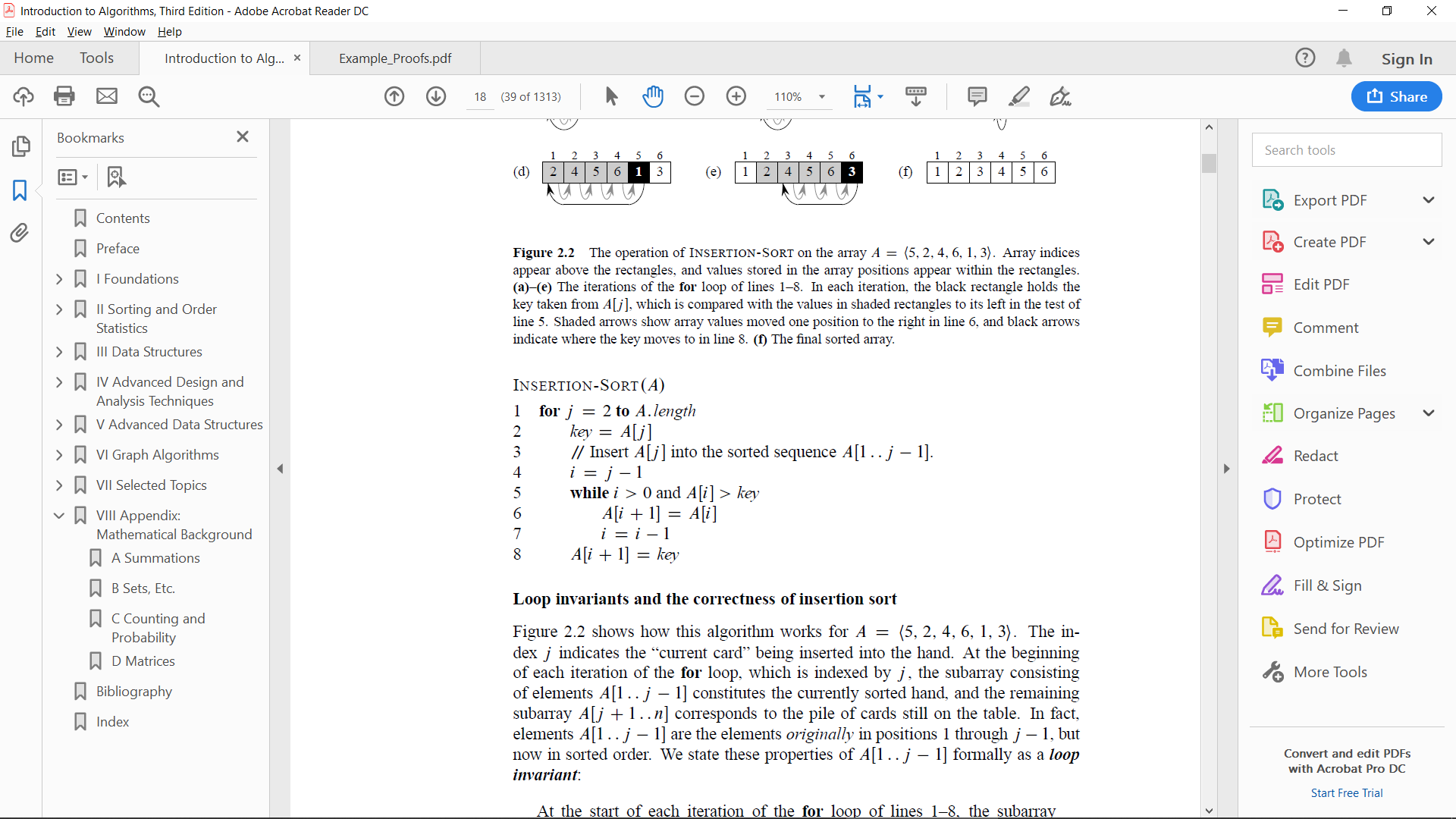
Insertion Sort:

