Algorithms: Sorting

Amotz Bar-Noy

CUNY

Spring 2012
The Sorting Problem

- **Keys:** Entities from a **well ordered** domain.

- **Comparison:** For 2 keys $K_1$ and $K_2$
  - either $K_1 < K_2$
  - or $K_2 < K_1$
  - or $K_1 = K_2$.


- **Goal:** Minimize number of comparisons between keys.
Complexity of the Sorting Problem

- \( \Omega(n \log n) \) comparisons lower bound.
- \( O(n^2) \) comparisons simple upper bound.
- \( O(n \log n) \) comparisons upper bound.
- \( \Theta(n \log n) \) overall complexity.

Bounds are for both worst case and average case complexity.
Some Sorting Algorithms

**Simple Algorithms:**
- **Bubble-Sort:** $\Theta(n^2)$ worst & average case.
- **Insertion-Sort:** $\Theta(n^2)$ worst & average case.
- **Selection-Sort:** $\Theta(n^2)$ worst & average case.

**Efficient Sorting Algorithms:**
- **Merge-Sort:** $\Theta(n \log n)$ worst & average case.
- **Quick-Sort:** $\Theta(n \log n)$ average case; $\Theta(n^2)$ worst case.
- **Heap-Sort:** $\Theta(n \log n)$ worst & average case.
- **Binary-Tree-Sort:** $\Theta(n \log n)$ average case; $\Theta(n^2)$ worst case.
- **Balanced-Tree-Sort:** $\Theta(n \log n)$ worst & average case.
Quick-Sort

Divide and Conquer for \( n \geq 2 \):

- **Partition** the array \( A[1..n] \) into two sub-arrays \( A[1..q] \) and \( A[q + 1..n] \) such that all the keys in the sub-array \( A[1..q] \) are smaller or equal to all the keys in the sub-array \( A[q + 1..n] \).

- Recursively **Sort** the sub-arrays \( A[1..q] \) and \( A[q + 1..n] \).
The Partition Procedure


Partition\((p, r)\):

- \( r > p \).
- Return a value \( p \leq q < r \) such that \( A[i] \leq A[j] \) for any \( p \leq i \leq q \) and \( q + 1 \leq j \leq r \).

Method: Pivot partitioning.

- One key is compared with the rest of the keys.
- This key is the \((q - p)\)-smallest in the sub-array \( A[p..r] \).

Complexity: Number of comparisons is exactly \((r - p)\).
The Recursive Quick-Sort Procedure

Initial recursive call: \texttt{Quick-Sort}(1, n).

Recursive procedure:

\begin{verbatim}
Quick-Sort(p, r)
  if \( r > p \) then
    q = Partition(p, r)
    Quick-Sort(p, q)
    Quick-Sort(q + 1, r)
\end{verbatim}
Quick-Sort – Correctness

**Assumption:** Make sure that \( p \leq q < r \).

**Proof:**

- By **induction** on \( r - p \).
- Case \( r = p \) the array is sorted trivially.
- Case \( p \leq q < r \) the **induction hypothesis** is true:
  - For sub-array \( A[p..q] \) since \( q - p < r - p \).
  - For sub-array \( A[q + 1..r] \) since \( r - (q + 1) < r - p \).
- The **induction step** is correct since procedure **Partition** guarantees that all the keys in \( A[p..q] \) are **smaller or equal** to all the keys in \( A[q + 1..r] \).
Quick-Sort – Complexity

- \( T(n) \) - number of comparisons.
- \( T(1) = 0 \).
- \( T(n) \leq T(q) + T(n - q) + (n - 1) \).
Quick-Sort – Complexity

- $T(n)$ - number of comparisons.
- $T(1) = 0$.
- $T(n) \leq T(q) + T(n - q) + (n - 1)$.

**Best:** $T(n) \leq 2T(n/2) + (n - 1) = \Theta(n \log n)$.

**Good:** $T(n) \leq T(n/10) + T(9n/10) + (n - 1) = \Theta(n \log n)$.

**Worst:** $T(n) \geq T(n - 1) + (n - 1) = \Theta(n^2)$. 
A good pivot: Greater than at least $n/4$ keys and smaller than at least $n/4$ keys.

