**EXAM 4 STUDY GUIDE**

* **Graphs**
  + ***A Graph*** is a data structure that consists of several vertices and several edges that connect vertices.
    - **Vertex**- can contain data and may have edges coming in or going out
    - **Edge** - can be weighted (i.e. have a value) and connects two vertices (which are considered adjacent)
  + **Graph Terminology:**
    - **Directed** - The edge pair (vi , vj) denotes a single direction from vi to vj , but not the other way around.
    - **Undirected** - The edge pair (vi , vj) denotes that vi and vj can go back and forth. Equivalent to two directed edges.
    - **Cyclic** - There exists a cycle in the graph. A cycle is defined as a path from a node back to itself, where no edges are used twice
    - **Acyclic** - There are no cycles in the graph.
    - **A complete graph -**  is one such that every vertext is adjacent to every other vertex.
      * **Adjacent vertices** - are those that are directly connected via an edge (i.e. have a path of length 1 to the other node).
    - We usually denote the sizes of the vertex and edge sets with V and E respectively, and their sizes as |V| and |E| which is known as **set** **cardinality**
  + **Graph Implementation:**
    - A graph consists of a set of vertices. We can store these in a variety of ways (though we can think about it as an array-list). The graph also needs a way to reference the edges and link them to the vertices. There are two ways this is typically done:
      * **Adjacency List** - where each Vertex object houses a list of edges
        + We can implement the graph data structure as a array-based list of vertices. Each Vertex will be another data structure that contains a list of edges, with indices to the connect vertex locations in the vertex list.
        + **Pros/Cons:**

Memory Efficient

Easy/Cheap to add edges

Could take a long time to iterate through the graph

Typically used when |E| <<< |V|^2 , a.k.a. **sparse graph**

* + - * **Adjacency Matrix** - where the graph data structure contains a 2D Array with vertices represented on the rows and columns
        + We can implement the graph data structure as a array-based list of vertices as before, but the edges are also stored on the graph, this time there will be a 2D array created of size |V|x|V|, where it is initialized to all zeros (no edges) and everytime there is an edge between vertices a 1 will be store to indicate there is an edge from the row vertex to the column vertex.
        + **Pros/Cons:**

Very fast look up and iteration

Costly to add edges (grow matrix)

Takes up a lot of memory especially for graphs containing a lot of vertices.

Typically used when |E| ≈ |V|^2 , a.k.a. **dense graph**

* + **Graphing Algorithms:**
    - **Depth First Search**
      * **Problem**: Does there exist a path from vertex vi to vertex vj
      * **Solution** **1**: Begin looking at an adjacent vertex from vi , then look at an adjacent vertex to that one and so on as far as you can from the original vertex, backtracking when you have reached a visited vertex or there are no more edges
      * **Depth First Search** - is when vertices of distance d are visited then a vertex d + 1 distance away, and so on until cannot go further, back up and try a different path.
        + By looking at deeper and deeper we are wanting to explore the latest vertex examined. So the last one in should be the first one out. **LIFO = use of stacks!**
  + **Breadth First Search:**
    - **Problem:** Does there exist a path from vertex vi to vertex vj.
    - **Solution 2**: Begin looking at all adjacent vertices from vi , then look at all vertices that have path size 2, then path size 3, then....
    - **Breadth First Search** - is when all vertices of distance d are visited before looking at any vertices > d distance away. We visit the vertices we discovered first before moving on sounds like a job for **FIFO = Queues!**
    - **Minimum Spanning Tree** - A spanning tree Is a subtree such that the edges of the tree such that every vertex is contained within the tree. Minimum refers minimizing the weight of edges when added (So far we have assumed all weights to be equal...1 in all previous examples).
  + **Prim’s Algorithm:**
    - 1. Initialize a tree with a single vertex, chosen arbitrarily from the graph
    - 2. Grow the tree by one edge; of the possible edges that connect the tree to the vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree
    - 3. Repeat until all vertices have been added
      * This is a **Greedy algorithm** - in that it only chooses the best possible solution given limited knowledge (proof that this works here is beyond this course - greedy algorithms do not always work)
        + **NOTE**: Only guaranteed to work with undirected edges...to be seen...can work with directed as shown in class, but must meet specific requirements.

