccccc}
34 & 8 & 64 & 2 & 51 & 32 & 21 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
8 & 2 & 1 \\
\end{array}
\]

\[
\begin{array}{c}
x \leq v \\
\end{array}
\]

\[
\begin{array}{c}
v \\
\end{array}
\]

\[
\begin{array}{c}
21 \\
\end{array}
\]
Quick Sort

• Another divide-and-conquer algorithm.

• Pick any **pivot** element $v$.
• Partition the array into elements
  • $x \leq v$ and $x \geq v$.
• Recursively sort the partitions, then concatenate them.
Quick Sort

- Another divide-and-conquer algorithm.
- Pick any **pivot** element $v$.
- Partition the array into elements
  - $x \leq v$ and $x \geq v$.
- Recursively sort the partitions, then concatenate them.
Quick Sort

• Another divide-and-conquer algorithm.

• Pick any **pivot** element v.
• Partition the array into elements
  • \( x \leq v \) and \( x \geq v \).
• Recursively sort the partitions, then concatenate them.
Quick Sort

34  8  64  2  51  32  21  1

8  2  1  21

34  64  51  32
Quick Sort

34  8  64  2  51  32  21  1

8  2  1  21

34  64  51  32

1  2  8  34  32  51  64
Quick Sort

```
34  8  64  2  51  32  21  1
8  2  1
21
34  64  51  32
1  2  8
34  32  51  64
32  34
```
Quick Sort

34 8 64 2 51 32 21 1

8 2 1 21

34 64 51 32

1 2 8

32 34 51 64
Quick Sort

34  8  64  2  51  32  21  1

8   2   1                        21

34  64  51  32

1  2  8                                  32  34  51  64
Quick Sort

34 8 64 2 51 32 21 1

8 2 1 21 32 34 51 64

1 2 8
Quick Sort

34  8  64  2  51  32  21  1

1  2  8  21  32  34  51  64
Quick Sort

34 8 64 2 51 32 21 1

1 2 8 21 32 34 51 64
Quick Sort

1  2  8  21  32  34  51  64
Quick Sort

| 1 | 2 | 8 | 21 | 32 | 34 | 51 | 64 |

• How do we partition the array efficiently (in place)?

• How do we pick a pivot element?
  • Running time performance on quick sort depends on our choice.
  • Bad choice leads to $\Theta(N^2)$ running time.
Partitioning the Array

• We don’t want to use any extra space. Need to partition the array in place.

• Use swaps to push all elements $x \leq v$ to the left and elements $x \geq v$ to the right.
Partitioning the Array

• We don’t want to use any extra space. Need to partition the array in place.

• Use swaps to push all elements $x \leq v$ to the left and elements $x \geq v$ to the right.

Move the pivot to the end.
Partitioning the Array

• While True:
  • Move i right until we find an element \( \text{array}[i] \geq v \)
  • Move j left until we find an element \( \text{array}[j] \leq v \).
  • if \( i \geq j \) break
  • Swap \( \text{array}[i] \) and \( \text{array}[j] \).
Partitioning the Array

- While True:
  - Move i right until we find an element $array[i] \geq v$
  - Move j left until we find an element $array[j] \leq v$.
  - if $i \geq j$ break
  - Swap $array[i]$ and $array[j]$.

```
| 21 | 8  | 64 | 2  | 51 | 1  | 34 | 32 |
```

• i

• j
Partitioning the Array

• While True:
  • Move i right until we find an element array[i] ≥ v
  • Move j left until we find an element array[j] ≤ v.
  • if i ≥ j break
  • Swap array[i] and array[j].
Partitioning the Array

- While True:
  - Move i right until we find an element \( \text{array}[i] \geq v \)
  - Move j left until we find an element \( \text{array}[j] \leq v \).
  - if \( i \geq j \) break
  - Swap \( \text{array}[i] \) and \( \text{array}[j] \).
Partitioning the Array

• While True:
  • Move i right until we find an element $array[i] \geq v$
  • Move j left until we find an element $array[j] \leq v$.
  • if $i \geq j$ break
  • Swap $array[i]$ and $array[j]$.
Partitioning the Array

• While True:
  • Move i right until we find an element array[i] ≥ v
  • Move j left until we find an element array[j] ≤ v.
  • if i ≥ j break
  • Swap array[i] and array[j].
Partitioning the Array

- While True:
  - Move i right until we find an element $\text{array}[i] \geq v$
  - Move j left until we find an element $\text{array}[j] \leq v$
  - if $i \geq j$ break
  - Swap $\text{array}[i]$ and $\text{array}[j]$.

