Quicksort

• Like Mergesort:
  • Divide and conquer
  • $O(n \log n)$ run time (kind of...)

• Unlike Mergesort:
  • Divide step is the “hard” part
  • *Typically* faster than Mergesort
Quicksort

Idea: pick a \textit{pivot} element, recursively sort two sublists around that element

- \textbf{Divide}: select \textit{pivot} element $p$, $\text{Partition}(p)$
- \textbf{Conquer}: recursively sort left and right sublists
- \textbf{Combine}: Nothing!
Partition (Divide step)

Given: a list, a pivot $p$

Start: unordered list

| 8 | 5 | 7 | 3 | 12 | 10 | 1 | 2 | 4 | 9 | 6 | 11 |

Goal: All elements $< p$ on left, all $> p$ on right

| 5 | 7 | 3 | 1 | 2 | 4 | 6 | 8 | 12 | 10 | 9 | 11 |
Partition, Procedure

If \textbf{Begin} value < \textit{p}, move \textbf{Begin} right
Else swap \textbf{Begin} value with \textbf{End} value, move \textbf{End} Left
Done when \textbf{Begin} = \textbf{End}
Partition, Procedure

If \texttt{Begin value} < p, move \texttt{Begin} right
Else swap \texttt{Begin} value with \texttt{End} value, move \texttt{End} Left
Done when \texttt{Begin} = \texttt{End}
Partition, Procedure

If \( \text{Begin} \) value \(<\ p \), move \( \text{Begin} \) right
Else swap \( \text{Begin} \) value with \( \text{End} \) value, move \( \text{End} \) left
Done when \( \text{Begin} = \text{End} \)

Case 1: meet at element \(< p \\
Swap \( p \) with pointer position (2 in this case)
Partition, Procedure

If $\text{Begin}$ value $< p$, move $\text{Begin}$ right
Else swap $\text{Begin}$ value with $\text{End}$ value, move $\text{End}$ Left
Done when $\text{Begin} = \text{End}$

Case 2: meet at element $> p$
Swap $p$ with value to the left (2 in this case)
Partition Summary

1. Put $p$ at beginning of list
2. Put a pointer (Begin) just after $p$, and a pointer (End) at the end of the list
3. While Begin < End:
   1. If Begin value < $p$, move Begin right
   2. Else swap Begin value with End value, move End Left
4. If pointers meet at element < $p$: Swap $p$ with pointer position
5. Else If pointers meet at element > $p$: Swap $p$ with value to the left

Run time? $O(n)$
Conquer

Recursively sort Left and Right sublists

All elements < $p$

All elements > $p$

Exactly where it belongs!
Quicksort Run Time (Best)

If the **pivot** is always the median:

Then we divide in half each time

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

\[ T(n) = O(n \log n) \]
Quicksort Run Time (Worst)

If the pivot is always at the extreme:

Then we shorten by 1 each time

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

\[ T(n) = O(n^2) \]
Quicksort Run Time (Worst)

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

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

\[ T(n) = \frac{n(n + 1)}{2} \]

\[ T(n) = O(n^2) \]
Quicksort on a (nearly) Sorted List

First element always yields unbalanced pivot

So we shorten by 1 each time

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

\[ T(n) = O(n^2) \]
Good Pivot

• What makes a good Pivot?
  • Roughly even split between left and right
  • Ideally: median

• There are ways to find the median in linear time, but it’s complicated and slow and you’re better off using mergesort

• In Practice:
  • Pick a random value as a pivot
  • Pick the middle of 3 random values as the pivot
Properties of Quick Sort

• Worst Case Running time:
  • $\Theta(n^2)$
  • But $\Theta(n \log n)$ average! And typically faster than mergesort!

• In-Place?
  • ....Debatable

• Adaptive?
  • No!

• Stable?
  • No!
More Formal Definition

• Input:
  • An array $A$ of items
  • A comparison function for these items
    • Given two items $x$ and $y$, we can determine whether $x < y$, $x > y$, or $x = y$

• Output:
  • A permutation of $A$ such that if $i \leq j$ then $A[i] \leq A[j]$
  • Permutation: a sequence of the same items but perhaps in a different order
Improving Running time

• Recall our definition of the sorting problem:
  • Input:
    • An array \( A \) of items
    • A comparison function for these items
      • Given two items \( x \) and \( y \), we can determine whether \( x < y \), \( x > y \), or \( x = y \)
  • Output:
    • A permutation of \( A \) such that if \( i \leq j \) then \( A[i] \leq A[j] \)

• Under this definition, it is impossible to write an algorithm faster than \( n \log n \) asymptotically.

