CLRS SUMMARY

1. Role of algos

- skip

2. Getting started

- read the whole chapter. it sets notation for the rest of the book.
- don't worry too much about correctness and loop invariants but try to understand. read the complexity analysis for insertion sort. read section 2.3.1 carefully.
- the complexity analysis for merge sort is the template for how almost every other complexity analysis for a recursive function is analyzed.
- must do problems: 2.1-2, 2.1-4, 2.2-2 (and write the algorithm), 2.3-2, 2.3-4 (and write the algorithm), 2.3-5, 2.3-7 (hint: use binary search). do as many of the others that you think you couldn't solve on the spot if someone gave them to you.

3. Growth of functions

- understand the differences between all of the complexity notations, $\Omega, \Theta, O, \omega, o$
- skim the stuff on functions. you really should know it given that you're an adult but it won't kill you (other than that you won't understand some of the complexity analyses later).

4. Divide and conquer

- read about divide and conquer. skim the recurrence methods.
- read the maximum subarray problem closely.
- skim matrix multiplication. look up exponentiation by squaring on wikipedia and read this page https://www.nayuki.io/page/fast-fibonacci-algorithms. this is exactly the kind of thing you'd get asked on an interview.
- skip 4.3, 4.4, 4.5, 4.6
- must do problems: 4.1-1 (if you code it up you'll see), 4.1-2, 4.1-5 (hint: this is called kadane's algorithm. don't look it up until you think about it at least for a day), 4-4 (hint for part a: write out the right hand side multiplying the zs in), 4-5, 4-6

5. Probabilistic analysis

- read 5-1
- skip 5-2
- read 5-3 skim the proofs
- read 5.4.4
- must do problems: 5.1-1, 5.1-2, 5.1-3, 5.3-7 (this is called reservoir sampling [essentially]. look it up and study the priority queue variant), look at 5-1 and 5-2. try to do them.

6. heapsort

- read everything. understand the complexity analyses here - they're basically like merge sort but will come up again for tree algorithms (because a heap is basically a binary tree).
- must do problems: 6.1-1 - 6.1-7, 6.2-2, 6.3-3, 6.5-3, 6.5-6,6.5-7,6.5-8,6.5-9 (hint: the smallest elements of each of the sorted lists should always be in "direct competition"), 6-2a,6-2b, 6-3 (hint for part f: at which corner of the matrix can you be sure of things?)

7. Quicksort

- very important. most often used sort in practice.
- look up the implementation of partition on wiki instead though. it's not different, just clearer.
- read the complexity analysis in 7.2. it's important to know why/how quicksort can fail.
- read 7.3
- skim 7.4
- must do problems: 7.1-4, 7.4-3 (because you're an adult), 7-4, 7-5 (important), 7-6

8. Sorting in linear time

- skim 8-1. it's important theory and you should know about the result but no one will ask you to reproduce it
- read 8.2, especially the last paragraph (stability is the reason the last loop goes in reverse - think about why) and 8.3
- figure out how to implement counting sort so that the last loop goes in the forward direction but the sort is still stable
- figure out how to extend counting sort so that it sorts all integers (not just positive numbers).
- implement radix sort
- must do problems: 8.2-3, 8.2-4,8.4-4,8.4-5,8-2e (hint: you should have already figured out how to do this if you implemented radix sort),8-3,8-4ac,8-5a-e

9. Medians and Order Statistics

- implement simultaneous max and min
- implement randomized select
- must do problems: 9.1-1 (and code it up), 9.2-3,9.3-5,9.3-6 (and code it up if you really want some exercise),9.3-7,9.3-8 (this one is also a tough implementation problem),9.3-9,9-2

10. Elementary data structures

- implement a linked list in python with insert, search, delete, and a constructor that takes an array as initializer (harder than it seems: http://nedbatchelder.com/text/names1.html).
- implement a binary tree (not as simple as it seems because of above)
- skip 10-3 what a dumb section
- must do problems: 10.1-2, 10.1-5 (hint: do 10.1-2 first), 10.1-6 (don't worry about time analysis - you need ammortized complexity for this), 10.1-7 (same as 10.1-6), 10.2-6, 10.2-8, 10.4-2, 10.4-3

11. Hash Tables

- implement hashing with chaining using the multiplication hash function with knuth's constant. if you're interested in the theoretical properties of knuth's constant http://cstheory.stackexchange.com/questions/9639/how-did-knuth-derive-a
- implement open address hashing if you're feeling masochistic.
- study the proof of the average number of collisions. it's important to understand why hash tables are efficient but not magical. note that the number of collisions is a function of the load factor α, so to keep the expected number of collisions low you need to keep α bounded. this is done by expanding/contracting the table. this is discussed in ch 17 section 4. i suggest you read 11.1,11.2,11.4 and then all of ch 17 (which you have to in order to understand 17.4) and then come back and think about the analyses in this chapter.
- skip 11.3
- read 11.4 closely but don't obsess.
- must do problems: 11.1-2, 11.1-3, 11.1-4 (hint: check to make sure key1->index->(key2,value) key1==key2), 11.2-5 (hint: pigeonhole principle), 11.2-6 (hint: keyword being random. the answer is the obvious thing, the trick is proving it's correct), 11.3-1,

12. BSTs

- implement binary search tree. implement all of the methods in 12.2 and 12.3
- read the explanations before trying to implement the code and absolutely don't just blindly implement the code. understand the cases
- must do problems: 12.1-2, 12.1-3,12.1-4 (hint: use a prev pointer), 12.2-2, 12.2-3, 12.3-3 (this is the point of balanced binary trees), 12.3-4 (show a counter-example), 12.3-6 (just think about this one), 12-2 and implement it (hint: look at the picture closely and figure out how to traverse that tree to get an ordered representation of the lightly shaded nodes) .

