Chapter 2:
- Loop Invariants
  - Initialization: Prove its true at the start
  - Maintenance: Prove its true at the intermediate steps
  - Termination: Prove its true at the end
- Worst Case provides an upper bound on runtime
- Recurrence: T(n)= aT(n/b)+D(n)+C(n)
- a=#calls, b=split size
- height often equals log_b_n + 1
- Divide, Conquer, Combine
- Recurrence Tree runtime is the work at each level
  - Multiplied by the amount of levels
  - Plus the amount of work done on a trivial size

Chapter 4: Runtime of Divide and Conquer Algorithms
- Substitution Method
  - Guess the form of the solution
  - Use mathematical induction to find the constants
    - Show the solution works
- Recurrence Trees for Guessing
  - SUM all the levels
  - Replace sum with infinitely increasing for lower bound
  - Infinitely decresing sum for upper bound
  - Sum((a/b)^i) to infinity = 1/(1-(a/b))
- Master Method
  - ONLY if T(n)=aT(n//b)+f(n)
  - Compare $n^{\log\_b\_a}$ *to* $f(n)$
  - If the former is larger, case 1, latter case 3
  - If they are the same, case 2

Chapter 5: Indicator Random Variables
- Find a way to make it so values are only 0 and 1
  - Expected number of occurences is the sum
    - Of attempts on the probability n*Pr

Chapter 6: Heaps

Chapter 7: Quicksort

Chapter 8: Comparison Sorting

Chapter 9: Randomized Select

Chapter 15: Dynamic Programming
- Optimal Substructure
- Optimization
- Overlapping Subproblems
- Recursive Solutions

Graph Algorithms

Effects on Runtime: Sparsity of graph and adjacency representation.

Keep in mind that graphs are models of elements in the world that have a relationship or connection to each other.

Ch. 25. All Pairs Shortest Path

Given a graph, can you determine the shortest path between every pair of vertices? These algorithms use a dynamic programming approach in which an entire matrix is saved at each iteration, which represents the solution to subproblems of the global problem. (Compare this to the previous dynamic programming algorithms in which a single entry in a matrix was calculated at each iteration.) These algorithms can be more efficient than a repeated application of the Single Source Shortest Paths algorithms of Ch. 24.

Algorithms:

Faster-All-Pairs: Each iteration increases the number of edges allowed in a path from u to v, Floyd-Warshall: Each iteration increases the set of vertices that can be on the path from u to v.

Ch. 24 Single Source Shortest Path

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Ch. 2 and 4 Analyzing Runtime

These chapters are the foundation of algorithm analysis (which is at the core of computer science). You have applied the techniques of these chapters countless times throughout the semester. I don't think you will need to *study* these chapters, as they are likely second-nature to you now, but just in case, you might want to look at how you define a recurrence relation for a recursive definition (like MergeSort or QuickSort) in case you have to write one on the exam and analyze its runtime.

Ch. 6 Heaps

These data structures can facilitate efficient sorting but are most useful for maintaining priority queues. Extracting the minimum element from a list is extremely useful in many of the algorithms seen later in the textbook. The runtime of heaps is concentrated on the height of the "tree" (although it is implemented as an array), which has a bound of lg n.

As with all dynamic sets, key functionality includes insertion and deletion, which is implemented efficiently in heaps, however they do not facilitate efficient search.

Given a graph, can you determine the shortest path from a given source vertex s to all other vertices? These algorithms are distinguished both by their approach and the graphs on which they operate correctly. However, both approaches use RELAX and their correctness is proven with a set of lemma's of various properties of shortest paths.
Algorithms:
Bellman-Ford: Path-relaxation property is key for correctness. Handles negative weights.
Dijkstra: Greedy algorithm. Not correct for negative weights. Runtime depends on data structure.
Understand the general functionality of these algorithms and how to calculate the runtime. You will not need to prove any of the lemmas or the correctness of the algorithms.