Probabilities facts:
- With probability $1/2$ a random pivot is good.
- Expected number of random pivots until finding a good pivot is 2.
\( \Theta(n \log n) \) Expected Number of Comparisons

- \( \Theta(n) \) to perform one partition.
- \( \Theta(n) \) until a good partition is performed.
- For a recursive inequality of the type
  \[
  T(n) \leq T(\gamma n) + T((1 - \gamma)n) + \Theta(n)
  \]
  the worst case is when \( \gamma \to 1 \).
- \( \gamma \leq 3/4 \) for a good pivot.
- Therefore for Quick-Sort:
  \[
  T(n) \leq T(3n/4) + T(n/4) + \Theta(n) = \Theta(n \log n).
  \]
Solving the Recursive Formula

- **Assumption:** ignore floors and ceilings.
- **Formula:** $T(n) \leq T(3n/4) + T(n/4) + \alpha n$ for constant $\alpha > 0$.
- **Claim:** $T(n) \leq \beta n \log n$ for constant $\beta > 1.25\alpha$. 
Solving the Recursive Formula

**Induction step:**

\[
T(n) \leq \beta \frac{3n}{4} \log_2 \left( \frac{3n}{4} \right) + \beta \frac{n}{4} \log_2 \left( \frac{n}{4} \right) + \alpha n
\]

\[
= \beta \left( \frac{3n}{4} \log_2 n + \frac{n}{4} \log_2 n \right) - \beta \frac{3n}{4} \log_2 \left( \frac{4}{3} \right) - \beta \frac{n}{4} \log_2 n + \alpha n
\]

\[
= \beta n \log_2 n + \left( \alpha - \frac{\beta}{2} - \frac{3\beta}{4} \log_2 \left( \frac{4}{3} \right) \right) n
\]

\[
\leq \beta n \log_2 n.
\]
Solving the Recursive Formula

- The coefficient of $n$ must be negative if $T(n) \leq \beta n \log_2 n$.

- $\alpha < \frac{\beta}{2} + \frac{3\beta}{4} \log_2 \left(\frac{4}{3}\right)$.

- $\beta > \frac{1}{0.5 + 0.75 \log_2(1.333)} \alpha \approx 1.233 \alpha$. 
Quick-Sort – Average Case Complexity

- **Assumption:** For $n \geq 2$ and $1 \leq q < n$, with probability $1/(n - 1)$ the value of $q$ is $1, 2, \ldots, n - 1$.

- **Fix $q$:** $(T(q) + T(n - q))$ comparisons in the recursive calls.

- **Procedure Partition:** Exactly $n - 1$ comparisons.
The recursive formula for average case complexity is given by:

\[
T(n) = (n - 1) + \frac{1}{n-1} \sum_{q=1}^{n-1} (T(q) + T(n - q))
\]

\[
= (n - 1) + \frac{2}{n-1} \sum_{q=1}^{n-1} T(q)
\]

\[
= \Theta(n \log n).
\]
Bounding the Sum $\sum_{q=1}^{n-1} q \ln(q)$

- $f(x) = x \ln(x)$ is a **monotonic non-decreasing** function.

- $\int x \ln(x) \, dx = \frac{x^2 \ln x}{2} - \frac{x^2}{4}$.

\[
\sum_{q=1}^{n-1} q \ln(q) \leq \int_1^n x \ln(x) \, dx
= \frac{1}{2} n^2 \ln(n) - \frac{1}{4} n^2 + \frac{1}{4}.
\]
Solving the Recursive Formula for Quick-Sort

**Induction hypothesis:** \( T(q) \leq cq \ln(q) \) for \( 1 \leq q < n \).

\[
T(n) = (n-1) + \frac{2}{n-1} \sum_{q=1}^{n-1} T(q)
\]

\[
\leq (n-1) + \frac{2c}{n-1} \sum_{q=1}^{n-1} q \ln(q)
\]

\[
\leq (n-1) + \frac{2c}{n-1} \left( \frac{1}{2} n^2 \ln(n) - \frac{1}{4} n^2 + \frac{1}{4} \right)
\]

**Conclusion:** \( T(n) \leq cn \ln(n) \) for some constant \( c \).
Another Method

\[ T(n) = (n - 1) + \frac{2}{n-1} \sum_{q=1}^{n-1} T(q). \]

\[ T(n - 1) = (n - 2) + \frac{2}{n-2} \sum_{q=1}^{n-2} T(q). \]

\[ (n - 1) T(n) - (n - 2) T(n - 1) = (2n - 3) + 2 T(n - 1). \]

\[ (n - 1) T(n) - n T(n - 1) = 2n - 3. \]

\[ \frac{T(n)}{n} - \frac{T(n-1)}{n-1} = \frac{2n-3}{n(n-1)}. \]
Another Method – Continue

- \( S(n) = \frac{T(n)}{n} \) and \( S(1) = 0 \).