**Prim’s Algorithm**: Does not work on directed graphs. It will sometimes, but many times it will fail because a local minimum does not guarenteed global minimum

* + - * **Prim’s Algorithm**: Initialize a tree with a single vertex, chosen arbitarily from the graph
      * **Prim’s Algorithm:** Examine the costs of the adjecent edges leading to not yet visited nodes
  + **Dijkstra’s Algorithm:**
    - **Shortest Path** - Is the shortest path between two vertices. Dijkstra’s algorithm is one notable algorithms for computing the shortest path between vertex vi and vj :
      * **1)** Assign an infinite distance to all other vertices from the source.
      * **2)** . Mark the source as current. Mark all other vertices unvisited. Create a set of all the unvisited vertices called the unvisited set (Priority Queue).
      * **3)** For the current vertex, consider all of its unvisited neighbors: •
        + Calculate their tentative distances.
        + Compare the newly calculated tentative distance to the current assigned value and assign the smaller one (May need to reorganize PQ).
      * **4)** Mark current vertex as visited and remove it from the unvisited.
      * **5)** If the destination vertex has been marked visited or the smallest tentative distance among the vertices in the unvisited set is infinity - Done, found it or cannot find path respectively
  + **A\* Search Algorithm:**
    - **A\*** is a generalization of Dijkstra’s algorithm in which each path has a cost like before, but also a **heuristic cost** associated with getting to the source (an estimate, which could be as simple as assumed number of edges)
* **Dynamic Programming:**
  + **Top-Down** - Looking at the big picture and breaking it down into smaller and smaller problems. Recursion and Memoization.
  + **Bottom-Up** - Looking at the base/simple parts and build up all possible solutions. Iterative and Dynamic Programming.
  + **Dynamic Programming** - is a type of algorithm that finds a solution to sub-problems and uses those solutions to generate the solution to the next problem.
* **Greedy Algorithms:**
  + **Greedy algorithm** - is a type of algorithm that finds a solution to an optimization problem. The heuristic that a greedy algorithms uses is choosing a local optimal path/choice/solution.
* **Floyd – Warshall Algorithm:**
  + Given that we maintain the shortest path from vi → vj everytime a cell gets updated we no longer have to account for previous calculations (do not recompute double links, triple, ... therefore we can eliminate the redundancies and compute all-pairs shortest path into O(n 3 ) running time
* **Longest Common Subsequence:**
  + The longest common subsequence (LCSS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences). It differs from problems of finding common substrings: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences. Used frequently in **Bioinformatics.**
* **Sorting**
  + **Selection Sort:**
    - **Selection sort** - can be thought of as looking for the minimal element in the list and copying it over to a new list repeating with the remaining elements. This could be memory inefficient as it would maintain two lists. Instead we can swap the smallest value with what remains.
      * Repeat searching for minimal location, swapping with back of sorted portion
  + **Bubble Sort:**
    - **Bubble sort** - can be thought of as bubbling up the smallest value in the array. We will begin at the end of the array and swap as we go to ensure the smaller value comes first.
      * Continue swapping to the front to build a sorted part, each minimum will be swapped the whole way when found (other swaps may occur, but it is only the minimum in the unsorted section that really matters.
        + If the data is almost sorted, our swaps may take care and we can see if no swaps were needed the data has been sorted and we can end early!
  + **Shell Sort:**
    - **Shell sort** - is an extension of Bubble sort in that it uses a set of decreasing gaps (Ex: 31, 15, 7, 3, 1) to use for comparisions/swaps in order to get data on roughly the correct size before settling on pure bubble sort, because as we have seen the closer the data is to sorted the better bubble sort will perform.
      * **Bubble v2 can be better than Shell** when the data is perfectly sorted (as shell cannot stop until it gets down to 1 space shifting
  + **Insertion Sort:**
    - **Insertion sort** - works like adding elements to a list shifting elements as needed just like when we created a sorted array-based list.
      * Continue swapping each new element to be added into sorted set until the previous element is less than
  + **Merge Sort:**
    - **Merge Sort** - is a type of algorithm that is known as divide-and-conquer – essentially top-down strategy that breaks apart the problem into smaller problems to solve (hmmm...sounds like recursion). Merge sort splits the the array in half and sorts each half, then the results are merged together
  + **Quick Sort:**
    - **Quick Sort** - is also a divide-and-conquer algorithm, but unlike merge sort that splits the data in half on size, we will split on a value we think (estimate to be in the middle) known as the pivot value, placing values smaller than the pivot to the left and larger values on the right.
      * Continue to process each split that has not gone down to single/two entry sections.