Ch. 23 Minimum Spanning Tree

Given a connected, undirected graph, can you find a set of edges that maintain connectivity and have the minimal weight over all such sets of edges? Correctness of both algorithms rely on the loop invariant: "A is set of edges that is a subset of some minimum spanning tree." The concept of a cut is also important in understanding correctness.
Algorithms:
MST-KRUSKAL: At each iteration, add the minimum weight edge not in the set A that does not create a cycle in A. Relies on the implementation of disjoint sets, as covered in Ch. 21.
MST-PRIM: At each iteration, add the minimum weight edge that crosses the cut {S, V-S} such that S contains all vertices included in the edges of A.
Understand the functionality of these algorithms and how to calculate the runtime. You will not need to prove the correctness of these algorithms.

Ch. 22 Elementary Graph Functions

Algorithms:
Breadth-First-Search: Start from a given source and build a tree that represents the shortest path from the source to any vertex (given unit edge weight).
Depth-First-Search: Start from any vertex and build a forest that represents the connectedness of vertices within the graph.
Understand the functionality of these algorithms. No proofs required.

Paradigms: Ch. 15 and 16 Dynamic Programming and Greedy Choice

Key Concepts: Optimal substructure, greedy choice property, optimization, overlapping subproblems, recursive solution
Greedy: the locally greedy choice will lead to a globally optimal solution.
Dynamic: a brute force approach that can take advantage of overlapping subproblems by making memo's of calculated costs of subproblems and use those in future calculations.
Can you recognize a greedy choice problem or a dynamic programming problem? Can you follow the proofs in the textbook for greedy choice and optimal substructure for the specific problems presented (i.e. matrix multiplication, LCS, activity selection, and huffman codes)? You will not need to construct a proof, but you might be asked to fill in a piece or to start the proof.

Ch. 12 : Binary Search Trees

Data structure for a dynamic dataset that facilitates efficient generation of a sorted list, yet has efficient insert and delete. HOWEVER, runtime is highly dependent on the height of the tree, and there is no guarantee of a balanced tree to achieve the ideal O(lgn) runtime. Note that Red-Black trees are BST's with additional properties that help maintain balance in the tree.
Binary Search Tree Property: Left subtree is all <=, Right subtree is all >=
Algorithms: Tree-Minimum, Tree-Maximum, Tree-Successor, Tree-Insert, Tree-Delete

Ch. 11 Hash Tables (i.e. dictionaries)

Data structure that functions like an array whose *index* values can be of any type. It makes for highly efficient retrieval of a specific element, but does not maintain a sorted order of elements.
Vocabulary: Direct Addressing, Chaining (to deal with collisions), Hash Function, Load Factor, Simple Uniform Hashing/Distribution.
Algorithms: Chained-Hash-Insert, Chained-Hash-Search (O(1+alpha)), Chained-Hash-Delete
Hash Functions: Division Method, Multiplication Method.

Sorting: Ch. 2, 6, 7, 8

Less emphasis will be placed on these earlier chapters, but keep in mind that the ability to sort is essential to many of the graph algorithms, so don't completely forget about these.
Loop invariants are key to proving the correctness of most of these algorithms.
Runtime analysis is an important part of understanding under what conditions each sorting approach is most appropriate. Think about randomization and how that plays an important role in controlling the expected runtime of these algorithms.
Algorithms: Comparison Sorts (Omega(nlgn)): Insertion-Sort, Merge-Sort, Quicksort. Linear-Time Sorts: Counting-Sort, Bucket-Sort.

Floyd-Warshall Finds the shortest path between every pair of vertices in a graph
- Supports Negative edge-weights, but not negative-weight cycles
- $O(n^3)$ (n^2 boxes in a table, looked at n times)
- Works by increasing the number of hops allowed by 1 with each iteration
- Create a table, Distance from point to itself = 0
- Distance from point to directly connect point = weight of that edge
- Distance from point to not connect point = infinity
- Create another table of predecessor vertices
- In successive iterations, check if making a stop at a specific vertex improves the distance of every path in the table
- A change in value on the diagonal indicates a negative weight cycle and should halt

Bellman-Ford Finds the shortest path from a source to any vertex
- Works on negative edge graphs, but not with negative weight cycle
- An array of distances, and an array of predecessor vertices
- D[v] = current shortest path length, W[u,v] = weight of edge between two vertices
- Source inits to 0, every other vertex inits to infinity
- If path is shorter than the current length, update d[v] and pi[v]
- Do this for every vertex, then do all of it |V|-1 times, working on one less vertex from the source every time
- Do another iteration, after everything has been done, if an edge still updates there is a negative weight cycle
- O(VE+V+E)=O(VE)