- \( S(n) = S(n - 1) + \frac{2n - 3}{n(n-1)} \)

- \( S(n) = \sum_{i=2}^{n} \frac{2i - 3}{i(i-1)} < \sum_{i=2}^{n} \frac{2}{i} = 2H(n) - 2 \leq 2 \ln(n) \).

- \( T(n) = nS(n) \leq 2n \ln(n) \approx 1.386n \log_2(n) \).
The algorithm goal: Find a permutation of \(1, \ldots, n\).
- There are \(n! = n(n - 1)(n - 2) \cdots 2 \cdot 1\) permutations.

The Adversary goal:
- Force the algorithm to have an \(\Omega(n \log n)\) worst case complexity.
- For any algorithm, select a permutation that is found by the algorithm with \(\Omega(n \log n)\) comparisons.
Adversary Strategy

- Maintain a set $S_k$ of all the candidate permutations that are consistent with the first $k$ comparisons.

- Initially, $S_0$ is the set of all $n!$ permutations.

- At the end $S_h$ contains exactly one permutation.

- Let the $k$-th comparison be $(A[i] : A[j])$:
  - $S_k = S'$ if $|S| \leq |S'|$.
  - $S_k = S$ if $|S| > |S'|$. 

Example $n = 4$

Initially, there are $4! = 24$ candidate permutations.

The adversary strategy forces any algorithm to perform at least $\lceil \log_2(24) \rceil = 5$ comparisons:

- After 1 comparison, there are at least 12 candidates.
- After 2 comparisons, there are at least 6 candidates.
- After 3 comparisons, there are at least 3 candidates.
- After 4 comparisons, there are at least 2 candidates.
- After 5 comparisons, the permutation is found.
Example $n = 4$

- Assume the numbers 1, 2, 3, 4 are stored at $x, y, z, w$.

- A permutation is represented by the letters $x, y, z, w$:
  - $x = 2, y = 3, z = 1, w = 4$ implies permutation $zxyw$.
  - Permutation $wyxz$ implies $x = 3, y = 2, z = 4, w = 1$.
  - If $y < z$ then $wyzx$ could be a candidate permutation and $zyxw$ cannot be a candidate permutation.
\[ n = 4: \, S_0 \]

\[
\begin{array}{cccc}
\text{xyzw} & \text{yxzw} & \text{zxyw} & \text{wxyz} \\
\text{xywz} & \text{yxwz} & \text{zxwy} & \text{wxzy} \\
\text{xzyw} & \text{yzxw} & \text{zyxw} & \text{wyxz} \\
\text{xzwy} & \text{yzwx} & \text{zywx} & \text{wyzx} \\
\text{xwyz} & \text{ywzx} & \text{zwxy} & \text{wzxy} \\
\text{xwzy} & \text{ywzx} & \text{zwxy} & \text{wzxy}
\end{array}
\]

\[ \star |S_0| = 24 = 4! \]
$n = 4$: $S_1$ after $x < y$ is True

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$xyzw$</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>$xywz$</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>$xzyw$</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>$xzwy$</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>$xwyz$</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
<tr>
<td>$xwzy$</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
<td>⋆</td>
</tr>
</tbody>
</table>

$|S_1| = 12$ for every comparison and every answer.
$n = 4$: the Comparison is $x < z$

