Preview

- Sorting and Order
  - Basic Sorting Algorithms
    - Selection Sort
    - Insertion Sort
    - Bubble Sort
    - Shell Sort
  - Divide and Conquer
    - Merge Sort
    - Quick Sort
    - Heap Sort
  - Sorting in Linear Time
    - Counting Sort
    - Radix Sort
    - Bucket Sort

Preview

- Sorting in Linear Time
  - Counting Sort
  - Radix Sort
  - Bucket Sort
Sorting in Linear Time

Sorting algorithms such as Insertion, Selection, Bubble, Merge, Quick and Heap Sort share a common property -- the sorted order they determine is based only on comparisons between the input elements.

- It can be shown that such comparison sorts must have a lower bound of $\Omega(n \log_2 n)$ comparison operations in the worst case.

A few linear time sorting algorithms exist. However, these algorithms are available only for special forms of input.

- The following algorithms use operations other than comparisons to determine the sorted order, thus the $\Omega(n \log_2 n)$ lower bound does not apply to them.
  - Counting Sort
  - Radix Sort
  - Bucket Sort
Sorting in Linear Time
(Counting Sort)

- Counting sort assumes that each of the $n$ input elements is an integer in the range $0$ to $k$, for some integer $k$.
- With $k = O(n)$, the running time of counting sort is $\Theta(n)$.
- For each input element $x$, counting sort counts the number of elements less than $x$.
- Counting sort uses this counting information to place element $x$ directly into its position in the output array.
- For example, if there are 17 elements less than element $x$, the index of element $x$ must be the 18th location in the array.

Input list

1. $n$ input elements, each is an integer in the range $0$ to $k$ for some input integer $k$.
2. If the size of $k = O(n)$, the running time for counting sort takes $O(n)$.
3. We need two other arrays:
   - $B[1..n]$ – holds the output
   - $C[0..k]$ – holds the counting information
Sorting in Linear Time
(Counting Sort)

\begin{algorithm}
\textbf{CountingSort} (A, B, k)
\{
\begin{align*}
1 \text{ for } i = 0 \text{ to } k & \quad \text{// array } C \text{ is used to count the} \\
2 \quad C[i] = 0; & \quad \text{// occurrences of the } i\text{th in the list } A \\
3 \text{ for } j = 1 \text{ to } |A| & \quad \text{// count the number of } i \\
4 \quad C[A[j]] = C[A[j]] + 1; & \\
5 \text{ for } i = 1 \text{ to } k & \\
6 \quad C[i] = C[i] + C[i-1]; & \\
7 \text{ for } j = |A| \text{ down to } 1 & \quad \{ \\
8 \quad B[C[A[j]]] = A[j] & \quad \ldots \quad \text{0}(k) \\
9 \quad C[A[j]] = C[A[j]] - 1; & \quad \ldots \quad \text{0}(n) \\
\}
\}
\end{align*}
\end{algorithm}

Running time of counting sort is $O(k + n)$
Sorting in Linear Time
(Counting Sort)

CountingSort(A, B, 6)

<table>
<thead>
<tr>
<th>A</th>
<th>2 0 3 2 5 4 3 6 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1 2 3 4 5 6 7 8 9 10</td>
</tr>
<tr>
<td>C</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

CountingSort(A, B, k)

```
for i = 0 to k
    C[i] = 0;
for j = 1 to |A| do
    C[A[j]] = C[A[j]] + 1;
for i = 1 to k
    C[i] = C[i] + C[i-1];
for j = |A| down to 1 do
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]] - 1;
```

CountingSort(A, B, 6)
Sorting in Linear Time
(Counting Sort)

CountingSort(A, B, k)

\[
\begin{align*}
\text{CountingSort}(A, B, 6) & \quad \text{for } i = 0 \text{ to } k & \quad C[i] &= 0; \\
\text{for } j = 1 \text{ to } |A| & \quad C[A[j]] &= C[A[j]] + 1; \\
\text{for } i = 1 \text{ to } k & \quad C[i] &= C[i] + C[i-1]; \\
\text{for } j = |A| \text{ down to } 1 & \quad B[C[A[j]]] &= A[j]; \\
& \quad C[A[j]] &= C[A[j]] - 1;
\end{align*}
\]
Sorting in Linear Time
(Counting Sort)

