Basic Sorting

Analysis of Algorithms
Selection Sort

• Find the smallest number in the list and move it to the correct position

• For each pass of the outer loop:
  • Find the smallest element
  • Exchange the first element in the list with the smallest
  • Increment the first element
# Selection Sort

Selection sort is a simple comparison-based sorting algorithm. It works by repeatedly finding the minimum element from the unsorted part of the list and putting it at the beginning.

## Trace of Selection Sort

<table>
<thead>
<tr>
<th>i</th>
<th>min</th>
<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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>S O R T E X A M P L E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>A O R T E X S M P L E</td>
<td></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>10</td>
<td>A E E R T O X S M P L E</td>
<td></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>9</td>
<td>A E E E T O X S M P L R</td>
<td></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>7</td>
<td>A E E E L O X S M P T R</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>7</td>
<td>A E E E L M X S O P T R</td>
<td></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>8</td>
<td>A E E E L M O S X P T R</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>10</td>
<td>A E E E L M O P X S T R</td>
<td></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>8</td>
<td>A E E E L M O P R S T X</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>9</td>
<td>A E E E L M O P R S T X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>A E E E L M O P R S T X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*entries in black are examined to find the minimum*

*entries in red are $a[\text{min}]$*

*entries in gray are in final position*

Trace of selection sort (array contents just after each exchange)
Selection Sort Implementation

```java
public class Selection {
    public static void sort(Comparable[] a) {
        int N = a.length;
        for (int i = 0; i < N; i++) {
            int min = i;
            for (int j = i+1; j < N; j++)
                if (less(a[j], a[min]))
                    min = j;
            exch(a, i, min);
        }
    }

    private static boolean less(Comparable v, Comparable w) {
        /* as before */
    }

    private static void exch(Comparable[] a, int i, int j) {
        /* as before */
    }
}```
Correctness of Selection Sort

• We can prove that selection sort is correct using *invariants*
  
  – An invariant is a property of the data that is true before the algorithm begins, stays true during execution, and is true when finished:
    1. The entries to the left of the most recently sorted item are in sorted order
    2. No entry to right of this item is smaller than it
Correctness of Selection Sort

- **Algorithm invariants:**
  1. The entries to the left of the most recently sorted item are in sorted order
  2. No entry to right of this item is smaller than it

- **During execution of selection sort:**
  - Move to the next item
  ```
  i++;
  ```
  - Identify the index of minimum item on the right
  ```
  int min = i;
  for (int j = i+1; j < N; j++)
    if (less(a[j], a[min]))
      min = j;
  ```
  - Exchange the next item with the minimum item
  ```
  exch(a, i, min);
  ```
Analysis of Selection Sort

• Selection sort uses \((N-1) + (N-2) + \ldots + 1 \approx \frac{1}{2} N^2\) compares and \(N\) exchanges

• Advantage: Data movement is minimal
  – Linear number of exchanges

• Disadvantage: The running time is insensitive to the input
  – Quadratic time, even if the input is already sorted or close to being sorted
Insertion Sort

• A one element list is sorted
• Insert the next element into its proper place by moving elements down if necessary
• Repeat until all elements have been inserted into the sorted part of the list
• Adding a new element to a sorted list will keep the list sorted if the element is inserted in the correct place
### Insertion Sort

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>OS</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>OS</td>
<td>R</td>
<td>T</td>
<td>S</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>O</td>
<td>S</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>E</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>E</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>A</td>
<td>E</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>A</td>
<td>E</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>A</td>
<td>E</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>A</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>E</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>A</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Trace of Insertion sort (array contents just after each insertion)**

- **entries in gray do not move**
- **entry in red is a[j]**
- **entries in black moved one position right for insertion**
public class Insertion
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        for (int i = 0; i < N; i++)
            for (int j = i; j > 0; j--)
                if (less(a[j], a[j-1]))
                    exch(a, j, j-1);
                else break;
    }

    private static boolean less(Comparable v, Comparable w)
    { /* as before */ }

    private static void exch(Comparable[] a, int i, int j)
    { /* as before */ }
}
Correctness of Insertion Sort

• Invariants:
  1. The entries to the left of the current item to be sorted are in sorted order
  2. Entries to the right of this item have not yet been seen
Correctness of Insertion Sort

• Algorithm invariants:
  1. The entries to the left of the current item to be sorted are in sorted order
  2. Entries to right of this item have not yet been seen

• During execution of insertion sort:
  – Move to the next item
  – Exchange this item with each larger item to its left