$S_2$ if $x < z$ is true

- $xyzw$
- $xywz$
- $xzyw$
- $xzwy$
- $xwyz$
- $xwzy$

$S_2$ if $z < x$ is true

- $wxyz$
- $wxzy$
- $** **$
- $** **$
- $** **$
- $** **$

$|S_2| = 8$ since the adversary answers $x < z$ true.
$n = 4$: $S_2$ after $z < w$ is True

\[
\begin{array}{cccc}
\text{xyzw} & \ast \ast \ast \ast & \text{zxyw} & \ast \ast \ast \ast \\
\ast \ast \ast \ast & \ast \ast \ast \ast & \text{zxwy} & \ast \ast \ast \ast \\
\text{xzyw} & \ast \ast \ast \ast & \ast \ast \ast \ast & \ast \ast \ast \ast \\
xzwy & \ast \ast \ast \ast & \ast \ast \ast \ast & \ast \ast \ast \ast \\
\ast \ast \ast \ast & \ast \ast \ast \ast & \ast \ast \ast \ast & \ast \ast \ast \ast \\
\ast \ast \ast \ast & \ast \ast \ast \ast & \ast \ast \ast \ast & \ast \ast \ast \ast \\
\end{array}
\]

$|S_2| = 6$ also if $w < z$ is true.
The algorithm finds the permutation with $h$ comparisons.

$S_{k-1} = S \cup S' \Rightarrow |S_k| \geq \frac{|S_{k-1}|}{2}$.

$h \geq \log_2(|S_0|) = \log_2(n!) = \Omega(n \log n)$. 

Lower Bound for the Number of Comparisons
\[ \log_2(n!) = \Omega(n \log n) \]

**Direct approach:**

- \[ n! \geq n(n-1) \cdots \left\lceil \frac{n}{2} \right\rceil \geq \left( \left\lceil \frac{n}{2} \right\rceil \right)^{\left\lceil \frac{n}{2} \right\rceil} \geq \left( \frac{n}{2} \right)^{\frac{n}{2}}. \]
- \[ \log_2(n!) \geq \log_2 \left( \frac{n}{2} \right)^{\frac{n}{2}} = \frac{n}{2} \log_2 \left( \frac{n}{2} \right). \]
- \[ h \geq \log_2(n!) = \Omega(n \log n). \]
\[ \log_2(n!) = \Omega(n \log n) \]

**Direct approach:**

1. \( n! \geq n(n-1) \cdots \left\lceil \frac{n}{2} \right\rceil \geq \left( \left\lceil \frac{n}{2} \right\rceil \right) \geq \left( \frac{n}{2} \right)^{\frac{n}{2}}. \]
2. \( \log_2(n!) \geq \log_2 \left( \frac{n}{2} \right)^{\frac{n}{2}} = \frac{n}{2} \log_2 \left( \frac{n}{2} \right). \)
3. \( h \geq \log_2(n!) = \Omega(n \log n). \)

**Stirling’s approximation:**

1. \( n! = \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \left( 1 + \Theta \left( \frac{1}{n} \right) \right). \)
2. \( h \geq \log_2(n!) = \Theta(n \log n). \)
Comparison Tree Algorithm to Sort 3 Keys

A < B

B < C

A < C

B < A < C

C < B

A < C < B

B < C < A

C < B < A
A full binary tree with $n!$ leaves.

The root represents the first comparison.

Any internal node represents a comparison:
- If the answer is YES continue with the left child.
- If the answer is NO continue with the right child.

A leaf represents a permutation.
Binary Trees

- **Binary tree**: Each internal node has 1 or 2 children.
- **Full binary tree**: Each internal node has exactly 2 children.
  - A $k$-leaves full binary tree has exactly $k - 1$ internal nodes.

- **Heights**: 
  - **Leaf height**: Length of path from the leaf to the root.
  - **Root height**: 0.
  - **Tree height**: The maximum height of one of the leaves.

- **Balance binary trees**: Leaves heights are $h$ or $h + 1$ ($h \geq 1$).
Full Binary Trees: Height and Average Height

**Notations:**

- \( T \): a full binary tree with \( k \) leaves.
- \( h(\ell) \): height of leaf \( \ell \).
- \( h(T) = \max_\ell \{ h(\ell) \} \): height of \( T \).
- \( \hat{h}(T) = \frac{1}{k} \sum_\ell h(\ell) \): average height of \( T \).