* Proof by loop invariant
* comparison-based
* greedy

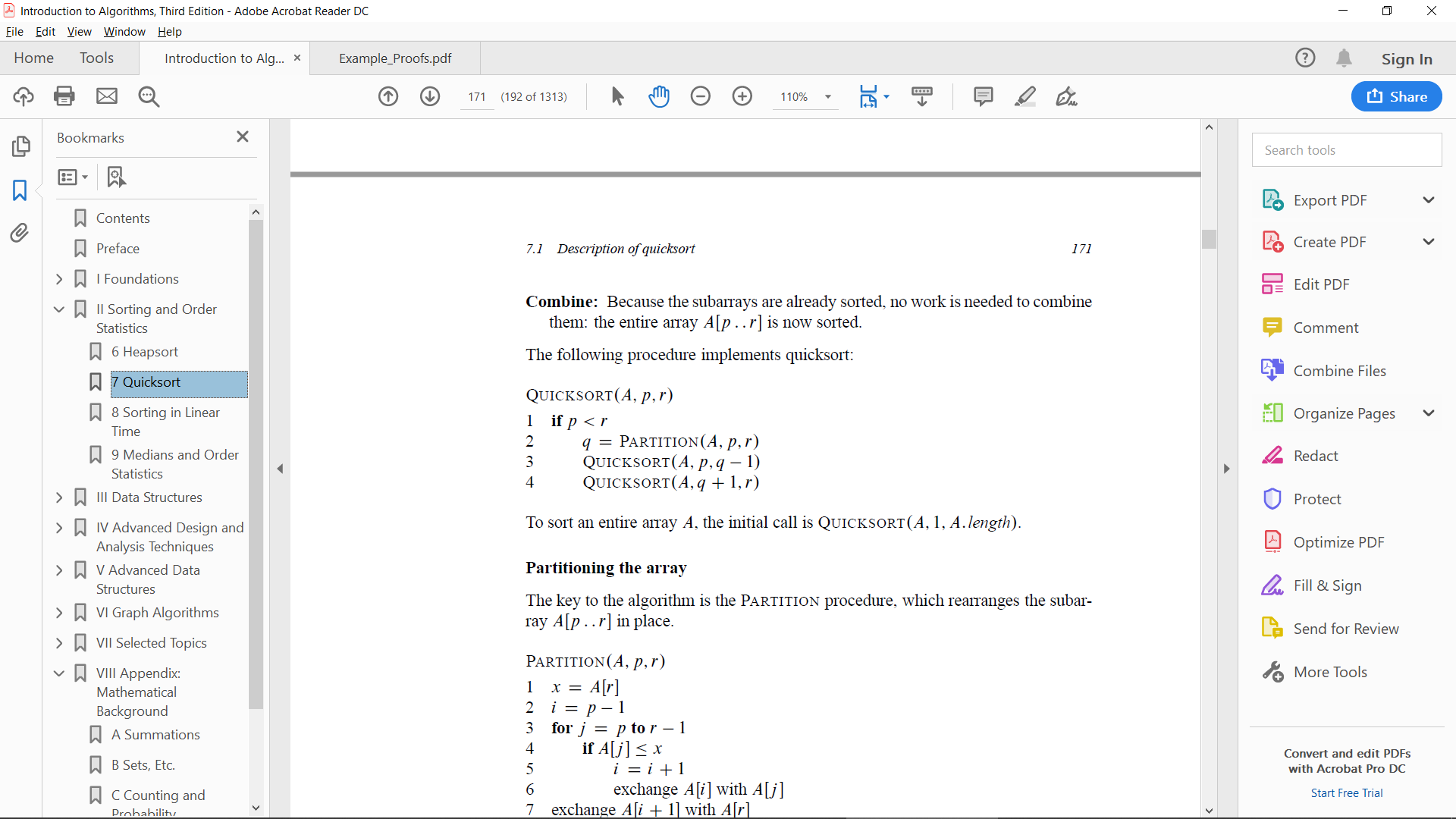


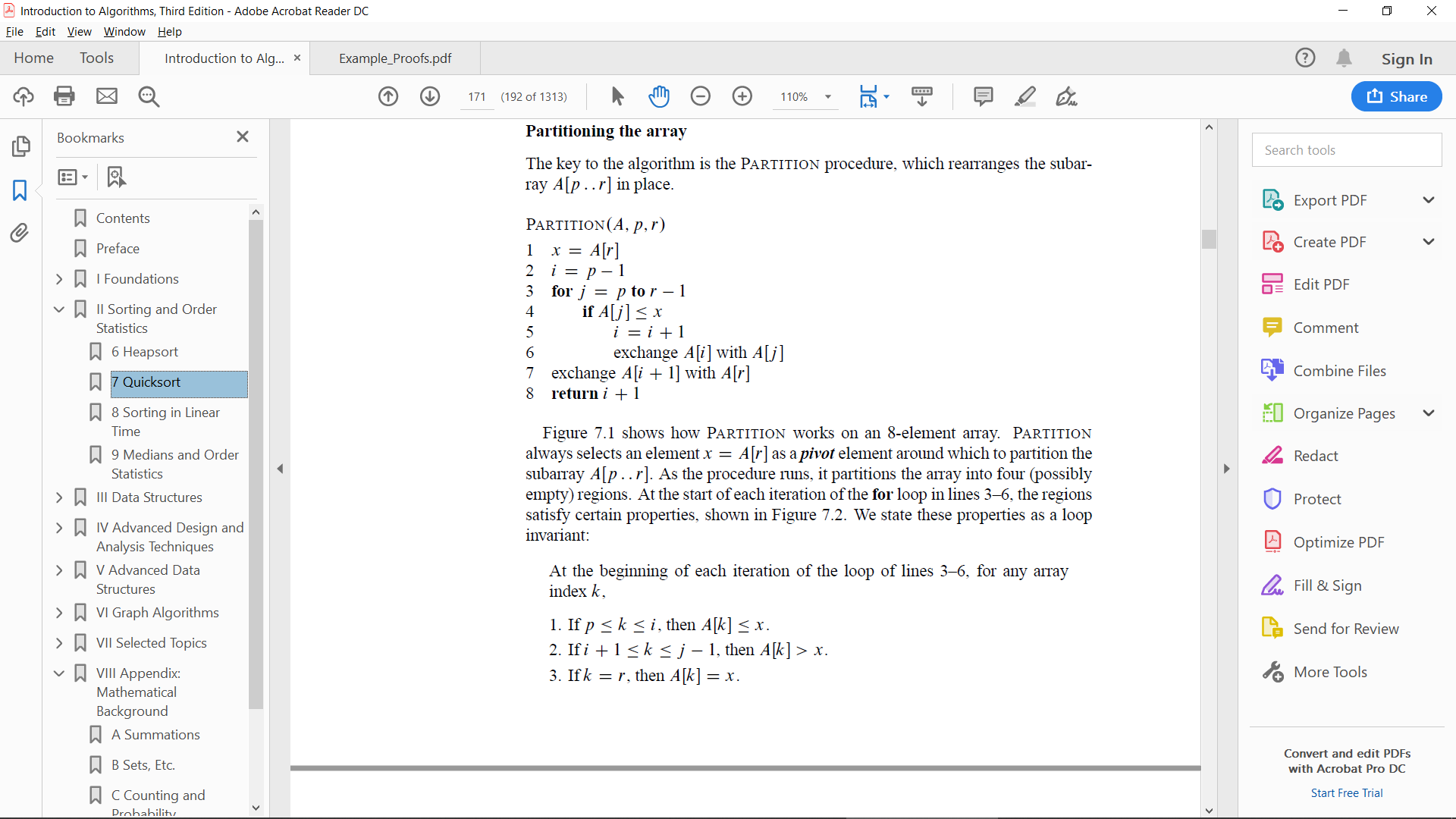
Quicksort:

* divide and conquer
* The worst-case behavior for quicksort occurs when

the partitioning routine produces one subproblem

with n-1 elements and one with 0 elements.