```c
for (int j = i; j > 0; j--)
    if (less(a[j], a[j-1]))
        exch(a, j, j-1);
    else break;
```
Analysis of Insertion Sort

**INSERTION-SORT** *(A)*

1. **for** *j = 2* **to** *A.length*
2. **key = A[j]*
3. // Insert *A[j]* into the sorted sequence *A[1..j-1]*.
   - **i = j - 1**
4. **while** *i > 0* **and** *A[i] > key*
5. **i = i - 1**
6. *A[i+1] = key*

**cost**

<table>
<thead>
<tr>
<th></th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>c₁</em></td>
<td><em>n</em></td>
</tr>
<tr>
<td><em>c₂</em></td>
<td><em>n - 1</em></td>
</tr>
<tr>
<td><em>c₄</em></td>
<td><em>n - 1</em></td>
</tr>
<tr>
<td><em>c₅</em></td>
<td>(\sum_{j=2}^{n} t_j)</td>
</tr>
<tr>
<td><em>c₆</em></td>
<td>(\sum_{j=2}^{n} (t_j - 1))</td>
</tr>
<tr>
<td><em>c₇</em></td>
<td>(\sum_{j=2}^{n} (t_j - 1))</td>
</tr>
<tr>
<td><em>c₈</em></td>
<td><em>n - 1</em></td>
</tr>
</tbody>
</table>
Analysis of Insertion Sort

• To compute the running time of insertion sort, sum the products of the cost and times columns:

\[ T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \]

\[ + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1) . \]

• The best case is if the array is already sorted; then \( t_j = 1 \), and the \( c_6 \) and \( c_7 \) terms go away:

\[ T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 (n - 1) + c_8 (n - 1) \]
\[ = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \]
Analysis of Insertion Sort

- If the array is in reverse sorted order—that is, in decreasing order—the worst case results, and we must include the $c_6$, and $c_7$ terms.

- In this case, each $j^{th}$ element must be compared with every other element in the entire sorted subarray $A[1 \ldots j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots n$:

$$
T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \left( \frac{n(n+1)}{2} - 1 \right)
\quad + c_6 \left( \frac{n(n-1)}{2} \right) + c_7 \left( \frac{n(n-1)}{2} \right) + c_8 (n-1)
\quad = \left( \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left( c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n
\quad - (c_2 + c_4 + c_5 + c_8).
$$
Analysis of Insertion Sort

• Best case
  – If the array is in ascending order, insertion sort makes $N-1$ compares and 0 exchanges

• Worst case
  – If the array is in descending order (and there are no duplicates), insertion sort makes $\approx \frac{1}{2} N^2$ compares and $\approx \frac{1}{2} N^2$ exchanges
Analysis of Insertion Sort

- To sort a randomly-ordered array with distinct values, insertion sort uses $\approx \frac{1}{4} N^2$ compares and $\approx \frac{1}{4} N^2$ exchanges on average
  - Each entry is expected to move approximately halfway back
  - Thus, in practice on randomly-ordered data, insertion sort is expected to be about twice as fast as selected sort
Inversions and Insertion Sort

• Definition: An *inversion* is a pair of items that are out of order
  
  • For example, A E E L M O T R X P S has 6 inversions (T,R), (T,P), (R,P), (X,P), (T, S), and (X,S)

• An array is *partially sorted* if the number of inversions is \( \leq \) constant \( \cdot \) \( N \)
  
  • For example, a subarray of size 10 appended to a sorted array of size \( N \) is partially sorted
  • Similarly, an array of size \( N \) with only 10 items out of place is partially sorted

• For a partially sorted array, insertion sort runs in \textit{linear time}
Shellsort

- Shellsort first sorts an array partially, and then uses insertion sort to “finish up”
  - Multiple passes over array are usually necessary
  - Main idea: move entries more than one position at a time by $h$-sorting the array
  - An $h$-sorted array is $h$ interleaved sorted sub-sequences
  - For example, this array is 4-sorted: L E E A M H L E P S O L T S X R because each subarray consisting of every 4th item is sorted:

  $h = 4$

  (L, M, P, T) is sorted
  (E, H, S, S) is sorted
  (E, L, O, X) is sorted
  (A, E, L, R) is sorted
Shellsort

- Shellsort (Shell, 1959): h-sort the array (using insertion sort) for decreasing values of h
- In the example below, the circled items of the same color form a single interleaved list to be sorted
Question: Show the results of each pass of Shellsort using \( h \) values of 5, 2, and 1 with an initial list of [7, 3, 9, 4, 2, 5, 6, 1, 8]. How many comparisons in total are made?

<table>
<thead>
<tr>
<th>Original List:</th>
<th>7 3 9 4 2 5 6 1 8</th>
<th># Comparisons</th>
<th>4 ((7,5), (3,6), (9,1), (4,8))</th>
</tr>
</thead>
<tbody>
<tr>
<td>after 5-sort:</td>
<td>5 3 1 4 2 7 6 9 8</td>
<td>5 + 3 = 8</td>
<td>((5,1,2,6,8)) and ((3,4,7,9))</td>
</tr>
<tr>
<td>after 2-sort:</td>
<td>1 3 2 4 5 7 6 9 8</td>
<td>11</td>
<td>((1,3,2,4,5,7,6,9,8))</td>
</tr>
<tr>
<td>sorted:</td>
<td>1 2 3 4 5 6 7 8 9</td>
<td></td>
<td>___</td>
</tr>
</tbody>
</table>

23 Comparisons total
Shellsort

• Which increment sequence to use?
  – Powers of two?  1, 2, 4, 8, 16, 32, ...
    • No – does not compare elements in even positions with those in odd positions until $h = 1$
  – Powers of two minus one?  1, 3, 7, 15, 31, 63, ...
    • Maybe – Shell suggested this originally
  – $3x + 1$, for $x = 0, 1, 4, 13, 40, 121, 364, ...$ ?
    • Knuth suggested this sequence
    • OK and easy to compute
Shellsort Implementation

