Sorting algorithms
Properties of sorting algorithms

A sorting algorithm is 

**Comparison based** If it works by pairwise key comparisons.

**In place** If only a constant number of elements of the input array are ever stored outside the array.

**Stable** If it preserve the relative order of elements with equal keys.

The space complexity of a sorting algorithm is the amount of memory it uses in addition to the input array.

- **Insertion Sort** $\Theta(1)$.
- **Merge Sort** $\Theta(n)$.
- **Quicksort** $\Theta(n)$ (worst case).
Lower bound for comparison based sorting algorithms

- All possible runs of a comparison based sorting algorithm can be modeled by a decision tree.
- The height of the decision tree is a lower bound on the time complexity of the algorithm.
- The decision tree of insertion sort for $n = 3$ is shown below.
Lower bound for comparison based sorting algorithms

**Theorem**

*For every comparison based sorting algorithm, the decision tree for n elements has height $\Omega(n \log n)$.**

**Proof.**

- The number of leaves in the tree is $\geq n!$.
- A binary tree with height $h$ has $\leq 2^h$ leaves.
- Therefore, $2^h \geq n! \Rightarrow h \geq \log(n!)$.
- By Stirling’s approximation, $n! \geq (n/e)^n$. Therefore,

$$h \geq \log((n/e)^n)$$

$$= n \log(n/e) = n \log n - n \log e = \Omega(n \log n)$$
Lemma

A binary tree with height $h$ has $\leq 2^h$ leaves.

Proof.

- The proof is by induction on $h$.
- The base $h = 0$ is true.
- If $T$ is a binary tree of height $h > 0$, then removing the root of $T$ gives two trees $T_1$, $T_2$ of height at most $h - 1$ each.
- By induction, each of $T_1$, $T_2$ has $\leq 2^{h-1}$ leaves.
- Therefore, $T$ has $\leq 2 \cdot 2^{h-1} = 2^h$ leaves.
We will see several sorting algorithms that run in $\Theta(n)$ time:

- Counting Sort.
- Radix Sort.
- Bucket Sort.

These algorithms are not comparison based.
Counting Sort

$A =$ Input array. Values in $A$ are from \{0, 1, \ldots, k\}.

$B =$ Output array.

**COUNTINGSORT**(A, B, k)

1. Let $C[0..k]$ be a new array
2. for $i = 0$ to $k$
   3. $C[i] \leftarrow 0$
4. for $j = 1$ to $A$.length
   5. $C[A[j]] \leftarrow C[A[j]] + 1$

% $C[i]$ now contains the number of elements $= i$

6. for $i = 1$ to $k$
   7. $C[i] \leftarrow C[i] + C[i - 1]$

% $C[i]$ now contains the number of elements $\leq i$

8. for $j = A$.length downto 1
Counting Sort

A = Input array. Values in A are from \{0, 1, \ldots, k\}.
B = Output array.

**COUNTINGSORT**(A, B, k)

1. Let C[0..k] be a new array
2. for i = 0 to k
3. \( C[i] \leftarrow 0 \)
4. for j = 1 to A.length
5. \( C[A[j]] \leftarrow C[A[j]] + 1 \)
   % C[i] now contains the number of elements = i
6. for i = 1 to k
7. \( C[i] \leftarrow C[i] + C[i - 1] \)
   % C[i] now contains the number of elements \( \leq i \)
8. for j = A.length downto 1
10. \( C[A[j]] \leftarrow C[A[j]] - 1 \)
Counting Sort

\( A = \) Input array. Values in \( A \) are from \( \{0, 1, \ldots, k\} \).

\( B = \) Output array.

**CountingSort**\((A, B, k)\)

1. Let \( C[0..k] \) be a new array
2. for \( i = 0 \) to \( k \)
3. \( C[i] \leftarrow 0 \)
4. for \( j = 1 \) to \( A.length \)
5. \( C[A[j]] \leftarrow C[A[j]] + 1 \)
   % \( C[i] \) now contains the number of elements = \( i \)
6. for \( i = 1 \) to \( k \)
7. \( C[i] \leftarrow C[i] + C[i - 1] \)
   % \( C[i] \) now contains the number of elements \( \leq i \)
8. for \( j = A.length \) downto \( 1 \)
10. \( C[A[j]] \leftarrow C[A[j]] - 1 \)

\( A, B, k, C \)
Counting Sort

Let $A$ be an input array. Values in $A$ are from $\{0, 1, \ldots, k\}$.