CountingSort(A, B, 6)

```
CountingSort(A, B, k)
{
  1 for i = 0 to k
  2 C[i] = 0;
  3 for j = 1 to |A| do
  4    C[A[j]] = C[A[j]] + 1;
  5 for i = 1 to k
  6    C[i] = C[i] + C[i-1];
  7 for j = |A| down to 1
  8    B[C[A[j]]] = A[j];
  9    C[A[j]] = C[A[j]] - 1;
}
```

B = 0 1 2 3 4 5 6 7 8 9 10
C = 1 2 3 4 5 6
A = 2 0 3 2 5 4 3 6 1 0
Sorting in Linear Time
(Counting Sort)

CountingSort(A, B, 6)

CountingSort\( (A, B, k) \)
\[
\begin{array}{l}
\text{1 for } i = 0 \text{ to } k \\
\text{2 } C[i] = 0 \\
\text{3 for } j = 1 \text{ to } |A| \text{ do} \\
\text{4 } C[A[j]] = C[A[j]] + 1; \\
\text{5 for } i = 1 \text{ to } k \\
\text{6 } C[i] = C[i] + C[i-1]; \\
\text{7 for } j = |A| \text{ down to } 1 \\
\text{8 } B[C[A[j]]] = A[j]; \\
\text{9 } C[A[j]] = C[A[j]] - 1; \\
\end{array}
\]

COSC 320 - Fall 2015
Slides courtesy of Dr. Sang-Eon Park
Sorting in Linear Time

(Counting Sort)

CountingSort(A, B, 6)

CountingSort\(A, B, k\)
\{
  1. for \(i = 0\) to \(k\)
  2. \(C[0] = 0;\)
  3. for \(j = 1\) to |\(A|\) do
    4. \(C[A[j]] = C[A[j]] + 1;\)
    5. if \(i = 1\) to \(k\)
      6. \(C[i] = C[i] + C[i-1];\)
    7. for \(j = |A|\) down to 1
      \{
        8. \(B[C[A[j]]] = A[j];\)
        9. \(C[A[j]] = C[A[j]] - 1;\)
      \}
\}

COSC 320 - Fall 2015
Slides courtesy of Dr. Sang-Eon Park
Sorting in Linear Time
(Counting Sort)

CountingSort(A, B, 6)

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
A & 2 & 0 & 3 & 2 & 5 & 4 & 3 & 6 & 1 & 0 \\
B & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
C & 1 & 2 & 4 & 6 & 7 & 8 & 9 \\
\end{array}
\]

CountingSort(A, B, k)

\[
\begin{array}{l}
\text{for } i = 0 \text{ to } k \\
\text{C[i] = 0;}
\end{array}
\]

\[
\begin{array}{l}
\text{for } j = 1 \text{ to } |A| \text{ do} \\
\text{C[A[j]] = C[A[j]] + 1;}
\end{array}
\]

\[
\begin{array}{l}
\text{for } i = 1 \text{ to } k \\
\text{C[i] = C[i] + C[i-1];}
\end{array}
\]

\[
\begin{array}{l}
\text{for } j = |A| \text{ down to 1} \\
\text{B[C[A[j]]] = A[j];}
\end{array}
\]

\[
\begin{array}{l}
\text{C[A[j]] = C[A[j]] - 1;}
\end{array}
\]

CountingSort(A, B, 6)

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
A & 2 & 0 & 3 & 2 & 5 & 4 & 3 & 6 & 1 & 0 \\
B & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
C & 1 & 2 & 4 & 6 & 7 & 8 & 9 \\
\end{array}
\]

COSC 320 - Fall 2015
Slides courtesy of Dr. Sang-Eon Park
### Sorting in Linear Time