Merge Sort:

* proves first that the subroutine Merge

correctly takes two sorted subarrays and

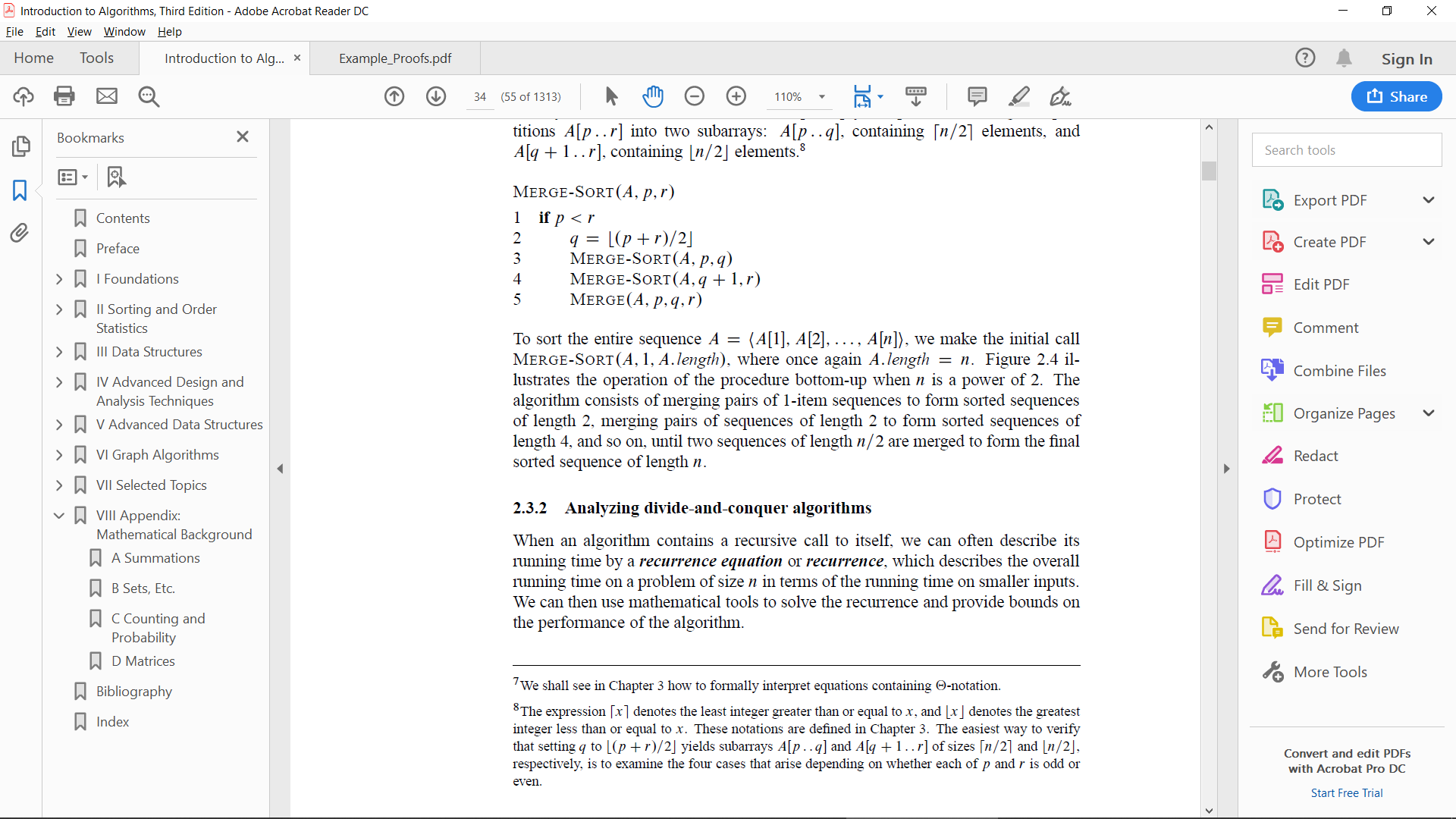
merges them into a single sorted array.