![Partitioning Array Example]

```plaintext
21 8 1 2 51 64 34 32
```

\( j \) \( i \)
Partitioning the Array

• While True:
  • Move i right until we find an element array[i] ≥ v
  • Move j left until we find an element array[j] ≤ v.
  • if i ≥ j break
  • Swap array[i] and array[j].
• Swap array[i] with v.

• i points to a value greater than the pivot.
Partitioning the Array

• While True:
  • Move i right until we find an element $\text{array}[i] \geq v$
  • Move j left until we find an element $\text{array}[j] \leq v$.
  • if $i \geq j$ break
  • Swap $\text{array}[i]$ and $\text{array}[j]$.
• Swap $\text{array}[i]$ with $v$.

- i points to a value greater than the pivot.
public static void quicksort(Integer[] a, int left, int right) {

    if (right > left) {
        int v = find_pivot_index(a, left, right);
        int i = 0;  int j = right - 1;

        // move pivot to the end
        Integer tmp = a[v]; a[v] = a[right]; a[right] = tmp;

        while (true) {  // partition
            while (a[++i] < v) {};
            while (a[++j] > v) {};
            if (i >= j) break;
            tmp = a[i]; a[i] = a[j]; a[j] = tmp;
        }

        // move pivot back
        tmp = a[i]; a[i] = a[right]; a[right] = tmp;
        // recursively sort both partitions
        quicksort(a, left, i-1);  quicksort(a, i+1, right);
    }
}
Partitioning the Array

```java
public static void quicksort(Integer[] a, int left, int right) {
    if (right > left) {
        int v = find_pivot_index(a, left, right);
        int i = 0;  int j = right - 1;

        // move pivot to the end
        Integer tmp = a[v];  a[v] = a[right];  a[right] = tmp;

        while (true) {  // partition
            while (a[++i] < v) {};
            while (a[++j] > v) {};
            if (i >= j) break;
            tmp = a[i];  a[i] = a[j];  a[j] = tmp;
        }

        // move pivot back
        tmp = a[i];  a[i] = a[right];  a[right] = tmp;

        // recursively sort both partitions
        quicksort(a, left, i-1);  quicksort(a, i+1, right);
    }
}
```

Quick Sort: Worst Case

• Running time depends on the how the pivot partitions the array.

• Worst case: Pivot is always the smallest or largest element. One of the partitions is empty!
Quick Sort: Worst Case

• Running time depends on the how the pivot partitions the array.

• Worst case: Pivot is always the smallest or largest element. One of the partitions is empty!

```
| 34 | 8 | 64 | 2 | 51 | 32 | 21 | 1 |
```

```
| 1 | 34 | 8 | 64 | 2 | 51 | 32 | 21 |
```
Quick Sort: Worst Case

- Running time depends on the how the pivot partitions the array.
- Worst case: Pivot is always the smallest or largest element. One of the partitions is empty!
Quick Sort: Worst Case

```
34  8  64  2  51  32  21  1
|
1  |
|
2  |
|
 ... |
```

T(1) = 1
Quick Sort: Worst Case

34  8  64  2  51  32  21  1

1  34  8  64  2  51  32  21

2  34  8  64  51  32  21

\[ T(2) = T(1) + 2 \]

\[ T(1) = 1 \]
Quick Sort: Worst Case

34 8 64 2 51 32 21 1

1 34 8 64 2 51 32 21

2 34 8 64 51 32 21

\[ T(N-2) = T(N-3) + (N-2) \]

\[ T(2) = T(1) + 2 \]

\[ T(1) = 1 \]