#### (Counting Sort)

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

```c
CountingSort(A, B, k) {
    1 for i = 0 to k
    2 C[i] = 0
    3 for j = 1 to |A| do
    4     C[A[j]] = C[A[j]] + 1;
    5 for i = 1 to k
    6     C[i] = C[i] + C[i-1];
    7 for j = |A| down to 1
    8     B[C[A[j]]] = A[j];
    9     C[A[j]] = C[A[j]] - 1;
}
```

**CountingSortII(A, k)**

```c
CountingSortII(A, k) {
    1 for i = 0 to k
    2     C[i] = 0;
    3 for j = 1 to |A| do
    4     C[A[j]] = C[A[j]] + 1;
    5 i = 1;
    6 for j = 0 to k
    7     m = C[j];
    8     while m > 0 do
    9         A[i] = j;
    10        i++;
    11        m = m - 1;
    12     }
}
```

COSC 320 - Fall 2015
Slides courtesy of Dr. Sang-Eon Park
CountingSortII(A, k)
{
  for i = 0 to k
  C[i] = 0;
  for j = 1 to |A| do
  C[A[j]] = C[A[j]] + 1;
  i = 1;
  for j = 0 to k
  {
    m = C[j];
    while m > 0 do
    {
      A[i] = j;
      i++;
      m = m - 1;
    }
  }
}
Sorting in Linear Time
(Counting Sort: Version II)

CountingSortII(A, B, 6)

<table>
<thead>
<tr>
<th>i=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9 10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 0 3 2 5 4 3 6 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 2 2 1 1 1</td>
</tr>
</tbody>
</table>

CountingSortII(A, k)

```java
for i = 0 to k
C[i] = 0;
for j = 1 to |A| do
C[A[j]] = C[A[j]] + 1;
i = 1;
for j = 0 to k
m = C[j];
while m > 0 do
A[i] = j;
i++;
m = m - 1;
```

J=0

<table>
<thead>
<tr>
<th>i=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9 10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 3 2 5 4 3 6 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 2 1 1 1</td>
</tr>
</tbody>
</table>

CountingSortII(A, k)

```java
for i = 0 to k
C[i] = 0;
for j = 1 to |A| do
C[A[j]] = C[A[j]] + 1;
i = 1;
for j = 0 to k
m = C[j];
while m > 0 do
A[i] = j;
i++;
m = m - 1;
```
Sorting in Linear Time
(Counting Sort: Version II)

CountingSortII(A, B, 6)

1 = 4

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
A & 0 & 0 & 1 & 2 & 5 & 4 & 3 & 6 & 1 & 0 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
C & 0 & 0 & 2 & 2 & 1 & 1 & 1 \\
\end{array}
\]

J = 2

CountingSortII(A, k)

\{
1 for i = 0 to k
2 \quad C[i] = 0;
3 for j = 1 to |A| do
4 \quad C[A[j]] = C[A[j]] + 1;
5 \quad i = 1;
6 for j = 0 to k
7 \quad m = C[j];
8 \quad while m > 0 do
9 \quad \quad A[i] = j;
10 \quad \quad i++;
11 \quad \quad m = m - 1;
\}

J = 3

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
A & 0 & 0 & 1 & 2 & 2 & 4 & 3 & 6 & 1 & 0 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
C & 0 & 0 & 0 & 2 & 1 & 1 & 1 \\
\end{array}
\]
Counting Sort: Version II

CountingSortII(A, B, 6)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, B, 6)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]

CountingSortII(A, k)

\[
\begin{array}{cccccccc}
A & 0 & 0 & 1 & 2 & 2 & 3 & 3 & 6 & 1 & 0 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\
\end{array}
\]
Sorting in Linear Time
(Counting Sort: Version II)

CountingSortII(A, B, 6)

<table>
<thead>
<tr>
<th>i=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9 10</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>0 0 1 2 2 3 3 4 5 0</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 1</td>
</tr>
</tbody>
</table>

CountingSortII(A, k)

```plaintext
for i = 0 to k
    C[i] = 0;
for j = 1 to |A| do
    C[A[j]] = C[A[j]] + 1;
for j = 0 to k
    m = C[j];
    while m > 0 do
        A[i] = j;
        i++;
        m = m - 1;
```

COSC 320 - Fall 2015
Slides courtesy of Dr. Sang-Eon Park
Sorting in Linear Time
(Radix Sort)

- Radix sort is the algorithm used by the card-sorting machines, which you can find only in the computer museum.
- The cards are organized into 80 columns, and in each column a hole can be punched in one of 12 places.
- The sorter can be mechanically programmed to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched.