    - **Heap Sort:**
      * Heap Sort uses the properties of Heaps to reorder the data by looking at the largest elements one at a time. First we must alter the data to be a heap, starting bottom-up with the subtree.
    - **Stability:**
      * **A Stable sort** - is a sort that maintains the order of equal elements within for example sorting {B1, A1, A2, B2} given A1 == A2 and B1 == B2
      * A stable **sort would result in {A1, A2, B1, B2}, whereas an unstable sort would** be something like {A1, A2, B2, B1}. Sorting techniques thus far:
        + **Selection Sort** - **Stable** if finding min use < not <=
        + **Bubble Sort** **- Stable** if swap only when < not <=
        + **Shell Sort** - It is **NOT stable** as larger gaps may change order
        + **Insertion Sort** - Stable if when inserting you insert when previous element is <=
        + **Merge Sort** - **Stable** if adding left to temporary before right if they are equal
        + **Quick Sort** - By definition **NOT Stable**
        + **Heap Sort** - By definition **NOT Stable**
* **Hashing and Linear Sorting:**
  + **Hashing**: is a technique that orders and accesses elements iteratively in constant time, by using a hash code as a index/key. This hash code is computed via a hash function.
    - **Hash function**: is a function/method that generates a numeric key to reference an object. In Java, the Object class has a hashCode() method built-in that uses the object’s memory address. This is what is print out by default toString() method. We can override this behavior to produce any hash code we want:
    - **Hash table**: is essentially an array where the hash code is related to the object. Therefore we can access our array very simply by calculating the hash code and looking up the object
    - When two different objects are hashed to the same value, a **collision** occurs. Of course, we would probably want to choose a better hash function, but we may be stuck with one (e.g. stuck with the java default). One way to handle this is starting at the hased location look for the next available spot. This is known as **linear probing.**
      * When we want to check for an object or remove it with **linear probing** (like adding) it is no longer perfectly constant. Our methods will now iteratively look for the object/place to add.
      * Rather than using **Linear probing** to avoid conflict we can rehash in other ways:
        + **Quadratic Probing** - adjusts the index by plus/minus squares of a shift
        + **Random Probing** - adjusts the index from a collision by a seeded (so we can access when same object added) random amount.
    - When we have a lot of collisions in one area of the table we say that there is **clustering** around a certain hash key/code/location
  + Rather than storing single values in the array and probing along the array (which can impact the spots we place the collided data), we can:
    - represent the table as a 2D array with multiple locations (columns) for each hash code (row/**bucket**). If a single row becomes full we can still rehash using some probing scheme.
    - have linked lists inside those array locations that contain each object with the specified hash code. **(chains)**
  + **Bucket Sort:**
    - **Bucket sort** - can be thought of like a hash function. Given the set of data we create a uniform set of buckets for our data (for example with text we could create buckets of starting letters...but essentially can be thought of in the same way as creating a hash table, and then sorting each bucket).
  + **Radix Sort:**
    - **Radix sort** - is similar to bucket sort, in that we will be placing elements into buckets (specifically buckets 0-9), as radix sort is only applicable to integer values. We will be looking at each digit of the number (can be most significant digit or least significant digit[LSD]) and sorting according to each digit of the numbers
  + **M-Way Quick Sort:**
    - **M-Way Quick Sort** - is a variant of the quick sort technique where M − 1 pivots are chosen in order to segment the data into M different segments, and like before a divide-and-conquer technique, where each segment has M-Way Quick sort performed on it. The data is iterated over and placed in the right location in the array
* **Java Garbage Collection**
  + **The Young Generation** is where all new objects are allocated and aged. When the young generation fills up, this causes a **minor garbage collection.**
    - **Minor garbage collections** are always **Stop the World events** (Halt application threads).
  + **The Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a **major garbage collection.**
    - **Major garbage collection** are also **Stop the World events**.
    - **Major collection** is much slower because it involves all live objects (so should be minimized).
  + **The Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application.
    - **The permanent generation** is populated by the JVM at runtime based on classes in use by the application
    - Java SE library classes and methods may be stored here
  + New objects are allocated to the **eden space**. **Survivor spaces** start out empty.
  + When the **eden space** fills up, **minor garbage collection** occurs.
  + **Referenced objects** are moved to the **first survivor space**. **Unreferenced** objects are deleted as **eden** is cleared.