Time for partitioning
Quick Sort: Worst Case

\[ T(1) = 1 \]
\[ T(2) = T(1) + 2 \]
\[ T(N-1) = T(N-2) + (N-1) \]
\[ T(N-2) = T(N-3) + (N-2) \]
\[ T(2) = T(1) + 2 \]
Quick Sort: Worst Case

\[ T(N) = T(N-1) + N \]

\[ T(N-1) = T(N-2) + (N-1) \]

\[ T(N-2) = T(N-3) + (N-2) \]

\[ T(2) = T(1) + 2 \]

\[ T(1) = 1 \]
Quick Sort: Worst Case

\[ T(N) = T(N - 1) + N \]
Quick Sort: Worst Case

\[ T(N) = T(N - 1) + N \]

\[ = T(N - 2) + (N - 1) + N \]
Quick Sort: Worst Case

\[ T(N) = T(N - 1) + N \]

\[ = T(N - 2) + (N - 1) + N \]

\[ = T(N - k) + (N - (k - 1)) + \cdots + (N - 1) + N \]

\[ \vdots \]

\[ = T(1) + 2 + 3 + \cdots + (N - 1) + N \]
Quick Sort: Worst Case

\[ T(N) = T(N - 1) + N \]

\[ = T(N - 2) + (N - 1) + N \]

\[ = T(N - k) + (N - (k - 1)) + \cdots + (N - 1) + N \]

\[ \vdots \]

\[ = T(1) + 2 + 3 + \cdots + (N - 1) + N \]

\[ = 1 + \sum_{i=2}^{N} i = \sum_{i=1}^{N} i \]
Quick Sort: Worst Case

\[ T(N) = T(N - 1) + N \]
\[ = T(N - 2) + (N - 1) + N \]
\[ = T(N - k) + (N - (k - 1)) + \cdots + (N - 1) + N \]
\[ = T(1) + 2 + 3 + \cdots + (N - 1) + N \]
\[ = 1 + \sum_{i=2}^{N} i = \sum_{i=1}^{N} i \]
\[ = N \frac{N + 1}{2} = \Theta(N^2) \]
Quick Sort: Best Case

• Best case: Pivot is always the median element. Both partitions have about the same size.

| 34 | 8  | 64 | 2  | 51 | 32 | 21 | 1  |
Quick Sort: Best Case

- Best case: Pivot is always the median element. Both partitions have about the same size.

```plaintext
34  8  64  2  51  32 21  1

8  2  1

21

34  64  51 32
```
Quick Sort: Best Case

- Best case: Pivot is always the median element. Both partitions have about the same size.
Quick Sort: Best Case

- Best case: Pivot is always the median element. Both partitions have about the same size.
Quick Sort: Best Case

- Best case: Pivot is always the median element.
  Both partitions have about the same size.

\[ T(N) = 2 \ T(N/2) + N \]

(we ignore the pivot element, so this overestimates the running time slightly)
Quick Sort: Best Case

- Best case: Pivot is always the median element.
  Both partitions have about the same size.

\[ T(N) = 2 \cdot T(N/2) + N \]

\[
\begin{array}{cccccc}
34 & 8 & 64 & 2 & 51 & 32 & 21 & 1 \\
\end{array}
\]

\[ T(N/2) = 2 \cdot T(N/4) + N/2 \]

\[
\begin{array}{cccc}
8 & 2 & 1 \\
21 \\
34 & 64 & 51 & 32 \\
1 & 2 & 8 \\
34 & 32 & 51 & 64 \\
\end{array}
\]

(we ignore the pivot element, so this overestimates the running time slightly)
Quick Sort: Best Case

• Best case: Pivot is always the median element.
  Both partitions have about the same size.

\[
T(N) = 2 \; T(N/2) + N
\]

\[
T(N/2) = 2 \; T(N/4) + N/2
\]

\[
T(1) = 1
\]

(we ignore the pivot element, so this overestimates the running time slightly)
Quick Sort: Best Case

\[ T(N) = 2 \cdot T\left(\frac{N}{2}\right) + N \]