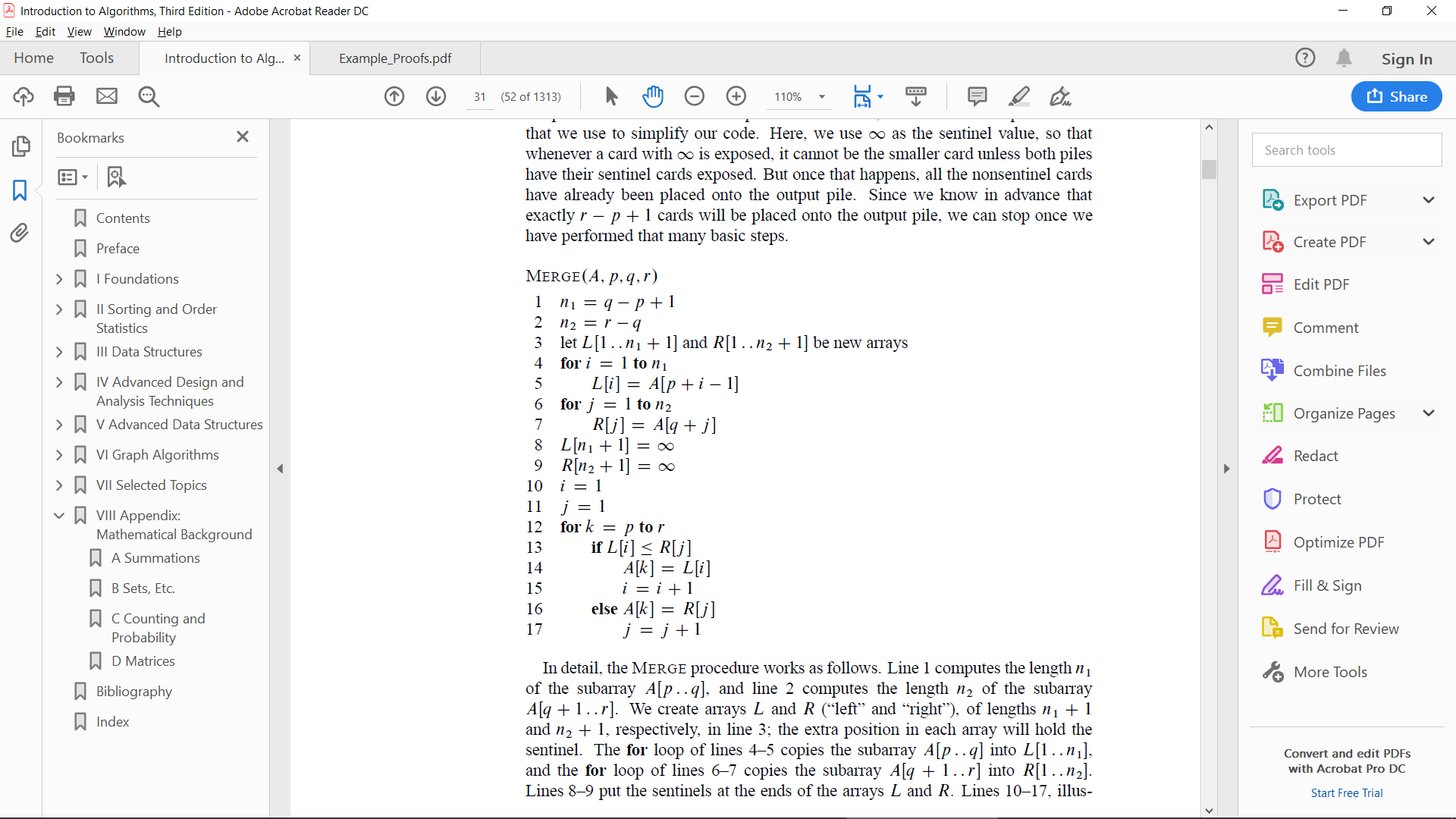
It then uses a Proof by Strong Induction

to show that the outer, Merge Sort function

properly sorts a given array.

* closely follows the divide-and-conquer paradigm
* Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
* Conquer: Sort the two subsequences recursively using merge sort.
* Combine: Merge the two sorted subsequences to produce the sorted answer.





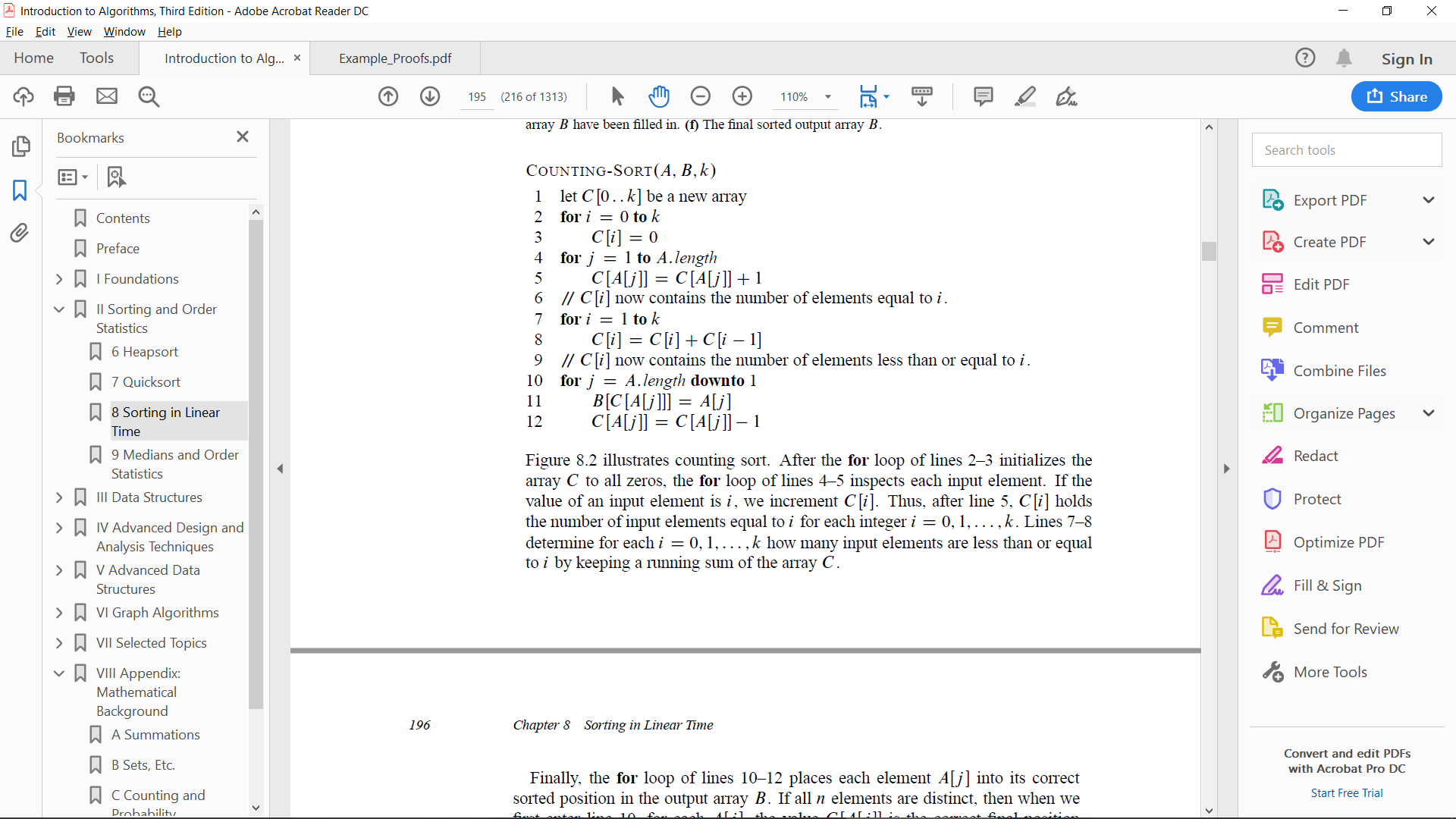
Counting sort:

* Linear-time
* Counting sort determines, for each input element x,

the number of elements less than x. It uses this

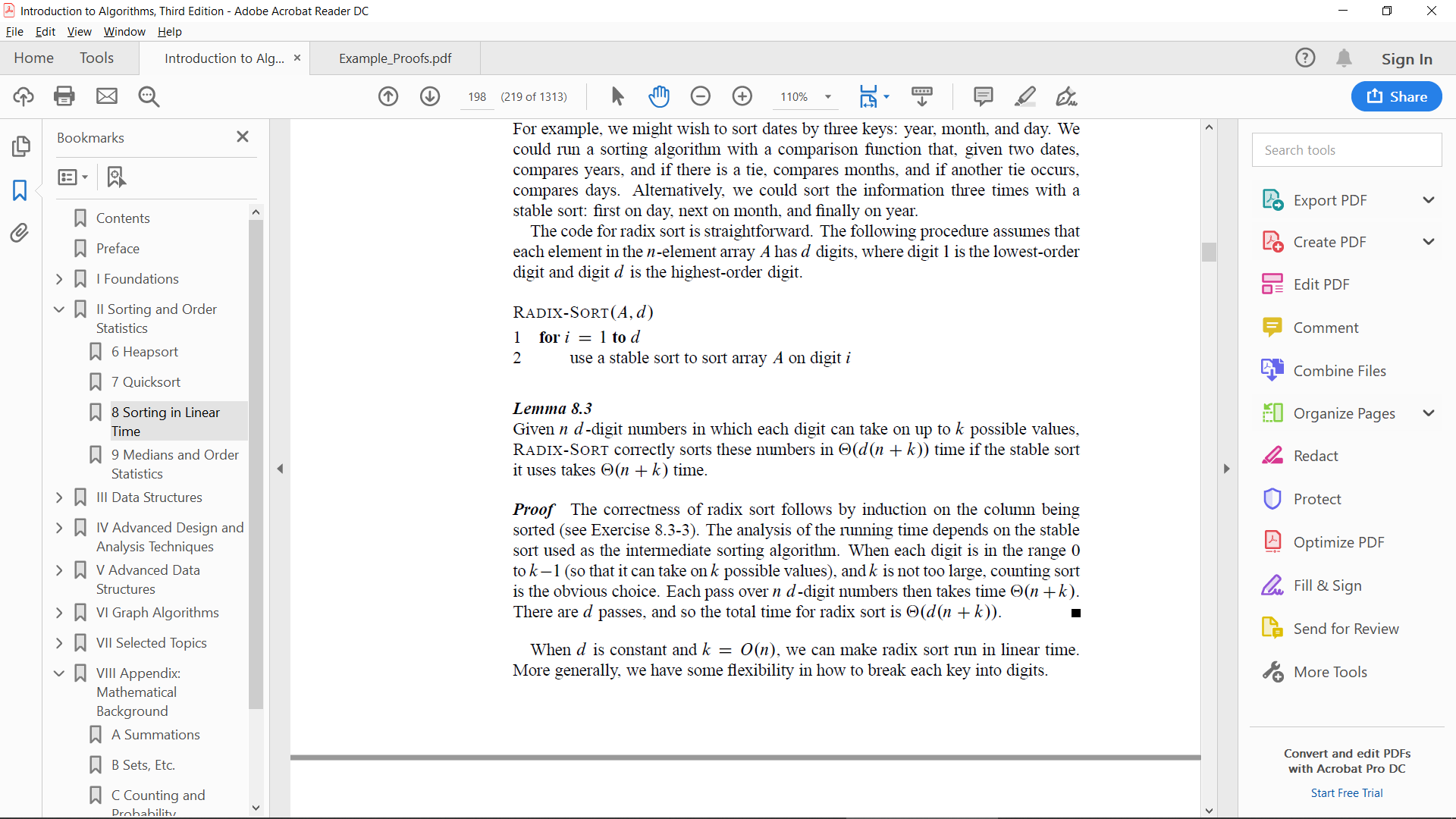
information to place element x directly into its

position in the output array.



