CS16 Section 4
Monday, June 21st - Wednesday, June 23rd
Agenda

1. Fun tree poems
2. Sorting
3. Selection
4. Graphs
5. Cycles
Introduction and Pleasant-Trees

Go through the room and talk about something confusing, cool, or interesting from what you’ve seen this week. Then, if you want, share your poem with the class!
Sorting Overview
Insertion Sort

- Builds a sorted array one at a time
- Compares each item to all items before it and swaps current item with any item larger than it
- Stops when reaches item that is smaller than current item to prevent rechecking the already-sorted portion of the array
- **TLDR:** Takes each element, INSERTS it as close to the beginning as possible
Insertion Sort, cont.

- Runtime: $O(n^2)$
- Space Complexity: $O(1)$
- Advantages?
- Disadvantages?

6 5 3 1 8 7 2 4
Selection Sort

- Maintains 2 arrays: sorted and unsorted
- Finds the minimum of the unsorted array at each iteration and places the element at the beginning of the unsorted array, making the unsorted array shorter by 1 element and the sorted array longer by 1 element

**TLDR:** SELECTS the minimum element each round and transfers to 2nd array
Selection Sort, cont.

- Runtime: $O(n^2)$
- Space Complexity: $O(1)$
- Advantages?
- Disadvantages?
Merge Sort

- Splitting an array using divide and conquer
  - Build the array back up in ascending or descending order
- Runtime: $O(n \log(n))$
- Space Complexity: $O(n)$
- Advantages?
- Disadvantages?
QuickSort

- Divide and Conquer
  - Picking a random pivot point and placing elements larger, smaller, and equal to pivot in different arrays

- Runtime:
  - Expected → $O(n \log(n))$
  - Worst Case → $O(n^2)$
    - What happens in this case?

- Space Complexity: $O(n)$
- Advantages?
- Disadvantages?
Radix Sort

- Start with lowest-order digit, move onto next order after each iteration
  - Add each number to a bucket
  - Concatenate the buckets to sort corresponding to the digit
- Runtime: $O(nd)$
  - Runtime is dominated by number of digits
- Space Complexity: $O(n + k)$
  - $k$ is the number of buckets
<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insertion Sort</strong></td>
<td>● Can be done in-place (uses constant memory)</td>
<td>● Very slow on larger datasets</td>
</tr>
<tr>
<td></td>
<td>● Fast if items are partially sorted</td>
<td></td>
</tr>
<tr>
<td><strong>Selection Sort</strong></td>
<td>● Can be done in-place</td>
<td>● Very slow on larger datasets</td>
</tr>
<tr>
<td></td>
<td>● Good for small datasets</td>
<td></td>
</tr>
<tr>
<td><strong>Merge Sort</strong></td>
<td>● Faster for larger datasets</td>
<td>● Uses more memory than other sorts</td>
</tr>
<tr>
<td><strong>Quicksort</strong></td>
<td>● Can be done in place</td>
<td>● Can be really slow if worst case</td>
</tr>
<tr>
<td></td>
<td>● Expected runtime</td>
<td></td>
</tr>
<tr>
<td><strong>Radix Sort</strong></td>
<td>● Can be very efficient O(nd)</td>
<td>● Requires extra memory</td>
</tr>
</tbody>
</table>
Selection
Quickselect (Hoare's Selection)

- Divide and conquer algorithm
  - divide: pick random element $p$ (called pivot) and partition set into
    - $L$: elements less than $p$
    - $E$: elements equal to $p$
    - $G$: elements larger than $p$
  - make recursive call:
    - if $k \leq |L|$: call quickselect($L, k$)
    - if $|L| < k \leq |L| + |E|$: return $p$
    - if $k > |L| + |E|$: call quickselect($G, k - (|L| + |E|)$)
  - conquer: return