Let $B$ be an output array.

**COUNTINGSORT**(A, B, k)

1. Let $C[0..k]$ be a new array
2. for $i = 0$ to $k$
   3. $C[i] \leftarrow 0$
4. for $j = 1$ to $A.length$
5.   $C[A[j]] \leftarrow C[A[j]] + 1$
   
   % $C[i]$ now contains the number of elements $= i$
6. for $i = 1$ to $k$
7.   $C[i] \leftarrow C[i] + C[i - 1]$
   
   % $C[i]$ now contains the number of elements $\leq i$
8. for $j = A.length$ downto 1
Counting Sort

\[ A = \text{Input array. Values in } A \text{ are from } \{0, 1, \ldots, k\}. \]
\[ B = \text{Output array.} \]

**COUNTINGSORT** \((A, B, k)\)

(1) Let \(C[0..k]\) be a new array
(2) for \(i = 0\) to \(k\)
(3) \(C[i] \leftarrow 0\)
(4) for \(j = 1\) to \(A\.length\)
(5) \(C[A[j]] \leftarrow C[A[j]] + 1\)
% \(C[i]\) now contains the number of elements = \(i\)
(6) for \(i = 1\) to \(k\)
(7) \(C[i] \leftarrow C[i] + C[i - 1]\)
% \(C[i]\) now contains the number of elements \(\leq i\)
(8) for \(j = A\.length\) **down to** 1
(9) \(B[C[A[j]]] \leftarrow A[j]\)
(10) \(C[A[j]] \leftarrow C[A[j]] - 1\)
Counting Sort

$A =$ Input array. Values in $A$ are from $\{0, 1, \ldots, k\}$.
$B =$ Output array.

\textbf{C}ounting\textbf{S}ort($A$, $B$, $k$)

(1) Let $C[0..k]$ be a new array
(2) \textbf{for} $i = 0$ \textbf{to} $k$
(3) \hspace{1em} $C[i] \leftarrow 0$
(4) \textbf{for} $j = 1$ \textbf{to} $A$.length
(5) \hspace{1em} $C[A[j]] \leftarrow C[A[j]] + 1$
\hspace{1em} \% $C[i]$ now contains the number of elements $=$ $i$
(6) \textbf{for} $i = 1$ \textbf{to} $k$
(7) \hspace{1em} $C[i] \leftarrow C[i] + C[i - 1]$
\hspace{1em} \% $C[i]$ now contains the number of elements $\leq i$
(8) \textbf{for} $j = A$.length \textbf{downto} 1
(9) \hspace{1em} $B[C[A[j]]] \leftarrow A[j]$
(10) \hspace{1em} $C[A[j]] \leftarrow C[A[j]] - 1$
Counting Sort

A = Input array. Values in A are from \{0, 1, \ldots, k\}.
B = Output array.

**COUNTINGSORT(A, B, k)**

1. Let C[0..k] be a new array
2. for \( i = 0 \) to \( k \)
3. \( C[i] \leftarrow 0 \)
4. for \( j = 1 \) to A.length
5. \( C[A[j]] \leftarrow C[A[j]] + 1 \)
   % \( C[i] \) now contains the number of elements = \( i \)
6. for \( i = 1 \) to \( k \)
7. \( C[i] \leftarrow C[i] + C[i - 1] \)
   % \( C[i] \) now contains the number of elements \( \leq i \)
8. for \( j = A.length \) downto 1
10. \( C[A[j]] \leftarrow C[A[j]] - 1 \)
Counting Sort

*A* = Input array. Values in *A* are from \{0, 1, \ldots, k\}.

*B* = Output array.

**CountingSort**(*A*, *B*, *k*)

1. Let *C*[0..*k*] be a new array
2. for *i* = 0 to *k*
3.   *C*[*i*] ← 0
4. for *j* = 1 to *A*.length
5.   *C*[*A*[*j*]] ← *C*[*A*[*j*]] + 1
   % *C*[*i*] now contains the number of elements = *i*
6. for *i* = 1 to *k*
7.   *C*[*i*] ← *C*[*i*] + *C*[*i* − 1]
   % *C*[*i*] now contains the number of elements ≤ *i*
8. for *j* = *A*.length downto 1
9.   *B*[*C*[*A*[*j*]]] ← *A*[*j*]
10. *C*[*A*[*j*]] ← *C*[*A*[*j*]] − 1
Counting Sort

\( A = \) Input array. Values in \( A \) are from \( \{0, 1, \ldots, k\} \).
\( B = \) Output array.