```java
public class Shell {
    public static void sort(Comparable[] a) {
        int N = a.length;

        int h = 1;
        while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, ...

        while (h >= 1) {
            // h-sort the array.
            for (int i = h; i < N; i++) {
                for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
                    exch(a, j, j-h);
            }

            h = h/3;
        }
    }

    private static boolean less(Comparable v, Comparable w) {
        /* as before */
    }

    private static boolean exch(Comparable[] a, int i, int j) {
        /* as before */
    }
}
```
Analysis of Shellsort

• The worst-case number of compares used by Shellsort using the 3x+1 increments might be $N^{3/2}$
• The number of compares used by Shellsort using the 3x+1 increments seems to be at most a small multiple of $N$ times the number of increments used (?

<table>
<thead>
<tr>
<th>N</th>
<th>Compares</th>
<th>$N^{1.289}$</th>
<th>2.5 $N \lg N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>93</td>
<td>58</td>
<td>106</td>
</tr>
<tr>
<td>10,000</td>
<td>209</td>
<td>143</td>
<td>230</td>
</tr>
<tr>
<td>20,000</td>
<td>467</td>
<td>349</td>
<td>495</td>
</tr>
<tr>
<td>40,000</td>
<td>1022</td>
<td>855</td>
<td>1059</td>
</tr>
<tr>
<td>80,000</td>
<td>2266</td>
<td>2089</td>
<td>2257</td>
</tr>
</tbody>
</table>

measured in thousands
Analysis of Shellsort

• Question: How many comparisons does Shellsort make (using the 3x + 1 increment sequence) on an input array of size $N$ that is already sorted?
  a) constant
  b) logarithmic
  c) linear
  d) linearithmic ($N \log N$)
Analysis of Shellsort

• Shellsort is useful in practice
  – Fast unless array size is huge
  – Tiny, fixed footprint for code (this is useful for embedded systems)
  – Hardware sort prototype

• Shellsort is a simple algorithm with nontrivial performance and interesting questions
  – What is the asymptotic growth rate?
  – What is the best sequence of increments?
  – What is the average-case performance?
Sorting in Linear Time

• The lowest possible runtime for all sorting algorithms that are based on comparisons is proportional to $N \log N$

• The following sorting algorithms are not based on comparisons, therefore they are considered linear (proportional to $N$):
  – Count sort (key-indexed counting)
  – Radix sort
  – Bucket sort
Count Sort

- Count sort requires three arrays:
  - Input array $a[0 ... N-1]$ of keys
  - Count array $count[0 ... R]$ where $R-1$ is the maximum value of any key ($R$ is called the radix)
  - Auxiliary array to store output $aux[0 ... N-1]$
  - In other words, each key of array $a$ must be a nonnegative integer no larger than $R-1$
Count Sort

Begin by filling up the count array: each element $i$ of count contains the number of elements in $a$ that have the value $i-1$. Note that count[0] = 0.

For example, since there are 2 elements in $a$ that have a value of 0, then count[0 + 1] = count[1] = 2.
Next, accumulate the count array: each element in count becomes the sum of the current element plus all the values that came before. For example, count[6] = 8 because the sum of the original values in count for indices 0 through 6 = 0 + 2 + 0 + 3 + 2 + 0 + 1 = 8.
Next, fill up the aux array by scanning through \( a \), using the value at each location to locate the index in count. The element at that index is used to find the correct location in aux for that value (don’t forget to increment count[index]). Finish by copying aux back to \( a \).
Count Sort

\[ \begin{array}{c}
\alpha \\
\begin{array}{cccccccc}
5 & 2 & 3 & 0 & 2 & 3 & 0 & 2 \\
\end{array}
\end{array} \]

\[ \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

\[ \begin{array}{cccccccc}
\text{count} \\
\begin{array}{cccccccc}
0 & 2 & 2 & 5 & 7 & 8 & 8 \\
\end{array}
\end{array} \]

\[ \begin{array}{cccccccc}
\text{aux} \\
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\end{array} \]

\[ \begin{array}{cccccccc}
\begin{array}{cccccccc}
2 &  &  &  &  &  & 5 \\
\end{array}
\end{array} \]
Count Sort

\[
\begin{array}{cccccccc}
\alpha & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
5 & 2 & 3 & 0 & 2 & 3 & 0 & 2 \\
\text{count} & 0 & 2 & 3 & 5 & 7 & 8 & 8 \\
\text{aux} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]
### Count Sort

<table>
<thead>
<tr>
<th>a</th>
<th>count</th>
<th>aux</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 2 3 0 2 3 0 2</td>
<td>0 2 3 6 7 8 8</td>
<td>0 2 3 5</td>
</tr>
</tbody>
</table>

**Final sorted array**

<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 2 2 2 3 3 5</td>
</tr>
</tbody>
</table>
Count Sort Implementation