(note that this is the same analysis as for Merge Sort)
Quick Sort: Best Case

\[ T(N) = 2 \cdot T\left(\frac{N}{2}\right) + N \]

\[ = 2 \cdot (2 \cdot T\left(\frac{N}{4}\right) + \frac{N}{2}) + N \]

\[ = 4 \cdot T\left(\frac{N}{4}\right) + N + N \]

(note that this is the same analysis as for Merge Sort)
Quick Sort: Best Case

\[ T(N) = 2 \cdot T\left(\frac{N}{2}\right) + N \]

\[ = 2 \cdot \left(2 \cdot T\left(\frac{N}{4}\right) + \frac{N}{2}\right) + N \]

\[ = 4 \cdot T\left(\frac{N}{4}\right) + N + N \]

\[ = 2^k \cdot T\left(\frac{N}{2^k}\right) + k \cdot N \]

(note that this is the same analysis as for Merge Sort)
Quick Sort: Best Case

\[
T(N) = 2 \cdot T\left(\frac{N}{2}\right) + N
\]

\[
= 2 \cdot \left(2 \cdot T\left(\frac{N}{4}\right) + \frac{N}{2}\right) + N = 4 \cdot T\left(\frac{N}{4}\right) + N + N
\]

\[
= 2^k \cdot T\left(\frac{N}{2^k}\right) + k \cdot N \quad \text{assume } k = \log N
\]
Quick Sort: Best Case

\[ T(N) = 2 \cdot T\left(\frac{N}{2}\right) + N \]

\[ = 2 \cdot (2 \cdot T\left(\frac{N}{4}\right) + \frac{N}{2}) + N \]

\[ = 4 \cdot T\left(\frac{N}{4}\right) + N + N \]

\[ = 2^k \cdot T\left(\frac{N}{2^k}\right) + k \cdot N \]

\[ = N \cdot T(1) + \log N \cdot N \]

(note that this is the same analysis as for Merge Sort)
Quick Sort: Best Case

\[ T(N) = 2 \cdot T\left( \frac{N}{2} \right) + N \]

\[ = 2 \cdot \left( 2 \cdot T\left( \frac{N}{4} \right) + \frac{N}{2} \right) + N \]

\[ = 4 \cdot T\left( \frac{N}{4} \right) + N + N \]

\[ = 2^k \cdot T\left( \frac{N}{2^k} \right) + k \cdot N \]

assume \( k = \log N \)

\[ = N \cdot T(1) + \log N \cdot N \]

\[ = N + N \cdot \log N = \Theta(N \log N) \]

(note that this is the same analysis as for Merge Sort)
Choosing the Pivot
Choosing the Pivot

• Ideally we want to choose the median in each partition, but we don’t know where it is!
Choosing the Pivot

• Ideally we want to choose the median in each partition, but we don’t know where it is!

• Computing the pivot should be a constant time operation.
Choosing the Pivot

- Ideally we want to choose the median in each partition, but we don’t know where it is!

- Computing the pivot should be a constant time operation.

- Choosing the element at the beginning/end/middle is a terrible idea!
  Better: Choose a random element.
Choosing the Pivot

• Ideally we want to choose the median in each partition, but we don’t know where it is!

• Computing the pivot should be a constant time operation.

• Choosing the element at the beginning/end/middle is a terrible idea!
  Better: Choose a random element.

• Good approximation for median: “Median-of-three”
Choosing a Pivot: Median of Three

Choose the median of array[0], array[n/2] and array[n/2].
Choosing a Pivot: Median of Three

Choose the median of array[0], array[n]m and array[n/2].
Choosing a Pivot: Median of Three

Choose the median of array[0], array[n]m and array[n/2].
Choosing a Pivot: Median of Three

Choose the median of array[0], array[n]m and array[n/2].
public static int find_pivot_index(Integer[] a, int left, int right) {
    int center = (left + right) / 2;
    Integer tmp;
    if (a[center] < a[left]) {
        tmp = a[center]; a[center] = a[left]; a[left] = tmp;
    }
    if (a[right] < a[left]) {
        tmp = a[right]; a[right] = a[left]; a[left] = tmp;
    }
    if (a[right] < a[center]) {
        tmp = a[right]; a[right] = a[center]; a[center] = tmp;
    }
    return center;
}
Analyzing Quick Sort

• Worst case running time: $\Theta(N^2)$

• Best and average case (random pivot): $\Theta(N \log N)$

• Is QuickSort stable?