**COUNTINGSORT**(\( A, B, k \))

1. Let \( C[0..k] \) be a new array
2. for \( i = 0 \) to \( k \)
3. \( C[i] \leftarrow 0 \)
4. for \( j = 1 \) to \( A.\text{length} \)
5. \( C[A[j]] \leftarrow C[A[j]] + 1 \)
6. for \( i = 1 \) to \( k \)
7. \( C[i] \leftarrow C[i] + C[i - 1] \)
8. for \( j = A.\text{length} \) downto \( 1 \)
10. \( C[A[j]] \leftarrow C[A[j]] - 1 \)
Counting Sort

$A =$ Input array. Values in $A$ are from $\{0, 1, \ldots, k\}$.

$B =$ Output array.

**CountingSort**($A$, $B$, $k$)

1. Let $C[0..k]$ be a new array
2. for $i = 0$ to $k$
3. \hspace{1em} $C[i] \leftarrow 0$
4. for $j = 1$ to $A$.length
5. \hspace{1em} $C[A[j]] \leftarrow C[A[j]] + 1$
\hspace{2em} % $C[i]$ now contains the number of elements $= i$
6. for $i = 1$ to $k$
7. \hspace{1em} $C[i] \leftarrow C[i] + C[i - 1]$
\hspace{2em} % $C[i]$ now contains the number of elements $\leq i$
8. for $j = A$.length downto 1
9. \hspace{1em} $B[C[A[j]]] \leftarrow A[j]$
10. \hspace{1em} $C[A[j]] \leftarrow C[A[j]] - 1$
Counting Sort

\[ A = \text{Input array. Values in } A \text{ are from } \{0, 1, \ldots, k\}. \]
\[ B = \text{Output array.} \]

**COUNTINGSORT** \((A, B, k)\)

1. Let \(C[0..k]\) be a new array.
2. **for** \(i = 0\) **to** \(k\)
3. \(C[i] \leftarrow 0\)
4. **for** \(j = 1\) **to** \(A\).length
5. \(C[A[j]] \leftarrow C[A[j]] + 1\)
\% \(C[i]\) now contains the number of elements = \(i\)
6. **for** \(i = 1\) **to** \(k\)
7. \(C[i] \leftarrow C[i] + C[i - 1]\)
\% \(C[i]\) now contains the number of elements \(\leq i\)
8. **for** \(j = A\).length **down to** 1
9. \(B[C[A[j]]] \leftarrow A[j]\)
10. \(C[A[j]] \leftarrow C[A[j]] - 1\)
Counting Sort

\( A = \) Input array. Values in \( A \) are from \( \{0, 1, \ldots, k\} \).

\( B = \) Output array.

**CountingSort**\((A, B, k)\)

(1) Let \( C[0..k] \) be a new array

(2) \textbf{for} \( i = 0 \) \textbf{to} \( k \)

(3) \( C[i] \leftarrow 0 \)

(4) \textbf{for} \( j = 1 \) \textbf{to} \( A.length \)

(5) \( C[A[j]] \leftarrow C[A[j]] + 1 \)

\% \( C[i] \) now contains the number of elements = \( i \)

(6) \textbf{for} \( i = 1 \) \textbf{to} \( k \)

(7) \( C[i] \leftarrow C[i] + C[i - 1] \)

\% \( C[i] \) now contains the number of elements \( \leq i \)

(8) \textbf{for} \( j = A.length \textbf{ downto} 1 \)

(9) \( B[C[A[j]]] \leftarrow A[j] \)

(10) \( C[A[j]] \leftarrow C[A[j]] - 1 \)
Counting Sort

$A =$ Input array. Values in $A$ are from $\{0, 1, \ldots, k\}$.
$B =$ Output array.