13. Red-Black Trees

- read everything. balanced binary search trees are important and red-black trees are the most basic kind. read the included pdf that i wrote.
- implement rotate-left and rotate-right in 13.2
- implement everything. read the explanation before trying to implement the code and absolutely don't just blindly implement the code. understand the cases
- must do problems: 13.1-5, 13.1-6, 13.2-5

14. Augmented Data structures

- implement order-statistic trees (good practice and you have to understand them to do some of the problems).
- skip 14.2
- read 14.3 closely. interval trees are important. implement. you need to implement insert and delete in a way that you maintain the added property and both require modifying left-rotate and right-rotate.
- must do problems: 14.1-3, 14.1-4, 14.1-5 (hint: what is the rank of this ith element?), 14.1-6, 14.1-7 (hint: use two arrays), 14.1-8 (hint: use 14.1-7), 14.3-3, 14.3-4 (hint: you should go down the right branch but only in certain instances), 14.3-6 (hint: use successor

and predecessor), 14.3-7, 14-1a (hint: by contradiction) 14-1b, 14-2a (hint: do the obvious thing and your complexity will come out to be $O(mn)$ but since m is a constant it's $O(n)$, 14-2b (hint: use an order statistic tree).

15. Dynamic Programming

- this is simultaneously the most important and worst written chapter of the book. read https://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf instead but read it twice and do all the problems. this is like one of the most tested things in interviews.

16. Greedy Algorithms

- This is a pretty good chapter. greedy algorithms are obvious but there are a good number of dynamic programming problems in the exercise in situations when the greedy solution isn't optimal
- read 16.1, 16.2, 16.3, skip 16.4, 16.5
- implement huffman trees/coding
- must do problems: 16.1-1, 16.1-4, 16.1-5, 16.2-2, 16.2-3,16.2-4,16.2-5, 16.2-6,16.2-7, 16.3-6 (hint: which traversals uniquely define a tree?), 16.3-7, 16-1, 16-2

17. Amortized Analysis

- important but not too important. read all the sections but don't stress. 17.4 is important though because it's how hash tables really have O(1) operations.
- must do problems: 17.2-3, 17.3-6, 17.4-1,17.4-2,17.4-3, 17-2,

18. B-Trees

- skip unless you have lots of time

19. Fibonacci Heaps

- skip unless you have lots of time

20. van Emde Boas Trees

- skip unless you have lots of time

21. Data structures for disjoint sets

- pretty good chapter. probably a useful data structure in practice too. read 21.1, 21.2, 21.3, skim 21.4 but the important parts are the results lemmas 21.11, 21.13, thm 21.14.
- must do problems: 21.1-3, 21.2-5, 21.3-2, 21.3-4, 21-1b,c,21-2a,b,c,d,21-3abc (these say prove but basically you should figure out how the algorithm works). there are notes in the ch21 folder that should help. extra credit: what's the naive way to do this?

22. Elementary Graph Algorithms

- read everything and do all the problems
- hints:
  - 22.1-6: can a universal sink have two 1s in its row
  - 22.2-7 use bfs somehow (since this question appears in the section on bfs duh)
  - 22.2-8 bfs gives you distances...
  - 22.2-9 if you didn't do tarjan's algorithm in chapter 21 do it now
  - 22.3-13 the obvious answer works you just have to prove it has the right time bounds
  - 22.4-2 dp
  - 22.4-3 again the obvious answer works , you just have to argue that the time bounds are right
  - 22-4 go in order of increasing L(u)

23. Minimum Spanning Trees

- read everything and do all the problems (ie attempt the theory problems - they're not too hard and it'll give you facility with the definitions).
- for dijkstra's algorithm you're going to need a min heap with decrease key. if you didn't implement it in chapter whatever with enough generaliy then use pythong HeapDict package .
- hints:
  - 23.1-11 what happens when you add the edge to the current mst?
  - 23.2-2 keep a list of min edge to every vertex and just take mins
  - 23.2-3,4 consider edges "in order"
  - 23.2-7 nm how fast. how would you do it? think about 23.1-11
  - 23-1a unique weights
  - 23-1b almost by definition
  - 23-1c a minimum spanning tree is a tree. that means what?
  - 23-1d sort the edges by weight. also be careful (you need to minimize the amount of weight added to the tree).

24. Sinlge-source shortest paths

- read the last section first. think moderately hard about the proofs. it'll help make the algos really make sense.
- really important so you should do a lot of the exercises, even the theoretical ones
- hints:
  - 24.1-3 what's a clear indicator you're done?
  - 24.1-5 create a universal source and run bellman-ford from that source (but be careful about their being only positive weight edges)
  - 24.1-6 there are two different ways to do this (one of them more clever than the other but more expensive)
  - 24.2-4 dp is really easy here
  - 24.3-4 there are a couple of properties to check but the most important one is whether any vertices can be relaxed
  - 24.3-6 use log
  - 24.3-7 creates more vertices

– 24.3-8 use buckets
– 24.3-9 what's the range of other edge weights given the last edge relax has weight x?
– 24-1b topological sort gives you what?
– 24-2 this one laid all nice for you so just "follow your nose"
– 24-3 take log. also the key question is how to find longest or shortest cycle? if (u, v) is an edge in a negative weight cycle you can use bellman-ford to find the shortest cycle containing (u, v) by using v as the source.
– 24-6 i'm proud of this one. think about path-relaxation property from section 24.5. then think about what you would do if the shortest paths were only monotonic.f