Radix sort:

* solves the problem of card sorting by sorting on the least significant digit first.
* Linear-time

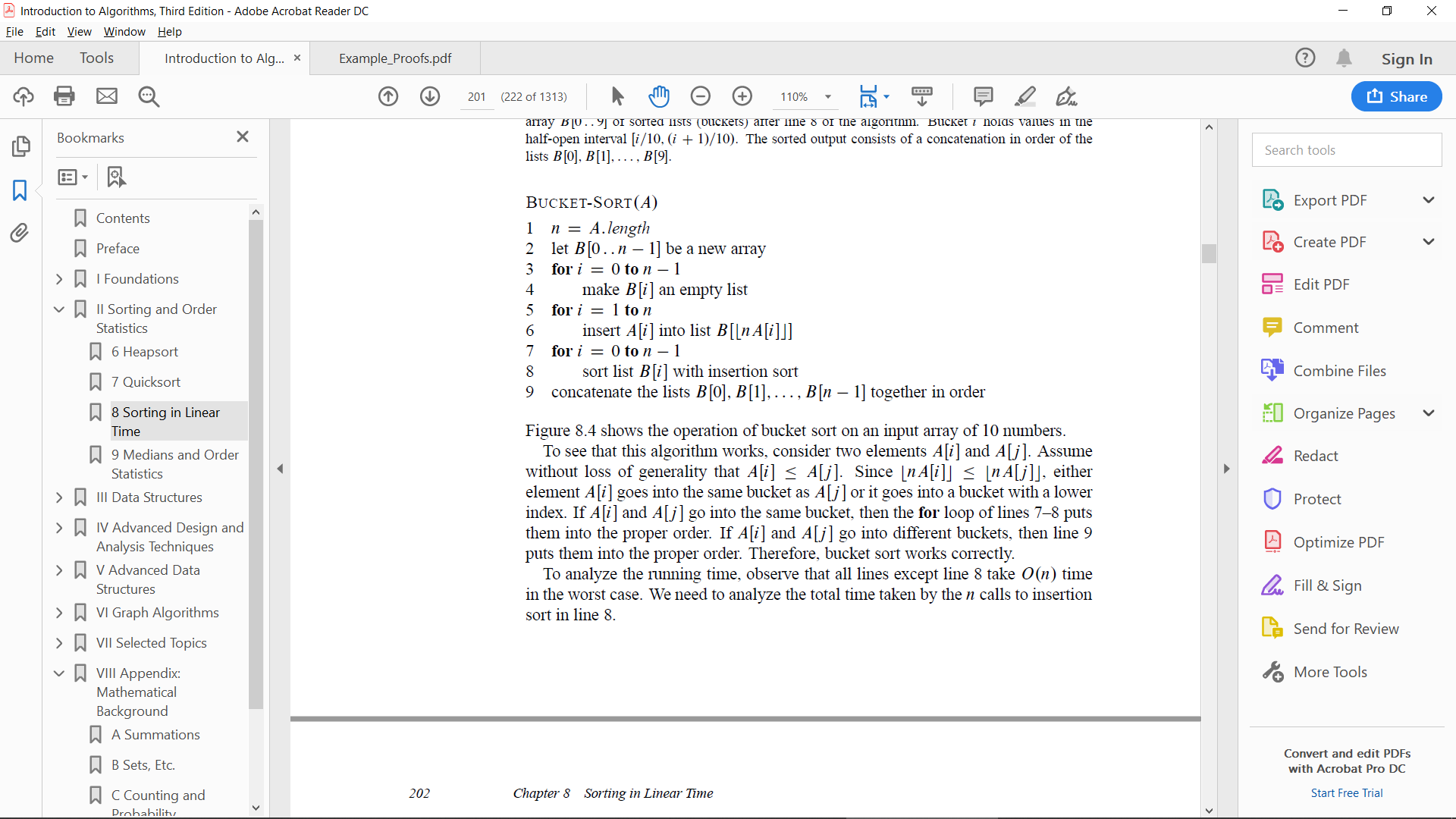


Bucket sort:

* Linear-time
* assumes that the input is drawn from a uniform distribution
* Bucket sort divides the interval [O,1) into n equal-sized

subintervals, or buckets, and then distributes the n input

numbers into the buckets.