Quickselect Pseudo-code

```python
def quickselect(list, k):
    if list has 1 element return it
    pivot = list[rand(0, list.size)]
    L = []
    E = []
    G = []
    for x in list:
        if x < pivot: L.append(x)
        if x == pivot: E.append(x)
        if x > pivot: G.append(x)
    if k <= L.size:
        return quickselect(L, k)
    else if k <= (L.size + E.size):
        return pivot
    else:
        return quickselect(G, k - (L.size + E.size))
```
Quick Sort vs. Quickselect

Quick Select

- **Recurs only on the part of the array that contains the $k^{th}$ smallest element**
  - If pivot index $> k \rightarrow$ recur on the left section of the array
  - If pivot index $= k \rightarrow$ FOUND the $k^{th}$ smallest element
  - If pivot index $< k \rightarrow$ recur on the right section of the array

- **Recall: Quick Sort recurs on the left and right sections of the array**
Graphs
Practice Problem
existsPath(v1, v2):
    Q = []  //Queue
    v1.visited = true
    Q.enqueue(v1)
    while Q not empty:
        cur = Q.dequeue()
        if cur == v2:
            return true
        for neighbor in cur’s adjacent nodes:
            if not neighbor.visited:
                neighbor.visited = true
                Q.enqueue(neighbor)
    return false
Different implementations of a Graph

Graphs can be represented as:

- Edge List (or set)
- Adjacency Lists (or sets)
- Adjacency Matrix
Edge List (or Set)

- Way of representing which vertices are adjacent to each other as a list of pairs
- Each element in the list is a single edge (a,b) from node a to node b
- Since order of this list doesn’t matter, we can use a single hash set instead to improve runtime

Edge List:

\[
[(1,1), (1,2), (1,5), (2,3), (2,5), (3,4), (4,5), (4,6)]
\]

Big-O Performance (Edge Set)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertices()</td>
<td>O(1)</td>
<td>Return the set of vertices</td>
</tr>
<tr>
<td>edges()</td>
<td>O(1)</td>
<td>Return the set of edges</td>
</tr>
<tr>
<td>incidentEdges(v)</td>
<td>O(</td>
<td>E</td>
</tr>
<tr>
<td>areAdjacent(v1,v2)</td>
<td>O(1)</td>
<td>Check if edge (v1,v2) exists in the set</td>
</tr>
<tr>
<td>insertVertex(v)</td>
<td>O(1)</td>
<td>Add vertex v to the vertex list</td>
</tr>
<tr>
<td>insertEdge(v1,v2)</td>
<td>O(1)</td>
<td>Add element (v1,v2) to the set</td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>O(</td>
<td>E</td>
</tr>
<tr>
<td>removeEdge(v1,v2)</td>
<td>O(1)</td>
<td>Remove edge (v1,v2)</td>
</tr>
</tbody>
</table>
Adjacency List (or Set)

**Adjacency Lists (or Sets)**

- Another way of representing which vertices are adjacent to each other
- **Each** vertex has an associated list representing the vertices it neighbors
- Since order of these lists doesn’t matter, they could be hash sets instead to make lookup faster

![Diagram of a graph with vertices 1, 2, 3, 4, 5, 6 and their adjacency lists]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1,2,5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[1,3,5]</td>
<td>[2,4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>[3,5,6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>[1,2,4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>[4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[1,2,4]</td>
<td>[4]</td>
</tr>
</tbody>
</table>

**Big-O Performance (Adjacency Set)**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertices()</td>
<td>O(1)</td>
<td>Return the set of vertices</td>
</tr>
<tr>
<td>edges()</td>
<td>O(</td>
<td>E</td>
</tr>
<tr>
<td>incidentEdges(v)</td>
<td>O(1)</td>
<td>Return v’s edge set</td>
</tr>
<tr>
<td>areAdjacent(v1,v2)</td>
<td>O(1)</td>
<td>Check if v2 is in v1’s set</td>
</tr>
<tr>
<td>insertVertex(v)</td>
<td>O(1)</td>
<td>Add vertex v to the vertex set</td>
</tr>
<tr>
<td>insertEdge(v1,v2)</td>
<td>O(1)</td>
<td>Add v1 to v2’s edge set and vice versa</td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>O(</td>
<td>V</td>
</tr>
<tr>
<td>removeEdge(v1,v2)</td>
<td>O(1)</td>
<td>Remove v1 from v2’s set and vice versa</td>
</tr>
</tbody>
</table>
Adjacency Matrices