**Lemma I:**
\[ h(T) \geq \lceil \log_2 k \rceil \]

**Lemma II:**
\[ \hat{h}(T) = \Omega(\log_2 k) \]
Full Binary Trees: Height and Average Height

**Notations:**
- \( T \): a full binary tree with \( k \) leaves.
- \( h(\ell) \): height of leaf \( \ell \).
- \( h(T) = \max_{\ell} \{h(\ell)\} \): height of \( T \).
- \( \hat{h}(T) = (1/k) \sum_{\ell} h(\ell) \): average height of \( T \).

**Lemma I:** \( h(T) \geq \lceil \log_2 k \rceil \).

**Lemma II:** \( \hat{h}(T) = \Omega(\log_2 k) \).
Example: Height and Average Height

- **Balanced tree:** $h(T) = 2$ and $\hat{h}(T) = 2$.

- **Non-balanced tree:** $h(T) = 3$ and $\hat{h}(T) = 9/4$. 
Proof I:

- The **shortest** full tree with $k$ leaves is the balance full binary tree.
- $h(T) \geq \lceil \log_2 k \rceil$ in a balance full binary tree $T$. 
Proof I:
- The shortest full tree with \( k \) leaves is the balance full binary tree.
- \( h(T) \geq \lceil \log_2 k \rceil \) in a balance full binary tree \( T \).

Proof II:
- The tree with the smallest average height among all full trees with \( k \) leaves is the balance full binary tree.
- \( \hat{h}(T) \geq \lfloor \log_2 k \rfloor \) in a balance full binary tree \( T \).
Why Balanced Trees are the Shortest?

- **Transform** a non-balanced tree to a balanced tree by **reducing** the height of **tall** leaves and **increasing** the height of **short** leaves while preserving the number of leaves.

- **Assume** there are 2 leaves $A$ and $B$ of height $x$ and one leaf $C$ of height $x - 2$.

- **Replace** these 3 leaves with the parent $D$ of $A$ and $B$ of height $x - 1$ and move $A$ and $B$ to be 2 new children of $C$ of height $x - 1$. 
Why Balanced Trees are the Shortest?

The proofs follow since

- $x > x - 1$ for the maximum height.

- $(x - 2) + (x - 1) + 2x > 4(x - 1)$ for the average height.
Any deterministic sorting algorithm that sorts \( n \) keys can be represented by a comparison tree with \( n! \) leaves.

The height of the comparison tree is the worst case number of comparisons performed by the algorithm.

**Lemma I** implies that any deterministic sorting algorithm must perform \( \lceil \log_2(n!) \rceil = \Omega(n \log n) \) comparisons.
Any randomized sorting algorithm that sorts $n$ keys can be represented by a comparison tree with $n!$ leaves.

The average height of the comparison tree is the average number of comparisons performed by the algorithm.

Lemma II implies that any randomized sorting algorithm must perform $\Omega(n \log n)$ comparisons.
Sort in Linear Time

- **Idea:** Sort without comparisons using memory locations.

- **Complexity:** Sometimes $o(n \log n)$ and even $O(n)$ operations for sorting an array of $n$ keys.

- **A contradiction? No!**
  - A different model.
  - A bounded range for the keys.
Bucket Sort

- **Input:**
  - Keys belong to a **bounded** domain of size \( k \).


- **Idea:** For each value between 1 and \( k \), **count** the number of times it appears in \( A \) and then **rearrange** \( A \).

- **Complexity:** \( \Theta(n + k) \) operations.
Bucket Sort – Implementation

Bucket-Sort\(\langle A[1], \ldots, A[n] \rangle\)

\[
\begin{align*}
&\text{for } i = 1 \text{ to } k \text{ do } B[i] = 0 \quad (\text{* prepare } k \text{ empty buckets } \ast) \\
&\text{for } j = 1 \text{ to } n \text{ do } B[A[j]] = B[A[j]] + 1 \\
&\quad (\text{* fill the buckets } \ast) \\
&j = 0 \\
&\text{for } i = 1 \text{ to } k \text{ do } \quad (\text{* spill the buckets } \ast) \\
&\quad \text{while } B[i] > 0 \text{ do } \quad (\text{* spill the buckets } \ast) \\
&\quad \quad j = j + 1; \quad A[j] = i; \quad B[i] = B[i] - 1
\end{align*}
\]

Complexity: \(\Theta(k) + \Theta(n) + \Theta(n + k) = \Theta(n + k)\).
Stable Sorting Algorithms

- A sorting algorithm is stable:
  - If keys with the same values appear in the output array in the same order as they do in the input array.