• Space requirement?
Analyzing Quick Sort

• Worst case running time: $\Theta(N^2)$

• Best and average case (random pivot): $\Theta(N \log N)$

• Is QuickSort stable?
  No. Partitioning can change order of elements. (but can make QuickSort stable).

• Space requirement?
Analyzing Quick Sort

• Worst case running time: $\Theta(N^2)$

• Best and average case (random pivot): $\Theta(N \log N)$

• Is QuickSort stable?
   No. Partitioning can change order of elements. (but can make QuickSort stable).

• Space requirement?
   In-place $O(1)$, but the method activation stack grows with the running time. $O(N)$
### Comparison-Based Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$T_{\text{Worst}}$</th>
<th>$T_{\text{Best}}$</th>
<th>$T_{\text{Avg}}$</th>
<th>Space</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$\Theta(N^2)$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(N^2)$</td>
<td>$O(1)$</td>
<td>✓</td>
</tr>
<tr>
<td>Shell Sort</td>
<td>$\Theta(N^{3/2})^*$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(N^{3/2})^*$</td>
<td>$O(1)$</td>
<td>✗</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$\Theta(N\log N)$</td>
<td>$\Theta(N\log N)$</td>
<td>$\Theta(N\log N)$</td>
<td>$O(1)$</td>
<td>✗</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$\Theta(N\log N)$</td>
<td>$\Theta(N\log N)$</td>
<td>$\Theta(N\log N)$</td>
<td>$O(N)$</td>
<td>✓</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$\Theta(N^2)$</td>
<td>$\Theta(N\log N)$</td>
<td>$\Theta(N\log N)$</td>
<td>$O(N)$</td>
<td>✗</td>
</tr>
</tbody>
</table>

*depends on increment sequence

Gray entries: not shown in class
Comparison-Based Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst Case</th>
<th>Best Case</th>
<th>Avg Case</th>
<th>Space</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shell Sort</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>Θ(NlogN)</td>
<td>Θ(NlogN)</td>
<td>Θ(NlogN)</td>
<td>O(1)</td>
<td>x</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>Θ(NlogN)</td>
<td>Θ(NlogN)</td>
<td>Θ(NlogN)</td>
<td>O(N)</td>
<td>✓</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>Θ(N^2)</td>
<td>Θ(NlogN)</td>
<td>Θ(NlogN)</td>
<td>O(N)</td>
<td>x</td>
</tr>
</tbody>
</table>

Ω(Nlog N) worst case lower bound on comparison based general sorting!
Can we do better if we make some assumptions?

*depends on increment sequence

gray entries: not shown in class
Bucket Sort

• Assume we know there are $M$ possible values.

• Keep an array $\text{count}$ of length $M$.

• Scan through the input array $A$ and for each $i$ increment $\text{count}[A_i]$.
Bucket Sort

• Assume we know there are $M$ possible values.

• Keep an array count of length $M$.

• Scan through the input array $A$ and for each $i$ increment $\text{count}[A_i]$. 

\[
\begin{array}{cccccccc}
A & 1 & 8 & 2 & 3 & 2 & 4 & 6 & 1 \\
\text{count} & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9
\end{array}
\]
Bucket Sort

• Assume we know there are $M$ possible values.

• Keep an array `count` of length $M$.

• Scan through the input array $A$ and for each $i$ increment `count[$A_i]$`.

```
A = [1, 8, 2, 3, 2, 4, 6, 1]
count = [0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
```
Bucket Sort

• Assume we know there are $M$ possible values.

• Keep an array $\text{count}$ of length $M$.

• Scan through the input array $A$ and for each $i$ increment $\text{count}[A_i]$.

\begin{center}
\begin{array}{cccccccccc}
A & 1 & 8 & 2 & 3 & 2 & 4 & 6 & 1 \\
\text{count} & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\end{center}

\begin{center}
\begin{array}{cccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\end{center}
Bucket Sort

• Assume we know there are $M$ possible values.