Djikstra's Finds the shortest path from a source to any vertex
- Does not work accurately with negative weight edges
- Initialize Source to 0 and every other vertex to infinity
- Relax every vertex directly connected to source, record the length and the predecessor
- Choose the minimum of the vertices and that vertex is the new source, and relax from that vertex again
- Repeat until every vertex has been selected as the source
- Uses a queue to keep track of selected vertices (a vertex gets popped after being used, and can no longer be updated)
- O(E log V)

Depth First Search. Calculates the time to visit every node in a graph (shortest path in an unweighted graph)
- Used for detecting cycles, finding the number of connected components and solving mazes
- Implemented with a Stack. Can start with any source, just end wit ha different ordering
- Visit a node and push it to a stack, continue from that node until can't continue further
- Pop off stack until the vertex can continue to an unvisited node, Repeat until every node is visited
- Result is the order in which nodes were pushed
- Starting time starts at 1 at source, increase count by 1 when a node is pushed, and when it is popped, every node should
- Have 2 times, finishing time is the finishing time on source after EVERY node has been visited

Breadth First Search Finds the shortest path in an unweighted graph, and the connectedness and bipartiteness of a graph
- Traverses graph by layer, visits all of a node's adjacent nodes before visiting those node's adjacent nodes
- Enqueue the node's adjacent nodes, dequeue when we look at that node's adjacent nodes,
- After the queue is empty, every node that can be looked at, has been looked at, and the algorithm finishes
- The result is the order of enqueued nodes
- Every node's path length is the layer starting at 0 at the source

Minimum Spanning Tree
- Spanning Tree: A graph that connects all of the vertices once
- Minimum Spanning Tree: Connects all vertices and minimizes weights
- LI: "A is a set of edges that is a subset of some minimum spanning tree"
- Tree = No cycles

Prim's Finds a minimum spanning tree
- Similar to Djikstra's but records path weight rather than length
- Select node set with minimum distance for every node. Every pair can only be connected once
- Greedy approach, always take the minimum weighted edges

Kruskal's Finds a minimum spanning tree
- Greedy Approach, O(E log V)
- Adds the cheapest edge that connects two trees of the forest
- Pairs are ordered by weight, Start with the smallest weight edge

Go through, choosing the minimum weights that don't form a cycle
If a pair is after that forms a cycle, it won't be added since its length won't be shorter
Stops when every vertex is in the tree

Topological Sort: Depth First Search, placing each "finished" node in a linked list, leaving with a sorted list, Theta(V+E)

Strongly Connected Components: Basically groups of nodes that form cycles
Computer finishing times by DFS, sort them in decreasing order
Compute DFS of the transpose in the order computed above
Each tree in the final forest is its own SCC.

Graph Cuts with respect to the set

Hash Table
Load Factor = Elements / Slots = alpha
Division Method: h(k) = k mod m. A prime not too close to an exact power of 2 is often a good m
Multiplication Method: 0<A<1. H(k) = floor(m * (kA mod 1)). A power of 2 is often a good m
kA mod 1 means the fractional part of kA

Binary Search Tree: Organizes data for quick search and modify
BST Property: For each node Left Subtree <= node, Right >= node
Data Structure with Search, Insert, Remove O(log n) when balanced, Worst Case: O(n)
Each Subtree is a BST
Balanced: |Depth of left - Depth of Right subtrees| <= 1

Greedy Algorithms
Greedy Choice Property Proof
for the existence for some optimal solution that contains the greedy choice
Oracle gives you optimal solution S, Consider if S does not contain greedy choice
Swap some element(s) in S with the greedy choice
2 Arguments you have to prove with S' is a new solution with greedy choice
1. Prove that S' is still a solution (follows all rules of the problem)
2. S' is an optimal solution

Dynamic Programming

Works the same no matter the problem

1. "Oracle" gives you the best first choice … creates subproblem(s)
2. Claim: The global solution = solution(s) to resulting subproblem(s)
3. Contradicting Assumption: global solution != solution(s) to subproblem(s)
4. Replace subproblem solution with optimal
   a. Show this results in a **better** global solution
5. Therefore, the assumption is false