- Another way of representing which vertices are adjacent to each other
- Matrix is \( n \times n \), where \( n \) is the number of nodes in the graph
  - true = edge
  - false = no edge
- If \( m[u][v] \) is true, then node \( u \) has an edge to node \( v \) (and if the graph is undirected, we can assume the opposite is true as well)
### Big-O Performance (Adjacency Matrix)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertices()</td>
<td>O(1)</td>
<td>Return the set of vertices</td>
</tr>
<tr>
<td>edges()</td>
<td>O(</td>
<td>V</td>
</tr>
<tr>
<td>incidentEdges(v)</td>
<td>O(</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: row/col are the same in an undirected graph.</td>
</tr>
<tr>
<td>areAdjacent(v1,v2)</td>
<td>O(1)</td>
<td>Check index (v1,v2) for a <code>true</code></td>
</tr>
<tr>
<td>insertVertex(v)</td>
<td>O(</td>
<td>V</td>
</tr>
<tr>
<td>insertEdge(v1,v2)</td>
<td>O(1)</td>
<td>Set index (v1,v2) to <code>true</code></td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>O(</td>
<td>V</td>
</tr>
<tr>
<td>removeEdge(v1,v2)</td>
<td>O(1)</td>
<td>Remove v1 from v2’s set and vice versa</td>
</tr>
</tbody>
</table>
Cycles
Given a singly linked list, return true if there is a cycle (return false otherwise)
Cycles Pseudocode

```python
isCycle(head):
    if (head == null):
        return false
    tortoise = head
    hare = head.next
    while hare hasn’t reached the end of the list:
        if tortoise == hare:
            return true
        if hare.next == null:
            return false
        hare = hare.next.next
        Tortoise = tortoise.next
    return false
```

Mini-assignment
Mini Assignment

- Connected: there exists a path from each vertex to every other vertex
  
  **NOTE:** Connected does not mean that each vertex is *directly* connected to every other vertex (that characteristic is called *completeness*).

- Acyclic: The graph has no cycles
Mini Assignment

- Unconnected: There does not exist a path from each vertex to every other vertex
- Cyclic: There exists at least one cycle in the graph
  - Recall: A cycle is a path that starts and ends at the same vertex
Master Theorem
Master Theorem: Review

**Master Theorem Equation**

\[ T(n) = a \cdot T \left( \frac{n}{b} \right) + \Theta(n^d) \]

- **a**: number of sub-problems
- **n / b**: size of each sub-problem
- **n^d**: work to prepare sub-problems & combine their solutions

**Runtime Interpretation**

- \( T(n) = \Theta(n^d) \) if \( a < b^d \)
- \( T(n) = \Theta(n^d \log(n)) \) if \( a = b^d \)
- \( T(n) = \Theta(n^{\log_b a}) \) if \( a > b^d \)

*Note: b should be subscripted*
Master Theorem: Example

Let $a = 2$, $b = 2$, $d = 1$, and $T(n) = 2T(n/2) + O(n^d)$. What is the runtime of this function?

a) $\Theta(n^d)$

a) $\Theta(n^d \log n)$

a) $\Theta(n^{\log_b a})$ *Note: $b$ should be subscripted*
Let $a = 2$, $b = 2$, $d = 1$, and $T(n) = 2T(n/2) + O(n^1)$. What is the runtime of this function?

a) $\Theta(n^d)$

a) $\Theta(n^d \log n)$

a) $\Theta(n^{\log_{ba}})$ *Note: $b$ should be subscripted*

Solution: B

$a = b^d$
so runtime is $T(n) = \Theta(n^d \log n)$