“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

<table>
<thead>
<tr>
<th>103</th>
<th>801</th>
<th>401</th>
<th>323</th>
<th>255</th>
<th>823</th>
<th>999</th>
<th>101</th>
<th>113</th>
<th>901</th>
<th>555</th>
<th>512</th>
<th>245</th>
<th>800</th>
<th>018</th>
<th>121</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

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></th>
<th>800</th>
<th>801</th>
<th>401</th>
<th>101</th>
<th>901</th>
<th>121</th>
<th>512</th>
<th>103</th>
<th>323</th>
<th>823</th>
<th>113</th>
<th>255</th>
<th>555</th>
<th>245</th>
<th>018</th>
<th>999</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>800</td>
<td>801</td>
<td>401</td>
<td>101</td>
<td>901</td>
<td>121</td>
<td>512</td>
<td>103</td>
<td>323</td>
<td>823</td>
<td>113</td>
<td>255</td>
<td>555</td>
<td>245</td>
<td>018</td>
<td>999</td>
</tr>
<tr>
<td>1</td>
<td>512</td>
<td>103</td>
<td>323</td>
<td>823</td>
<td>113</td>
<td>255</td>
<td>555</td>
<td>245</td>
<td>018</td>
<td>999</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>800</td>
<td>801</td>
<td>401</td>
<td>101</td>
<td>901</td>
<td>121</td>
<td>512</td>
<td>103</td>
<td>323</td>
<td>823</td>
<td>113</td>
<td>255</td>
<td>555</td>
<td>245</td>
<td>018</td>
<td>999</td>
</tr>
<tr>
<td>3</td>
<td>255</td>
<td>555</td>
<td>245</td>
<td>018</td>
<td>999</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>255</td>
<td>555</td>
<td>245</td>
<td>018</td>
<td>999</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>255</td>
<td>555</td>
<td>245</td>
<td>018</td>
<td>999</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>255</td>
<td>555</td>
<td>245</td>
<td>018</td>
<td>999</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>255</td>
<td>555</td>
<td>245</td>
<td>018</td>
<td>999</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>255</td>
<td>555</td>
<td>245</td>
<td>018</td>
<td>999</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>255</td>
<td>555</td>
<td>245</td>
<td>018</td>
<td>999</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>

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

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?
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 $E = \{(1, 2), (2, 3), (1, 3), \ldots\}$
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). Graphs with neither self-edges nor duplicate edges are called **simple graphs**.
Weighted Graphs

Definition: $G = (V, E)$

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

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

$E = \{(1, 2), (2, 3), (1, 3), \ldots\}$
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
  • Nodes: Accounts, Edges: Friendship
  • Undirected
  • Simple
  • maybe

• Twitter followers
  • Nodes: Accounts, Edges: following
  • Directed
  • Simple
  • maybe

• Java inheritance
  • Nodes: Classes, Edges: extends, implements
  • Directed
  • Simple
  • Unweight

• Airline Routes
  • Nodes: Cities, edges: flights
  • Directed
  • Non-simple
  • weight
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)
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): $\Theta(\text{deg}(v))$

$|V| = n$
$|E| = m$
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: $\Theta (?)$
Add Edge: $\Theta (?)$
Remove Edge: $\Theta (?)$
Check if Edge Exists: $\Theta (?)$
Get Neighbors (incoming): $\Theta (?)$
Get Neighbors (outgoing): $\Theta (?)$

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

![Graph Image]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

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 \((v_1, v_2, \ldots, v_k)\)

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.
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”
Breadth-First Search

• Input: a node \( s \)
• Behavior: Start with node \( s \), visit all neighbors of \( s \), then all neighbors of neighbors of \( s \), ...
• Output:
  • How long is the shortest path?
  • Is the graph connected?
void bfs(graph, s){
    found = new Queue();
    found.enqueue(s);
    mark s as “visited”;
    While (!found.isEmpty()){
        current = found.dequeue();
        for (v : neighbors(current)){
            if (!v marked “visited”){
                mark v as “visited”;
                found.enqueue(v);
            }
        }
    }
}

Running time: $\Theta(|V| + |E|)$
Shortest Path (unweighted)

```java
int shortestPath(graph, s, t){
    found = new Queue();
    layer = 0;
    found.enqueue(s);
    mark s as “visited”;
    While (!found.isEmpty()){
        current = found.dequeue();
        layer = depth of current;
        for (v : neighbors(current)){
            if (!v marked “visited”){
                mark v as “visited”;
                depth of v = layer + 1;
                found.enqueue(v);
            }
        }
    }
    return depth of t;
}
```

Idea: when it’s seen, remember its “layer” depth!
Depth-First Search
Depth-First Search

- **Input:** a node $s$
- **Behavior:** Start with node $s$, visit one neighbor of $s$, then all nodes reachable from that neighbor of $s$, then another neighbor of $s$, ...
- **Output:**
  - Does the graph have a cycle?
  - A **topological sort** of the graph.
void dfs(graph, s){
    found = new Stack();
    found.pop(s);
    mark s as “visited”; 
    While (!found.isEmpty()){
        current = found.pop();
        for (v : neighbors(current)){
            if (! v marked “visited”){
                mark v as “visited”;
                found.push(v);
            }
        }
    }
}
DFS Recursively (more common)

```c
void dfs(graph, curr){
    mark curr as “visited”;  
    for (v : neighbors(current)){
        if (! v marked “visited”){
            dfs(graph, v);
        }
    }
    mark curr as “done”;  
}
```