**COUNTINGSORT**($A$, $B$, $k$)

(1) Let $C[0..k]$ be a new array
(2) for $i =$ 0 to $k$
(3) $C[i] \leftarrow 0$
(4) for $j =$ 1 to $A$.length
(5) $C[A[j]] \leftarrow C[A[j]] + 1$
   % $C[i]$ now contains the number of elements $= i$
(6) for $i =$ 1 to $k$
(7) $C[i] \leftarrow C[i] + C[i - 1]$
   % $C[i]$ now contains the number of elements $\leq i$
(8) for $j =$ $A$.length **down to** 1
(9) $B[C[A[j]]] \leftarrow A[j]$
(10) $C[A[j]] \leftarrow C[A[j]] - 1$

*Example:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 3 0 2 3 0 3</td>
<td>0 1 2 3 4 5</td>
<td>1 2 4 5 7 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Counting Sort

\( A = \) Input array. Values in \( A \) are from \( \{0, 1, \ldots, k\} \).
\( B = \) Output array.

\textbf{COUNTING\text{\textsc{sort}}(A, B, k)}

1. Let \( C[0..k] \) be a new array
2. \textbf{for} \( i = 0 \) \textbf{to} \( k \)
3. \( C[i] \leftarrow 0 \)
4. \textbf{for} \( j = 1 \) \textbf{to} \( A.\text{length} \)
5. \( C[A[j]] \leftarrow C[A[j]] + 1 \)
   \( \% \) \( C[i] \) now contains the number of elements = \( i \)
6. \textbf{for} \( i = 1 \) \textbf{to} \( k \)
7. \( C[i] \leftarrow C[i] + C[i - 1] \)
   \( \% \) \( C[i] \) now contains the number of elements \( \leq i \)
8. \textbf{for} \( j = A.\text{length} \) \textbf{downto} \( 1 \)
10. \( C[A[j]] \leftarrow C[A[j]] - 1 \)
Counting Sort

Time complexity: $\Theta(n + k)$.
If $k = O(n)$ then the time is $\Theta(n)$.

$\text{COUNTINGSORT}(A, B, k)$

1. Let $C[0..k]$ be a new array
2. for $i = 0$ to $k$
   3. $C[i] \leftarrow 0$
4. for $j = 1$ to $A$.length
   5. $C[A[j]] \leftarrow C[A[j]] + 1$
      % $C[i]$ now contains the number of elements $= i$
6. for $i = 1$ to $k$
   7. $C[i] \leftarrow C[i] + C[i - 1]$
      % $C[i]$ now contains the number of elements $\leq i$
8. for $j = A$.length downto 1
Counting Sort is stable (elements with same key maintain their relative order).

```
A  2 5 3 0 2 3 0 3
B  0 0 2 2 3 3 3 5
```
Radix Sort

$A =$ Input array. Values in $A$ are numbers with $d$ digits.

**RadixSort**($A, d$)

(1) **for** $i = 1$ **to** $d$

(2) Use a stable sort to sort array $A$ on digit $i$

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

In the following, assume that the stable sorting algorithm used by radix sort is Counting Sort.

Claim
Given $n$ numbers, where each number has $d$ digits in base $k$, the time complexity of Radix Sort is $\Theta(d(n + k))$.

Proof.
- The algorithm has $d$ iterations.
- The time of a single iteration is $\Theta(n + k)$. 
Claim
Given $n$ $b$-bits numbers, and $r \leq b$, the time complexity of Radix Sort is $\Theta((b/r)(n + 2^r))$.

Proof.
- Each number can be viewed as a number in base $k = 2^r$ with $d = \lceil b/r \rceil$ digits.
- The time complexity is $\Theta(d(n + k)) = \Theta((b/r)(n + 2^r))$. 
Choosing \( r \) — example

Suppose that \( n = 50000 \) and \( b = 32 \).