```java
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
Analysis of Count Sort

- \( N + R + N + N = R + 3N \)
- If \( R \approx N \) then the runtime of Count Sort is proportional to \( N \)
- Count Sort is a **stable sort**, i.e., the ordering of duplicate keys in the final array is the same as the ordering in the original array
  - Stable sorting is important for satellite data, and is crucial for other linear sorts such as radix sort
Bucket Sort

• Assume the input is drawn from a uniform distribution
  • Each input value is equally likely
• Create an array of “buckets”, where each bucket is the head of a linked list
• The sorted output consists of a concatenation in order of the lists from each of the buckets
Bucket Sort
Bucket Sort

• Question: How would you sort an array of N strings so that all of the anagrams are next to each other?
  – Anagrams are words that have the same characters but in different orders
  – For example: acre, race, and care are all anagrams of each other, so they should be placed next to each other
Radix Sort

• Also known as least-significant digit (LSD) sort
• Sort of choice for many applications that have fixed-length keys
• On each pass we look at a different digits, starting with the rightmost digit
  – On the first pass, we sort the keys (using count sort) based on the “ones” digit
  – On the second pass, we sort the keys based on the “tens” digit,
  – On the third pass we sort them based on the “hundreds” digit, etc.
# Radix Sort

Example of sorting a list of 3-digit numbers:

<table>
<thead>
<tr>
<th>1st pass</th>
<th>2nd pass</th>
<th>3rd pass</th>
<th>Sorted</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>
</tbody>
</table>
Radix Sort

- Notice that the sorting routine must be *stable* - i.e., any digits that are the same must be in the same order relative to one another after they are sorted as they were before.
- Count sort is used as the auxiliary sort for radix sort because it is stable.
- Insertion sort and mergesort are stable, selection sort, quicksort, and Shellsort are not.
Radix Sort Implementation

```java
public class LSD {
    public static void sort(String[] a, int W) {
        int R = 256;
        int N = a.length;
        String[] aux = new String[N];

        for (int d = W-1; d >= 0; d--)
            int[] count = new int[R+1];
        for (int i = 0; i < N; i++)
            count[a[i].charAt(d) + 1]++;
        for (int r = 0; r < R; r++)
            count[r+1] += count[r];
        for (int i = 0; i < N; i++)
            aux[count[a[i].charAt(d)]++] = a[i];
        for (int i = 0; i < N; i++)
            a[i] = aux[i];
    }
}
```
Analysis of Radix Sort