• Observation:
  • Sometimes there might be ways to determine the position of values without comparisons!
“Linear Time” Sorting Algorithms

• Useable when you are able to make additional assumptions about the contents of your list (beyond the ability to compare)
  • Examples:
    • The list contains only positive integers less than $k$
    • The number of distinct values in the list is much smaller than the length of the list

• The running time expression will always have a term other than the list’s length to account for this assumption
  • Examples:
    • Running time might be $\Theta(k \cdot n)$ where $k$ is the range/count of values
BucketSort

• Assumes the array contains integers between 0 and $k - 1$ (or some other small range)

• Idea:
  • Use each value as an index into an array of size $k$
  • Add the item into the “bucket” at that index (e.g. linked list)
  • Get sorted array by “appending” all the buckets
BucketSort Running Time

• Create array of $k$ buckets
  • Either $\Theta(k)$ or $\Theta(1)$ depending on some things...
• Insert all $n$ things into buckets
  • $\Theta(n)$
• Empty buckets into an array
  • $\Theta(n + k)$
• Overall:
  • $\Theta(n + k)$
• When is this better than mergesort?
Properties of BucketSort

- In-Place?
  - No

- Adaptive?
  - No

- Stable?
  - Yes!
RadixSort

• Radix: The base of a number system
  • We’ll use base 10, most implementations will use larger bases

• Idea:
  • BucketSort by each digit, one at a time, from least significant to most significant

Place each element into a “bucket” according to its 1’s place
RadixSort

- Radix: The base of a number system
  - We’ll use base 10, most implementations will use larger bases

- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

<table>
<thead>
<tr>
<th>800</th>
<th>801</th>
<th>401</th>
<th>101</th>
<th>901</th>
<th>121</th>
<th>9</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>100</td>
<td>801</td>
<td>401</td>
<td>101</td>
<td>901</td>
<td>121</td>
<td>9</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>
</tr>
</tbody>
</table>

Place each element into a “bucket” according to its 10’s place
RadixSort

• Radix: The base of a number system
  • We’ll use base 10, most implementations will use larger bases

• Idea:
  • BucketSort by each digit, one at a time, from least significant to most significant

Place each element into a “bucket” according to its 100’s place
RadixSort

• Radix: The base of a number system
  • We’ll use base 10, most implementations will use larger bases

• Idea:
  • BucketSort by each digit, one at a time, from least significant to most significant

<table>
<thead>
<tr>
<th>018</th>
<th>101</th>
<th>103</th>
<th>113</th>
<th>121</th>
<th>245</th>
<th>255</th>
<th>323</th>
<th>401</th>
<th>512</th>
<th>555</th>
<th>800</th>
<th>801</th>
<th>823</th>
<th>901</th>
<th>999</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Convert back into an array
RadixSort Running Time

• Suppose largest value is $m$
• Choose a radix (base of representation) $b$
• BucketSort all $n$ things using $b$ buckets
  • $\Theta(n + k)$
• Repeat once per each digit
  • $\log_b m$ iterations
• Overall:
  • $\Theta(n \log_b m + b \log_b m)$
• In practice, you can select the value of $b$ to optimize running time
• When is this better than mergesort?
ARPANET
Undirected Graphs

Definition: $G = (V, E)$

Vertices/Nodes

$V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Edges

$E = \{(1, 2), (2, 3), (1, 3), \ldots \}$
Directed Graphs

Definition: \( G = (V, E) \)

Vertices/Nodes

\( V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \)

Edges

\( E = \{(1,2), (2,3), (1,3), \ldots\} \)
Self-Edges and Duplicate Edges

Some graphs may have duplicate edges (e.g. here we have the edge (1,2) twice). Some may also have self-edges (e.g. here there is an edge from 1 to 1). Graph with Neither self-edges nor duplicate edges are called simple graphs
Weighted Graphs

Definition: \( G = (V, E) \)