• Keep an array `count` of length $M$.

• Scan through the input array $A$ and for each $i$ increment `count[$A_i$].`
Bucket Sort

- Assume we know there are $M$ possible values.
- Keep an array `count` of length $M$.
- Scan through the input array $A$ and for each $i$ increment $\text{count}[A_i]$.

$$
\begin{array}{cccccccc}
A & 1 & 8 & 2 & 3 & 2 & 4 & 6 & 1 \\
\text{count} & 0 & 1 & 2 & 1 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
$$
Bucket Sort

• Assume we know there are $M$ possible values.

• Keep an array $\text{count}$ of length $M$.

• Scan through the input array $A$ and for each $i$ increment $\text{count}[A_i]$.

\[ A = \begin{bmatrix} 1 & 8 & 2 & 3 & 2 & 4 & 6 & 1 \end{bmatrix} \]

\[ \text{count} = \begin{bmatrix} 0 & 1 & 2 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \]
Bucket Sort

• Assume we know there are $M$ possible values.

• Keep an array **count** of length $M$.

• Scan through the input array $A$ and for each $i$ increment **count**[$A_i$].
Bucket Sort

• Assume we know there are $M$ possible values.

• Keep an array `count` of length $M$.

• Scan through the input array $A$ and for each $i$ increment `count[$A_i]$`.

```
<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Bucket Sort

- Assume we know there are $M$ possible values.

- Keep an array \texttt{count} of length $M$.

- Scan through the input array $A$ and for each $i$ increment $\texttt{count}[A_i]$. 

\begin{align*}
A & \begin{array}{cccccccc}
1 & 8 & 2 & 3 & 2 & 4 & 6 & 1
\end{array} \\
\text{count} & \begin{array}{cccccccc}
0 & 2 & 2 & 1 & 1 & 0 & 1 & 0 & 1 & 0
\end{array}
\end{align*}
Bucket Sort

- Then iterate through count. For each $i$ write $\text{count}[i]$ copies of $i$ to $A$. 

<table>
<thead>
<tr>
<th>count</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bucket Sort

- Then iterate through `count`. For each `i` write `count[i]` copies of `i` to `A`.

```
A
1 1

count
0 2 2 1 1 0 1 0 1 0
0 1 2 3 4 5 6 7 8 9
```
Bucket Sort

- Then iterate through count. For each i write count[i] copies of i to A.
Bucket Sort

• Then iterate through `count`. For each `i` write `count[i]` copies of `i` to `A`.
Bucket Sort

• Then iterate through \texttt{count}. For each \texttt{i} write \texttt{count[i]} copies of \texttt{i} to \texttt{A}.
Bucket Sort

• Then iterate through \texttt{count}. For each \texttt{i} write \texttt{count[i]} copies of \texttt{i} to \texttt{A}.

\begin{center}
\begin{tabular}{ccccccc}
A & 1 & 1 & 2 & 2 & 3 & 4 \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{cccccccccccc}
\texttt{count} & 0 & 2 & 2 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \_ & \_ \\
\end{tabular}
\end{center}
Bucket Sort

- Then iterate through $\text{count}$. For each $i$ write $\text{count}[i]$ copies of $i$ to $A$. 

<table>
<thead>
<tr>
<th></th>
<th>count</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

$A$: 1 1 2 2 3 4 6
Bucket Sort

- Then iterate through \texttt{count}. For each \texttt{i} write \texttt{count[i]} copies of \texttt{i} to \texttt{A}.

\begin{itemize}
\item \texttt{A}:
\begin{tabular}{c|c|c|c|c|c|c|c}
  0 & 1 & 1 & 2 & 2 & 3 & 4 & 6 \\
\end{tabular}
\item \texttt{count}:
\begin{tabular}{c|c|c|c|c|c|c|c|c|c|c}
  0 & 1 & 2 & 2 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
\end{tabular}
\end{itemize}
Bucket Sort

• Then iterate through `count`. For each `i` write `count[i]` copies of `i` to `A`.

