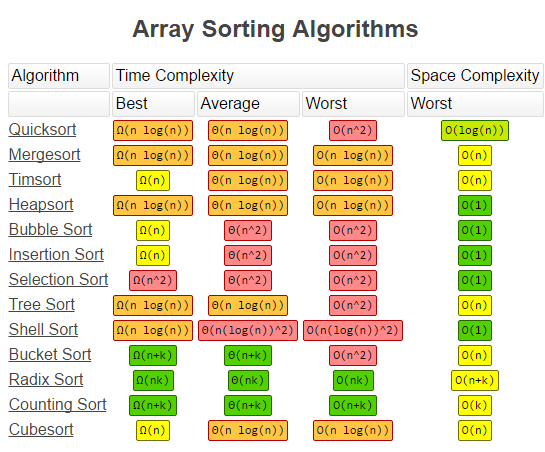
Chapter 2:



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

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

**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 many times analyzing the various algorithms of the chapters covered up to this point.

**Sorting: Ch. 2, 6, 7, 8**

Loop invariants are key to proving the correctness of most of these algorithms. Can you define a loop invariant, then write out the inductive proof? Try this on the assignment question about adding 2 binary numbers.

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. Can  you write out a recurrence relation for a divide-and-conquer algorithm, then use it to define the runtime?

Algorithms: Comparison Sorts (Omega(nlgn)): Insertion-Sort, Merge-Sort, Quicksort. Linear-Time Sorts: Counting-Sort, Bucket-Sort.

**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.

**Paradigms: Ch. 15 Dynamic Programming**

Key Concepts: Optimal substructure, optimization, overlapping subproblems, recursive 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 construct a recursive solution of the optimal cost/value of a given problem, then construct the algorithm from that?

Can you follow the proofs in the textbook for optimal substructure for the specific problems presented (i.e. matrix multiplication and LCS )? You will not need to construct a proof, but you might be asked to fill in a piece or to start the proof.