IBM 80-column punched card

- This IBM card format had rectangular holes, 80 columns with 12 punch locations each, one character to each column.
- The lower ten positions represented (from top to bottom) the digits 0 through 9.
- The top two positions of a column were called zone punches, 12 (top) and 11.
- Only numeric information was punched, with 1 punch per column indicating the digit.
- The zone punches was used for characters or symbols.
Sorting in Linear Time
(Radix Sort)

A \(d\)-digit number occupies a field of \(d\) columns.

Since a card sorter can look at only one column at a time, the problem of sorting \(n\) cards on \(d\)-digit number used radix sort.

The Radix Sort sorts numbers digit by digit working from the least significant digit to the most significant digit.

The digit sorts must be stable in order for radix sort to work correctly.
What is a “stable” sort?

Stable sorting algorithms maintain the relative order of items with equal keys (i.e., values).

In other words, a sorting algorithm is stable if whenever there are two items A and B with the same key and A appears before S in the original list, then A will appear before B in the sorted list.

We can use a radix sort to sort decimal numbers since there are only 10 different numbers that are used as digits.

Radix_Sort (A, d)
{
1  for i = 1 to d
2      Use a stable sort to sort array A on digit i
}
Sorting in Linear Time
(Radix Sort)

- The analysis of running time depends on the stable sort used as the intermediate sorting algorithm.
- If each digit is in the range 1 to $k$, and $k$ is not too large, counting sort ($O(n + k)$) is the obvious choice.
- If the numbers are all $d$ digit numbers, the running time of Radix Sort is $d \times O(n + k) = O(dn + dk)$
- If $d$ is constant and $k = O(n)$, then the running time of Radix sort is $O(n)$. 
Bucket sort assumes that the input is drawn from a uniform distribution. This results in an average-case running time of $O(n)$.

Bucket sort is fast because it assumes something about the input.

- Counting sort – assumes that the input consists of integers in a small range.
- Bucket sort – assumes that the input is generated by a random process that distributes elements uniformly over the interval $0 \leq k < 1$

The idea behind bucket sort is to divide the interval $[0, 1)$ into $n$ equal-sized subintervals called buckets, and then distribute the $n$ input numbers into the buckets.

Since the inputs are uniformly and independently distributed, there are not many elements in each bucket.

Each bucket is sorted using insertion sort.
Sorting in Linear Time  
(Bucket Sort)

**Bucket-Sort(A) {**

1. \( n = |A| \)
2. Let \( B[0..n-1] \) be a new array
3. for \( i = 0 \) to \( n-1 \)
4. make \( B[i] \) an empty list
5. for \( i = 0 \) to \( n-1 \)
6. insert \( A[i] \) into the list \( B[\lfloor nA[i] \rfloor] \).
7. for \( i = 0 \) to \( n-1 \)
8. sort bucket \( B[i] \) using insertion sort.
9. concatenate the lists from the buckets \( B[0], B[1], ..., B[|A|-1] \) together in order.