<table>
<thead>
<tr>
<th>( r )</th>
<th>( d = b/r )</th>
<th>( k = 2^r )</th>
<th>( (b/r)(n + 2^r) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>2</td>
<td>32 \cdot (50000 + 2) = 1600064</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>4</td>
<td>16 \cdot (50000 + 4) = 800064</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>8</td>
<td>11 \cdot (50000 + 8) = 550088</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>512</td>
<td>4 \cdot (50000 + 512) = 202048</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>1024</td>
<td>4 \cdot (50000 + 1024) = 204096</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>2048</td>
<td>3 \cdot (50000 + 2048) = 156144</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>4096</td>
<td>3 \cdot (50000 + 4096) = 162288</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>8192</td>
<td>3 \cdot (50000 + 8192) = 174576</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>16384</td>
<td>3 \cdot (50000 + 16384) = 199152</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>32768</td>
<td>3 \cdot (50000 + 32768) = 248304</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>65536</td>
<td>2 \cdot (50000 + 65536) = 231072</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
</tbody>
</table>

Sorting algorithms
Choosing $r$

**Claim**

Given $n$ $b$-bits numbers, the time complexity of Radix Sort is
- $\Theta(n)$ if $b \leq \log n$
- $\Theta(bn/\log n)$ if $b > \log n$.

**Proof.**

If $b \leq \log n$:
- For every $r \leq b$, $(b/r)(n + 2^r) \leq (b/r)(n + n) = 2b/r \cdot n$.
- For $b = r$, the time is $\Theta(n)$ which is optimal.

If $b > \log n$:
- For $r = \lfloor \log n \rfloor$, $(b/r)(n + 2^r) = \Theta((b/\log n)n)$.
- If $r < \lfloor \log n \rfloor$, the $b/r$ term increases while the $n + 2^r$ term remains $\Theta(n)$.
- If $r > \lfloor \log n \rfloor$, the $2^r$ term increases faster than the $r$ term in the denominator.
Choosing $r$ — example

- Suppose that $n = 50000$ and $b = 32$.
- $\log_2 n = 15.609$, so an asymptotically optimal choice is $r = 15$. 
The Bucket Sort algorithm sorts an array whose elements are numbers from the interval \([0, 1)\).

The main idea is to partition the interval \([0, 1)\) into \(n\) intervals of equal lengths. The elements of the array are partitioned into buckets according to the intervals. Then, each bucket is sorted using some sorting algorithm (for example, insertion sort).
Bucket Sort

A = Input array. 0 ≤ A[i] < 1 for all i.

**BucketSort(A)**

1. \( n \leftarrow A\.length \)
2. Let \( B[0..n - 1] \) be a new array
3. for \( i = 0 \) to \( n - 1 \)
4. \( B[i] \leftarrow \text{NULL} \)
5. for \( i = 1 \) to \( n \)
6. Insert \( A[i] \) to list \( B[\lfloor nA[i] \rfloor] \)
7. for \( i = 0 \) to \( n - 1 \)
8. Sort \( B[i] \) (with Insertion Sort)
9. Concatenate the lists \( B[0], B[1], \ldots, B[n - 1] \) in order
Bucket Sort

A = Input array. 0 ≤ A[i] < 1 for all i.

**BucketSort(A)**

1. \( n \leftarrow A\).length
2. Let \( B[0..n − 1] \) be a new array
3. for \( i = 0 \) to \( n − 1 \)
4. \( B[i] \leftarrow \text{NULL} \)
5. for \( i = 1 \) to \( n \)
6. Insert \( A[i] \) to list \( B[\lfloor nA[i]\rfloor] \)
7. for \( i = 0 \) to \( n − 1 \)
8. Sort \( B[i] \) (with Insertion Sort)
9. Concatenate the lists \( B[0], B[1], \ldots, B[n − 1] \) in order
Bucket Sort

A = Input array. \(0 \leq A[i] < 1\) for all \(i\).

**BucketSort(A)**

1. \(n \leftarrow \text{A.length}\)
2. Let \(B[0..n - 1]\) be a new array
3. for \(i = 0\) to \(n - 1\)
4. \(B[i] \leftarrow \text{NULL}\)
5. for \(i = 1\) to \(n\)
6. Insert \(A[i]\) to list \(B[\lfloor nA[i]\rfloor]\)
7. for \(i = 0\) to \(n - 1\)
8. Sort \(B[i]\) (with Insertion Sort)
9. Concatenate the lists \(B[0], B[1], \ldots, B[n - 1]\) in order
Bucket Sort

\[ A = \text{Input array. } 0 \leq A[i] < 1 \text{ for all } i. \]

**BucketSort** \( A \)

1. \( n \leftarrow A.\text{length} \)
2. Let \( B[0..n - 1] \) be a new array
3. for \( i = 0 \) to \( n - 1 \)
4. \( B[i] \leftarrow \text{NULL} \)
5. for \( i = 1 \) to \( n \)
6. Insert \( A[i] \) to list \( B[\lfloor nA[i] \rfloor] \)
7. for \( i = 0 \) to \( n - 1 \)
8. Sort \( B[i] \) (with Insertion Sort)
9. Concatenate the lists \( B[0], B[1], \ldots, B[n - 1] \) in order
Bucket Sort

$A =$ Input array. $0 \leq A[i] < 1$ for all $i$.  