- Each element is examined once for each of the digits it contains, so if the elements have at most $D$ digits (columns) of radix $R$ and there are $N$ elements, then the runtime is proportional to $D \cdot (N + R)$
- This means that sorting is linear based on the number of elements, as long as $D$ is a constant and $R \leq$ constant $\times N$
- Why then isn’t this the only sorting algorithm used?
Analysis of Radix Sort

• Though this is a very time efficient algorithm it is not space efficient

• Radix sort (as well as count sort) requires extra space for the count array
  — The extra space is proportional to $N + R$

• If linked lists are used for the digits you still have the overhead of pointers
Analysis of Radix Sort

Question: Which of the following is the most efficient algorithm to sort 1 million 32-bit integers?

a) Bubblesort
b) Mergesort
c) Quicksort
d) Radix sort
History of Radix Sort

• For the 1880 census, it took 1,500 people 7 years to manually process all of the data

• To automate processing census data, Herman Hollerith developed a sorting machine
  – It used punch cards to record the data
  – The machine sorted one column at a time into one of 12 bins
  – The 1890 census finished months early and under budget!
History of Radix Sort

• Hollerith's company later merged with 3 others to form Computing Tabulating Recording Corporation, later renamed IBM
Sorting Application: Convex Hull

• Computational Geometry
  – The convex hull of a set of N points is the smallest perimeter “fence” enclosing all of the points
  – Equivalent definitions:
    • Smallest convex set containing all the points
    • Smallest area convex polygon enclosing the points
    • Convex polygon enclosing the points, whose vertices are all points in set
Sorting Application: Convex Hull

- Convex hull is used for *robot motion planning*
  - Find the shortest path in the plane from a source, s, to a destination, t, that avoids a polygonal obstacle
  - The shortest path is either a straight line from s to t or it is one of two polygonal chains of the convex hull
Sorting Application: Convex Hull

- Convex hull is used in data mining for finding extreme feature vectors
  - Given N points in the plane, find a pair of points with the largest Euclidean distance between them
  - The farthest pair of points are extreme points on the convex hull
Sorting Application: Convex Hull

- How can we find the vertices on the convex hull?
Sorting Application: Convex Hull

- The convex hull could be traversed by making only counter-clockwise turns, starting at point $p$ (the point with the smallest $y$-coordinate).
- During traversal, the vertices of the convex hull will appear in decreasing order of polar angle with respect to the point $p$.
- Two sorts are necessary:
  - One to find the smallest $y$-coordinate
  - One to create a sorted list of polar angles
Graham Scan Algorithm

1. Find the point $p$ with the smallest $y$-coordinate
2. Sort the points by polar angle with respect to $p$
3. Consider points in order of decreasing polar angle; discard a point unless it makes a counter-clockwise turn
Can We Use Sorting to Shuffle an Array?

One way to shuffle (inefficient):

1. Generate a random real number for each entry

2. Sort the entries based on those real numbers
Is There a Faster Way to Shuffle?

• The Knuth shuffle:
  – Pass through the array once, from left to right
  – For each array element \( i \), pick an integer \( r \) between \( i \) and \( N \) uniformly at random
  – Swap \( \text{array}[i] \) and \( \text{array}[r] \)
  – The Knuth shuffling algorithm produces a uniformly random permutation of the input array in linear time, assuming the integers are uniformly random
Non-Uniform Shuffling

Example with 3 cards – card1, card2, card3, denoted 123 to start:

for \( i := 1 \) to 3

\[ \text{swap card } i \text{ with a card at a random position between 1 and 3} \]
Uniform Shuffling

Example with 3 cards – card1, card2, card3, denoted 123 to start:

for \((i := 1 \text{ to } 3)\)

swap card \(i\) with a card at a random position between \(i\) and 3

1\(^{st}\) iteration

2\(^{nd}\) iteration
Correctness of Shuffling

• Example of the consequences of incorrect shuffling: “How we learned to cheat at online poker: A study in software security”

• Bug 1: Random number $r$ was never 52
  – 52nd card can’t end up in 52nd place

• Bug 2: Shuffle was not uniform
  – $r$ should be between $i$ and 52, not 1 and 52
  – Many card sequences were duplicates

• Bug 3: random() uses 32-bit seed
  – Only $2^{32}$ possible shuffles

• Bug 4: Seed
  – Used milliseconds since midnight
  – Only 86.4 million shuffles
Best Practices for Shuffling

• Use a hardware random-number generator that has passed both the FIPS 140-2 and the NIST statistical test suites

• Continuously monitor statistic properties
  – Hardware random-number generators are fragile and might fail silently

• Use an unbiased shuffling algorithm!