**}"
Sorting in Linear Time
(Bucket Sort)

Bucket-Sort(A) {
    1 n = |A|
    2 Let B[0..n-1] be a new array
    3 for i = 0 to n-1
       4 make B[i] an empty list
    5 for i = 0 to n-1
       6 insert A[i] into the list B[⌊nA[i]⌋].
    7 for i = 0 to n-1
       8 sort bucket B[i] using insertion sort.
    9 concatenate the lists from the buckets B[0], B[1], ..., B[|A|-1] together in order.
}

COSC 320 - Fall 2015
Slides courtesy of Dr. Sang-Eon Park
Sorting in Linear Time
(Bucket Sort)

Bucket-Sort(A) {
    n = |A|
    Let B[0..n-1] be a new array
    for i = 0 to n-1
        make B[i] an empty list
    for i = 0 to n-1
        insert A[i] into the list B[⌊nA[i]⌋].
    for i = 0 to n-1
        sort bucket B[i] using insertion sort.
    concatenate the lists from the buckets B[0], B[1], .., B[|A|-1] together in order.
}

COSC 320 - Fall 2015
Slides courtesy of Dr. Sang-Eon Park
Bucket Sort

Bucket-Sort(A) {
    1. n = |A|
    2. Let B[0..n-1] be a new array
    3. for i = 0 to n-1
        3.1. make B[i] an empty list
    4. for i = 0 to n-1
        4.1. insert A[i] into the list B[⌊nA[i]]⌋.
    5. for i = 0 to n-1
        5.1. sort bucket B[i] using insertion sort.
    6. concatenate the lists from the buckets B[0], B[1], .., B[|A|-1] together in order.
}
Sorting in Linear Time
(Bucket Sort)

Bucket-Sort(A) {
  1  n = |A|
  2  Let B[0..n-1] be a new array
  3  for i = 0 to n-1
  4    make B[i] an empty list
  5  for i = 0 to n-1
  6    insert A[i] into the list B[⌊ nA[i] ⌋].
  7  for i = 0 to n-1
  8    sort bucket B[i] using insertion sort.
  9  concatenate the lists from the buckets B[0], B[1], ..., B[|A|-1] together in order.
}

COSC 320 - Fall 2015
Slides courtesy of Dr. Sang-Eon Park
Bucket Sort

Bucket-Sort(A) {
    n = |A|
    Let B[0..n-1] be a new array
    for i = 0 to n-1
        make B[i] an empty list
    for i = 0 to n-1
        insert A[i] into the list B[⌊nA[i]⌋].
    for i = 0 to n-1
        sort bucket B[i] using insertion sort.
    concatenate the lists from the buckets B[0], B[1], .., B[|A|-1] together in order.
}

COSC 320 - Fall 2015
Slides courtesy of Dr. Sang-Eon Park
Bucket Sort

Bucket-Sort(A) {
    1 n = |A|
    2 Let B[0..|A|-1] be a new array
    3 for i = 0 to n-1
        4 make B[i] an empty list
        5 for i = 0 to n-1
            6 insert A[i] into the list B[⌊nA[i]⌋].
            7 for i = 0 to n-1
                8 sort bucket B[i] using insertion sort.
                9 concatenate the lists from the buckets B[0], B[1], ..., B[|A|-1] together in order.
}

COSC 320 - Fall 2015
Slides courtesy of Dr. Sang-Eon Park
Sorting in Linear Time
(Bucket Sort)

Bucket-Sort(A) {
    1. n = |A|
    2. Let B[0..n-1] be a new array
    3. for i = 0 to n-1
       4. make B[i] an empty list
    5. for i = 0 to n-1
       6. insert A[i] into the list B[⌊nA[i]⌋].
    7. for i = 0 to n-1
       8. sort bucket B[i] using insertion sort.
    9. concatenate the lists from the buckets B[0], B[1], .., B[|A|-1] together in order.
}

COSC 320 - Fall 2015
Slides courtesy of Dr. Sang-Eon Park
Bucket Sort

Bucket-Sort(A) {
    1 n = |A|
    2 Let B[0..n-1] be a new array
    3 for i = 0 to n-1
        4 make B[i] an empty list
        5 for i = 0 to n-1
            6 insert A[i] into the list B[⌊nA[i]⌋].
            7 for i = 0 to n-1
                8 sort bucket B[i] using insertion sort.
                9 concatenate the lists from the buckets B[0], B[1], .., B[|A|-1] together in order.
}
Sorting in Linear Time  
(Bucket Sort)

