CSC148
Lecture 9

Quick Sort
Graphs

Quick Sort

- Recursive, like merge sort, but sorting is “in place”. That is, additional space is not required.
- The main idea behind quicksort is contained in the partition procedure. It works by choosing a “pivot” element and
  - finding the correct position of the pivot element in the final sorted list (this is called the “split point”)
  - moving elements less than the pivot before the split point, and other elements after the split point.

Quick Sort

- Quick sort works by partitioning the list (using the partition procedure described above), and then recursively sorting the lists before and after the split point.
- In the worst case, the split point can always be skewed to one side of the list, resulting in O(n^2) time complexity.
- On average, the time complexity of Quicksort is O(n log n)

Quick Sort

- Examples on board

Quick Sort

- Lets look at the quick sort procedure in Wing.

Graphs

- Graphs can be used to represent a number of real-world artifacts
- Intuitively, graphs consist of a number of “nodes” (vertices) connected by lines (known as “edges”.
- Edges express a relationship between the two nodes.
- Edges may be directed, in which case the relationship between the two nodes is directional.

Graphs

- Edges may be undirected, in which case the relationship between the two nodes is symmetrical.

Graphs

- A vertex has a label, just like vertices in a tree.
- A vertex can also have a value associated with the key. (Your textbook calls this the 'payload').
- Graphs containing directed edges are known as directed graphs (or 'digraphs').
- Edges may have values assigned to them, called “weights”. What this value expresses depends on the graph – for example, in a graph representing roads that connect one place to another, the weight may be the distance.

Graphs

- More formally, a Graph G is a pair (V,E), where V is a set of vertices, and E is a set of edges.
- Edges are tuples (v,w), where v and w are in the vertex set V.

Graphs

- A path in a graph is a sequence of vertices that are connected by edges
- A **simple** path is a path that contains no duplicate vertices.
- The length of a path is the number of edges in a path. The weighted path length is the sum of the weights of all edges in the path.
- The distance between two vertices is the length of the shortest path between them.

Graphs

- A cycle is a path that starts and ends at the same vertex
- A connected graph is a graph in which there is a path between any two vertices
- A complete graph is a graph that contains every possible edge.
- The degree of a vertex is the number of edges incident to a vertex

Graphs

- The following is known as the 'Handshaking Lemma': The sum of the degrees of all vertices is equal to twice the number of edges in the graph.
- A corollary to this is that the number of vertices of odd degree is even (otherwise the sum of degrees couldn't add up to an even number).
- (In any group of people, the number of people with an odd number of friends in the group is even).

Representing Graphs

- Adjacency Matrix for an unweighted graph
  - Tells you which vertices are “adjacent” (i.e., connected by an edge)
  - If entry (i,j) in the matrix is 1, then there is an edge from vertex i to vertex j. Entry (i,j) is 0 otherwise.
  - if the graph is undirected, then the adjacency matrix is symmetric (i.e, its transpose equals itself).
- Adjacency Matrix for a weighted graph
  - Entry (i,j) represents the weight of the edge from i to j. If 0 is a valid weight, another value is needed to represent the absence of an edge from i to j.

Representing Graphs

- Adjacency matrices can use up a lot of space:
  - If a graph has |V| vertices, then the adjacency matrix contains |V|^2 entries to represent all possible edges that can exist in the graph.
- There's a more efficient way of representing a graph: Adjacency list

Representing Graphs

- In an Adjacency List, we store a list of vertices, and with each vertex we store a list of adjacent vertices.

Breadth First Search (BFS)

```
enqueue start vertex into queue
while queue is not empty:
        u = dequeue vertex from queue
        visit u
        for each (u,v) in E:
                if v is not already discovered:
                        set v as discovered
                        enqueue v into queue
```

Depth First Search

```
def dfs(graph, start_vertex):
    dfs_helper(graph, start_vertex, set([]))

def dfs_helper(graph, vertex, discovered):
    add vertex to discovered
    visit vertex

    adjv = neighbours of vertex
    for vertex2 in adjvertices:
        if vertex2 is not in discovered set:
            dfs_helper(graph, vertex2, discovered)
```