```
A  1 1 2 2 3 4 6 8
count 0 2 2 1 1 0 1 0 1
      0 1 2 3 4 5 6 7 8 9
```
Bucket Sort

- Then iterate through \texttt{count}. For each $i$ write \texttt{count}[i] copies of $i$ to A.
Bucket Sort

- Then iterate through `count`. For each `i`, write `count[i]` copies of `i` to `A`. 

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 2 & 2 & 3 & 4 & 6 & 8 \\
0 & 2 & 2 & 1 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]
Bucket Sort

- Then iterate through `count`. For each `i` write `count[i]` copies of `i` to `A`.

Total time for Bucket Sort: $O(N + M)$
Radix Sort

• Generalization of Bucket sort for Large M.

• Assume M contains all base b numbers up to $b^p-1$ (e.g. all base-10 integers up to $10^3$)

• Do p passes over the data, using Bucket Sort for each digit.

• Bucket sort is stable!

064 008 216 512 027 729 000 001 343 125
Radix Sort

- Bucket sort according to least significant digit.
Radix Sort

- Bucket sort according to least significant digit.
Radix Sort

064 008 216 512 027 729 000 001 343 125

• Bucket sort according to least significant digit.
Radix Sort

064 008 216 512 027 729 000 001 343 125

• Bucket sort according to least significant digit.
Radix Sort

- Bucket sort according to least significant digit.
Radix Sort

- Bucket sort according to least significant digit.
Radix Sort

- Bucket sort according to least significant digit.
Radix Sort

064 008 216 512 027 729 000 001 343 125

- Bucket sort according to least significant digit.
Radix Sort

• Bucket sort according to least significant digit.
Radix Sort

064 008 216 512 027 729 000 001 343 125

• Bucket sort according to least significant digit.
Radix Sort

000 001 512 343 064 125 216 027 008 729

- read off new sequence
Radix Sort

000 001 512 343 064 125 216 027 008 729

- Bucket sort according to second-least significant digit.
Radix Sort

000 001 512 343 064 125 216 027 008 729

- Bucket sort according to second-least significant digit.
Radix Sort

- Bucket sort according to second-least significant digit.
Radix Sort

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

- Bucket sort according to second-least significant digit.
Radix Sort

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

- Bucket sort according to second-least significant digit.
Radix Sort

000 001 512 343 064 125 216 027 008 729

- Bucket sort according to second-least significant digit.
Radix Sort

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

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

- Bucket sort according to second-least significant digit.
Radix Sort

0 000 001 512 343 064 125 216 027 008 729

- Bucket sort according to second-least significant digit.
Radix Sort

000 001 512 343 064 125 216 027 008 729

- Bucket sort according to second-least significant digit.
Radix Sort

000 001 512 343 064 125 216 027 008 729

- Bucket sort according to second-least significant digit.
Radix Sort

000  001   008  512  216  125  027  729  343  064

0  000  001  008
1  512  216
2  125  027  729  • read off new sequence
3
4  343
5
6  064
7
8
9
Radix Sort

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000 001 008 512 216 125 027 729 343 064</td>
</tr>
</tbody>
</table>

- Bucket sort according to third-least significant digit.
**Radix Sort**

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>008</th>
<th>027</th>
<th>064</th>
<th>125</th>
<th>216</th>
<th>343</th>
<th>512</th>
<th>729</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>001</td>
<td>008</td>
<td>027</td>
<td>064</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>125</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>216</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>343</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>512</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>729</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- read off new sequence
- Sorted!
Radix Sort

000 001 008 027 064 125 216 343 512 729

| 0 | 000 001 008 027 064 |
| 1 | 125               |
| 2 | 216               |
| 3 | 343               |
| 4 |                   |
| 5 | 512               |
| 6 |                   |
| 7 | 729               |
| 8 |                   |
| 9 |                   |

- read off new sequence
- Sorted!

Each Bucket Sort: $O(N+b)$
There are $p$ Bucket Sorts, so total time for Radix sort: $O(p \cdot (N+b))$
## Sorting Strings with Radix Sort

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
<td>a</td>
<td>dna</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>b</td>
<td>bob nib</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>c</td>
<td>sic</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>d</td>
<td>bad fad bid</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>e</td>
<td>die pie pre</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>of by in</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>