Bucket-Sort(A) {
  1  n = |A|
  2  Let B[0..n-1] be a new array
  3  for i = 0 to n-1
  4    make B[i] an empty list
  5  for i = 0 to n-1
  6    insert A[i] into the list B[⌊ nA[i] ⌋].
  7  for i = 0 to n-1
  8    sort bucket B[i] using insertion sort.
  9  concatenate the lists from the buckets B[0], B[1], .., B[|A|-1] together in order.
}

COSC 320 - Fall 2015  
Slides courtesy of Dr. Sang-Eon Park
### Sorting in Linear Time (Bucket Sort)

**Algorithm**: Bucket Sort

1. Let $B[0..n-1]$ be a new array
2. For $i = 0$ to $n-1$
   - Make $B[i]$ an empty list
3. For $i = 0$ to $n-1$
4. For $i = 0$ to $n-1$
   - Sort bucket $B[i]$ using insertion sort.
5. Concatenate the lists from the buckets $B[0], B[1], ..., B[|A|-1]$ together in order.

```c
Bucket-Sort(A) {
    n = |A|
    Let B[0..n-1] be a new array
    for i = 0 to n-1
        make B[i] an empty list
    for i = 0 to n-1
        insert A[i] into the list B[\lfloor nA[i] \rfloor].
    for i = 0 to n-1
        sort bucket B[i] using insertion sort.
    concatenate the lists from the buckets B[0], B[1], ..., B[|A|-1] together in order.
}
```

**Example**:

1. Let $A = [0.34, 0.29, 0.49, 0.56, 0.35, 0.54, 0.66, 0.44, 0.21, 0.19, 0.24, 0.33, 0.83]$.
2. Divide $A$ into 9 buckets based on the range of values.
3. Sort each bucket using insertion sort.
4. Concatenate the sorted buckets to get the final sorted array.

**Sorted Array**: $[0.19, 0.21, 0.24, 0.24, 0.29, 0.33, 0.34, 0.44, 0.49, 0.54, 0.56, 0.66, 0.83]$.
Sorting in Linear Time
(Bucket Sort: Running Time)

Bucket-Sort(A)
{
1. \( n = |A| \)
2. for \( i = 0 \) to \( n-1 \)
3. \hspace{1cm} make \( B[i] \) an empty list
4. for \( i = 0 \) to \( n-1 \)
5. \hspace{1cm} insert \( A[i] \) into correct bucket.
6. for \( i = 0 \) to \( n-1 \)
7. \hspace{1cm} Sort list \( B[i] \) with insertion sort.
8. Concatenate the list of buckets \( B[0], B[1], \ldots B[|A| - 1] \) together in order.
}

\( T(n) = \Theta(n) + \sum_{0}^{n-1} O(n_i^2) \)

- Let \( n_i \) be the random variable denoting the number of elements placed in bucket \( B[i] \).
- Since insertion sort runs in quadratic time, the running time of bucket sort is
We analyze the average-case running time of bucket sort, by computing the expected value of the running time.

We take the expectation over the input distribution.

\[ E[T(n)] = E \left( n \sum_{i=0}^{n-1} 0(n_i^2) \right) \]

\[ = \theta(n) + \sum_{i=0}^{n-1} E[0(n_i^2)] \text{(by linearity of expectation)} \]

\[ = \theta(n) + \sum_{i=0}^{n-1} 0[E(n_i^2)] \]

(by property of linearity of expectation \( E[aX] = aE[X] \))

From equation

\[ E(T(n)) = \theta(n) + \sum_{i=0}^{n-1} 0[E(n_i^2)] \]

We claim that

\[ E(n_i^2) = 2 - \frac{1}{n} \ldots \ldots \ldots \ldots (A) \]

We can prove our claim (A) by defining indicator random variables (skip). If our claim is true, we can conclude the running time of Bucket sort as:

\[ T(n) = \theta(n) + \sum_{i=0}^{n-1} 0(n_i^2) = \theta(n) + n \left( 2 - \frac{1}{n} \right) = \theta(n) \]