\( w(e) = \text{weight of edge } e \)

Vertices/Nodes

\( V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \)

Edges

\( E = \{(1, 2), (2, 3), (1, 3), \ldots\} \)
Graph Applications

- For each application below, consider:
  - What are the nodes, what are the edges?
  - Is the graph directed?
  - Is the graph simple?
  - Is the graph weighted?
- Facebook friends
- Twitter followers
- Java inheritance
- Airline Routes
Some Graph Terms

- **Adjacent/Neighbors**
  - Nodes are adjacent/neighbors if they share an edge

- **Degree**
  - Number of “neighbors” of a vertex

- **Indegree**
  - Number of incoming neighbors

- **Outdegree**
  - Number of outgoing neighbors
Graph Operations

• To represent a Graph (i.e. build a data structure) we need:
  • Add Edge
  • Remove Edge
  • Check if Edge Exists
  • Get Neighbors (incoming)
  • Get Neighbors (outgoing)
Adjacency List

Time/Space Tradeoffs
Space to represent: $\Theta(n + m)$
Add Edge: $\Theta(1)$
Remove Edge: $\Theta(1)$
Check if Edge Exists: $\Theta(n)$
Get Neighbors (incoming): $\Theta(n + m)$
Get Neighbors (outgoing): $\Theta(\text{deg}(v))$

$|V| = n$
$|E| = m$
Adjacency List (Weighted)

Time/Space Tradeoffs
Space to represent: $\Theta(n + m)$
Add Edge: $\Theta(1)$
Remove Edge: $\Theta(1)$
Check if Edge Exists: $\Theta(n)$
Get Neighbors (incoming): $\Theta(?)$
Get Neighbors (outgoing): $\Theta(?)$

$|V| = n$
$|E| = m$
Adjacency Matrix

Time/Space Tradeoffs
Space to represent: $\Theta(\cdot)$
Add Edge: $\Theta(\cdot)$
Remove Edge: $\Theta(\cdot)$
Check if Edge Exists: $\Theta(\cdot)$
Get Neighbors (incoming): $\Theta(\cdot)$
Get Neighbors (outgoing): $\Theta(\cdot)$

$|V| = n$
$|E| = m$
Adjacency Matrix (weighted)

Time/Space Tradeoffs
Space to represent: $\Theta(n^2)$
Add Edge: $\Theta(1)$
Remove Edge: $\Theta(1)$
Check if Edge Exists: $\Theta(1)$
Get Neighbors (incoming): $\Theta(n)$
Get Neighbors (outgoing): $\Theta(n)$

$|V| = n$
$|E| = m$
Aside

• Almost always, adjacency lists are the better choice
• Most graphs are missing most of their edges, so the adjacency list is much more space efficient and the slower operations aren’t that bad
Definition: Path

A sequence of nodes \((v_1, v_2, ..., v_k)\) s.t. \(\forall 1 \leq i \leq k - 1, (v_i, v_{i+1}) \in E\)

Simple Path:
A path in which each node appears at most once

Cycle:
A path which starts and ends in the same place
Definition: (Strongly) Connected Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from $v_1$ to $v_2$
Definition: (Strongly) Connected Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from $v_1$ to $v_2$
Definition: Weakly Connected Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from $v_1$ to $v_2$ ignoring direction of edges.

Weakly Connected

Weakly Connected
Definition: Complete Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is an edge from $v_1$ to $v_2$
Graph Density, Data Structures, Efficiency

• The maximum number of edges in a graph is $\Theta(|V|^2)$:
  - Undirected and simple: $\frac{|V|(|V|-1)}{2}$
  - Directed and simple: $|V|(|V|-1)$
  - Directed and non-simple (but no duplicates): $|V|^2$

• If the graph is connected, the minimum number of edges is $|V|-1$

• If $|E| \in \Theta(|V|^2)$ we say the graph is dense

• If $|E| \in \Theta(|V|)$ we say the graph is sparse

• Because $|E|$ is not always near to $|V|^2$ we do not typically substitute $|V|^2$ for $|E|$ in running times, but leave it as a separate variable
Definition: Tree

A Graph $G = (V, E)$ is a tree if it is undirected, connected, and has no cycles (i.e. is acyclic). Often one node is identified as the “root”