- Important when satellite data are carried with the keys.

- Bucket Sort is stable: Crucial for Radix Sort.

- Most of the sorting algorithms can be implemented stable.
Tuples as Keys

- For positive integers $d, k$:
  - A key is a tuple $\langle d_1, \ldots, d_d \rangle$ of $d$ digits.
  - Digits from the range $[1..k]$.
  - Keys from the range $[1..k^d]$.
  - $d_1$ is the most significant digit.
  - $d_d$ is the least significant digit.
Lexicographic Order of Tuples

\[ \langle d_1, \ldots, d_d \rangle < \langle d'_1, \ldots, d'_d \rangle \text{ if} \]

1. \(d'_1 < d'_1\): (* 1999... < 2111... *)
2. \(d_1 = d'_1 \text{ and } d_2 < d'_2\): (* 12999... < 13111... *)
3. \(\vdots\)
4. \(\forall 1 \leq i < j < d \ d_i = d'_i \text{ and } d_j < d'_j\): (* 12...91999... < 12...92111... *)
5. \(\vdots\)
6. \(\forall 1 \leq i < d \ d_i = d'_i \text{ and } d_d < d'_d\): (* 12...88 < 12...89 *)
Lexicographic Sort of Tuples

Lexicographic-Sort($A[1], \ldots, A[n]$)

for $i = 1$ to $d$ do
    sort $A$ on digit $i$. 

Correctness: By definition of lexicographic order.

Implementation: A complicated memory handling.

Complexity: $\Theta(d(n+k))$ using Bucket-Sort.
Lexicographic Sort of Tuples

Lexicographic-Sort($A[1], \ldots, A[n]$)
for $i = 1$ to $d$ do
  sort $A$ on digit $i$.

- **Correctness:** By definition of lexicographic order.
- **Implementation:** A complicate memory handling.
- **Complexity:** $\Theta(d(n + k))$ using Bucket-Sort.
Radix Sort of Tuples

Radix-Sort($A[1], \ldots, A[n]$)

for $i = d$ downto 1 do

apply a stable sort to sort $A$ on digit $i$.

Correctness:
By induction on the digit being sorted relying on the
stability of the digit sort.

Implementation:
Easy due to the stability of the digit sorting.

Complexity:
$\Theta(d(n + k))$ using Bucket-Sort.
Radix Sort of Tuples

Radix-Sort($A[1], \ldots, A[n]$)
for $i = d$ downto 1 do
    apply a stable sort to sort $A$ on digit $i$.

**Correctness:** By induction on the digit being sorted relying on the stability of the digit sort.

**Implementation:** Easy due to the stability of the digit sorting.

**Complexity:** $\Theta(d(n + k))$ using Bucket-Sort.
Example

4555
4432
3345
7942
6168
2173
1741
1629
9733
8258
2199
<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>4555</td>
</tr>
<tr>
<td>4432</td>
</tr>
<tr>
<td>3345</td>
</tr>
<tr>
<td>7942</td>
</tr>
<tr>
<td>6168</td>
</tr>
<tr>
<td>2173</td>
</tr>
<tr>
<td>1741</td>
</tr>
<tr>
<td>1629</td>
</tr>
<tr>
<td>9733</td>
</tr>
<tr>
<td>8258</td>
</tr>
<tr>
<td>2199</td>
</tr>
</tbody>
</table>
## Example

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4555</td>
<td>1741</td>
<td>1629</td>
<td></td>
</tr>
<tr>
<td>4432</td>
<td>4432</td>
<td>4432</td>
<td></td>
</tr>
<tr>
<td>3345</td>
<td>7942</td>
<td>9733</td>
<td></td>
</tr>
<tr>
<td>7942</td>
<td>2173</td>
<td>1741</td>
<td></td>
</tr>
<tr>
<td>6168</td>
<td>9733</td>
<td>7942</td>
<td></td>
</tr>
<tr>
<td>2173</td>
<td>4555</td>
<td>3345</td>
<td></td>
</tr>
<tr>
<td>1741</td>
<td>3345</td>
<td>4555</td>
<td></td>
</tr>
<tr>
<td>1629</td>
<td>6168</td>
<td>8258</td>
<td></td>
</tr>
<tr>
<td>9733</td>
<td>8258</td>
<td>6168</td>
<td></td>
</tr>
<tr>
<td>8258</td>
<td>1629</td>
<td>2173</td>
<td></td>
</tr>
<tr>
<td>2199</td>
<td>2199</td>
<td>2199</td>
<td></td>
</tr>
</tbody>
</table>
Example