**BucketSort**(A)

1. $n \leftarrow A$.length
2. Let $B[0..n - 1]$ be a new array
3. for $i = 0$ to $n - 1$
4.   $B[i] \leftarrow$ NULL
5. for $i = 1$ to $n$
6.   Insert $A[i]$ to list $B[\lfloor nA[i] \rfloor]$
7. for $i = 0$ to $n - 1$
8.   Sort $B[i]$ (with Insertion Sort)
9. Concatenate the lists $B[0], B[1], \ldots, B[n - 1]$ in order
**Bucket Sort**

\( A = \) Input array. \( 0 \leq A[i] < 1 \) for all \( i \).

**BucketSort(A)**

1. \( n \leftarrow A.\text{length} \)
2. Let \( B[0..n-1] \) be a new array
3. for \( i = 0 \) to \( n-1 \)
4. \( B[i] \leftarrow \text{NULL} \)
5. for \( i = 1 \) to \( n \)
6. Insert \( A[i] \) to list \( B[\lfloor nA[i] \rfloor] \)
7. for \( i = 0 \) to \( n-1 \)
8. Sort \( B[i] \) (with Insertion Sort)
9. Concatenate the lists \( B[0], B[1], \ldots, B[n-1] \) in order

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.72</td>
<td>0.12</td>
</tr>
<tr>
<td>0.17</td>
<td>0.17</td>
</tr>
<tr>
<td>0.78</td>
<td>0.21</td>
</tr>
<tr>
<td>0.26</td>
<td>0.23</td>
</tr>
<tr>
<td>0.39</td>
<td>0.68</td>
</tr>
<tr>
<td>0.94</td>
<td>0.72</td>
</tr>
<tr>
<td>0</td>
<td>0.78</td>
</tr>
<tr>
<td>1</td>
<td>0.17</td>
</tr>
<tr>
<td>2</td>
<td>0.21</td>
</tr>
<tr>
<td>3</td>
<td>0.39</td>
</tr>
<tr>
<td>4</td>
<td>0.68</td>
</tr>
<tr>
<td>5</td>
<td>0.72</td>
</tr>
<tr>
<td>6</td>
<td>0.78</td>
</tr>
<tr>
<td>7</td>
<td>0.94</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Assume that the elements of $A$ are chosen randomly from $[0, 1)$ (uniformly and independently).

Let $n_i$ be the number of elements in $B[i]$.

The expected time complexity is

$$
E \left[ \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right] = \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \\
= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \\
= \Theta(n) + O(n \cdot E[n_1^2])
$$
Let $X_i = 1$ if $A[i]$ falls to bucket 1.

$n_1 = \sum_{i=1}^{n} X_i$

$$E[n_1^2] = E\left[\left(\sum_{i=1}^{n} X_i\right)^2\right]$$

$$= E\left[\sum_{i=1}^{n} X_i^2 + \sum_{i=1}^{n} \sum_{j=1}^{n} X_i X_j\right]$$

$$= \sum_{i=1}^{n} E[X_i^2] + \sum_{j=1}^{n} E[X_i X_j]$$

$$= nE[X_1^2] + n(n - 1)E[X_1 X_2]$$
Time complexity

\[ E[X_1^2] = 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n} = \frac{1}{n} \]

\[ E[X_1X_2] = 0 \cdot \left(1 - \frac{1}{n^2}\right) + 1 \cdot \frac{1}{n^2} = \frac{1}{n^2} \]

Therefore,

\[ E[n_1^2] = n \cdot \frac{1}{n} + n(n - 1) \cdot \frac{1}{n^2} = 1 + \frac{n - 1}{n} = 2 - \frac{1}{n}. \]

The expected time complexity of Bucket Sort is \( \Theta(n) \).