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4555</td>
<td>1741</td>
<td>1629</td>
<td>6168</td>
<td></td>
</tr>
<tr>
<td>4432</td>
<td>4432</td>
<td>4432</td>
<td>2173</td>
<td></td>
</tr>
<tr>
<td>3345</td>
<td>7942</td>
<td>9733</td>
<td>2199</td>
<td></td>
</tr>
<tr>
<td>7942</td>
<td>2173</td>
<td>1741</td>
<td>8258</td>
<td></td>
</tr>
<tr>
<td>6168</td>
<td>9733</td>
<td>7942</td>
<td>3345</td>
<td></td>
</tr>
<tr>
<td>2173</td>
<td>4555</td>
<td>3345</td>
<td>4432</td>
<td></td>
</tr>
<tr>
<td>1741</td>
<td>3345</td>
<td>4555</td>
<td>4555</td>
<td></td>
</tr>
<tr>
<td>1629</td>
<td>6168</td>
<td>8258</td>
<td>1629</td>
<td></td>
</tr>
<tr>
<td>9733</td>
<td>8258</td>
<td>6168</td>
<td>9733</td>
<td></td>
</tr>
<tr>
<td>8258</td>
<td>1629</td>
<td>2173</td>
<td>1741</td>
<td></td>
</tr>
<tr>
<td>2199</td>
<td>2199</td>
<td>2199</td>
<td>7942</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>4555</td>
<td>1741</td>
<td>1629</td>
<td>6168</td>
<td>1629</td>
</tr>
<tr>
<td>4432</td>
<td>4432</td>
<td>4432</td>
<td>2173</td>
<td>1741</td>
</tr>
<tr>
<td>3345</td>
<td>7942</td>
<td>9733</td>
<td>2199</td>
<td>2173</td>
</tr>
<tr>
<td>7942</td>
<td>2173</td>
<td>1741</td>
<td>8258</td>
<td>2199</td>
</tr>
<tr>
<td>6168</td>
<td>9733</td>
<td>7942</td>
<td>3345</td>
<td>3345</td>
</tr>
<tr>
<td>2173</td>
<td>4555</td>
<td>3345</td>
<td>4432</td>
<td>4432</td>
</tr>
<tr>
<td>1741</td>
<td>3345</td>
<td>4555</td>
<td>4555</td>
<td>4555</td>
</tr>
<tr>
<td>1629</td>
<td>6168</td>
<td>8258</td>
<td>1629</td>
<td>6168</td>
</tr>
<tr>
<td>9733</td>
<td>8258</td>
<td>6168</td>
<td>9733</td>
<td>7942</td>
</tr>
<tr>
<td>8258</td>
<td>1629</td>
<td>2173</td>
<td>1741</td>
<td>8258</td>
</tr>
<tr>
<td>2199</td>
<td>2199</td>
<td>2199</td>
<td>7942</td>
<td>9733</td>
</tr>
</tbody>
</table>
**Induction claim:** After sorting digit $i$, the suffixes of length $d - i + 1$ of all $n$ tuples are sorted.

**Verifying the claim for $i = d$:** By definition of sorting.

**Induction hypothesis:** Claim is true for length $d - i$ suffixes.

**Induction step:** Due to the stability of the digit sort, the induction claim is true for suffixes of length $d - i + 1$. 
Radix Sort of Integers

- Keys: Tuples of $d$ digits each from the range $[0..k - 1]$.
- Set $k = O(n)$.
- Keys are integers from the range $[0..(O(n))^d]$.
- $\Theta(d(n + k))$ complexity becomes $\Theta(dn)$ complexity.
- Constant $d$ implies a linear time algorithm.
- Bucket-Sort is linear only for a range $[1..O(n)]$. 