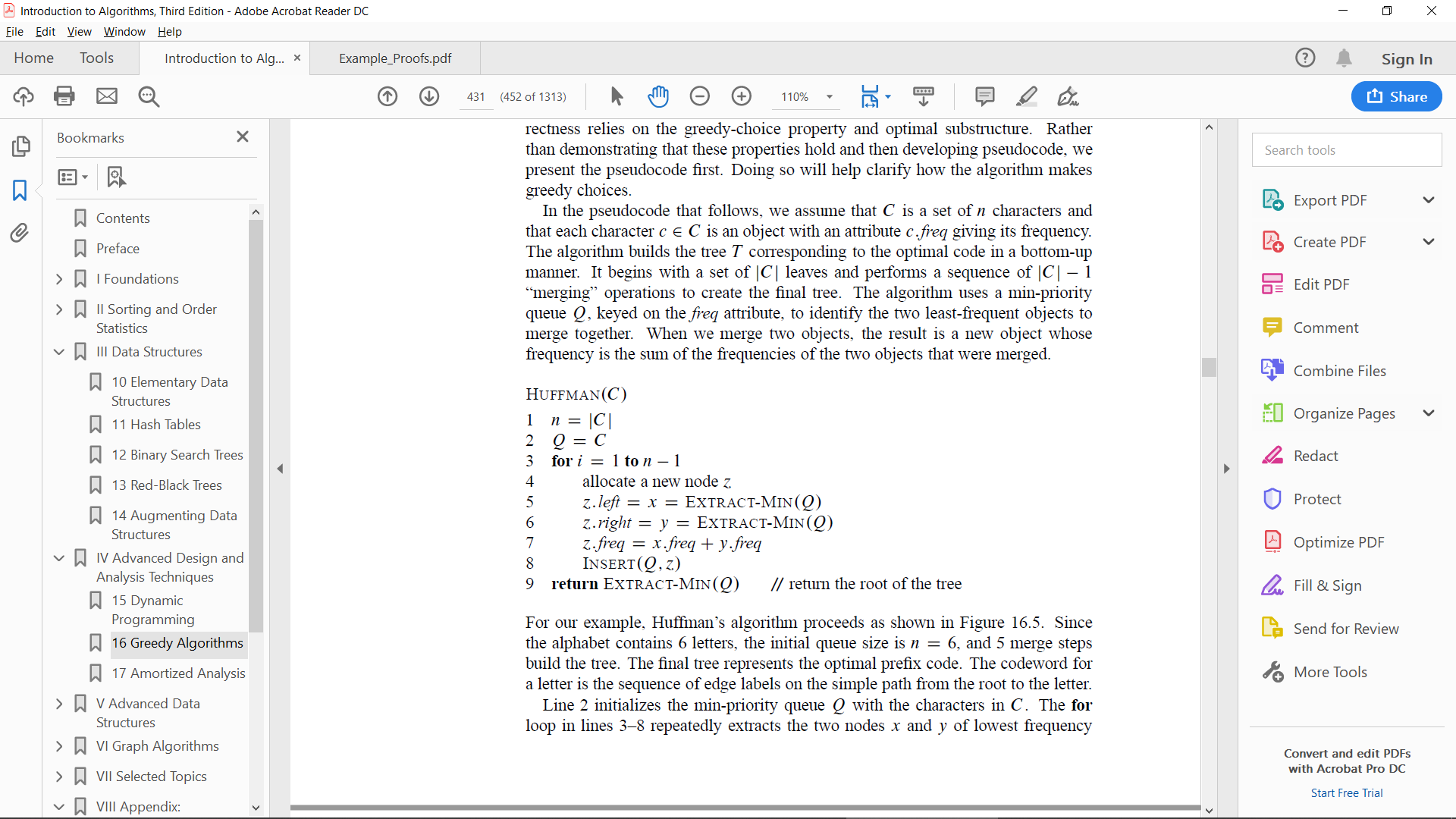


Huffman:

* total running time of HUFFMAN on a set of n characters is O(nlg(n))
* To prove that the greedy algorithm HUFFMAN is correct, we show that

the problem of determining an optimal prefix code exhibits the

greedy-choice and optimal substructure properties.



BFS:

* Uses FIFO queue
* finds the distance to each reachable vertex in a

graph G ={V,E} from a given source vertex *s* in V.

* BFS below assumes that the input graph G ={V,E} is

represented using adjacency lists.

DFS:

* a directed graph is acyclic if and only if a depth-first

search yields no “back” edges (Lemma 22.11).

* explores edges out of the most recently discovered vertex

that still has unexplored edges leaving it. Once all of ’s

edges have been explored, the search “backtracks” to explore

edges leaving the vertex from which was discovered. This

process continues until we have discovered all the vertices

that are reachable from the original source vertex. If any

undiscovered vertices remain, then depth-first search selects

one of them as a new source, and it repeats the search from that source.

The algorithm repeats this entire process until it has discovered every vertex.

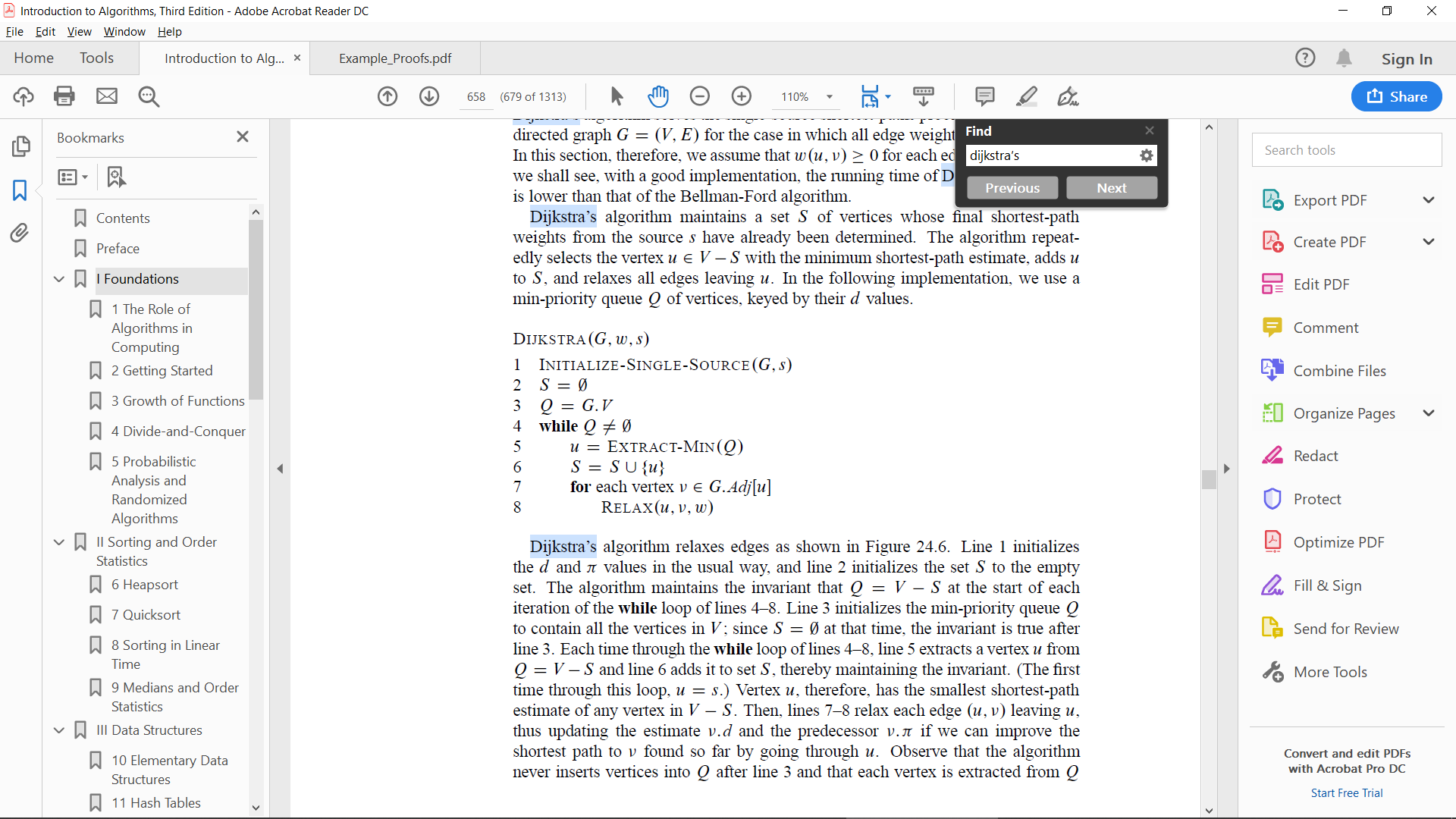
Dijkstra’s:

* solves the single-source shortest-paths problem on a

weighted, directed graph G ={V,E} for the case in

which all edge weights are nonnegative.

* uses a greedy strategy



Bellman-Ford:

* The Bellman-Ford algorithm runs in time O(VE), since

the initialization in line 1 takes‚ theta(V) time, each

of the |V|-1 passes over the edges in lines 2–4 takes

theta(E) time, and the for loop of lines 5–7 takes O(E) time.

* solves the single-source shortest-paths problem in the general

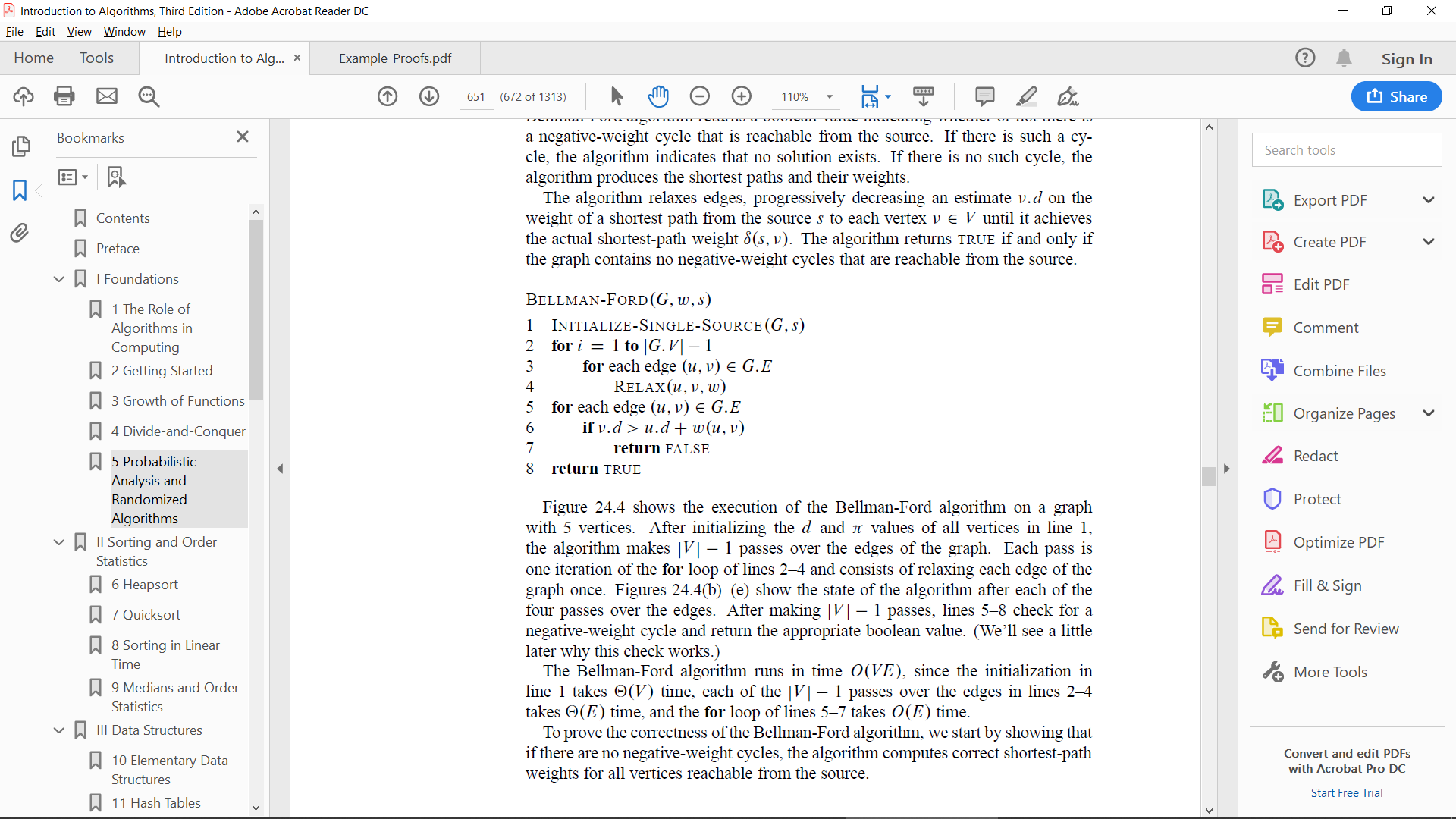
case in which edge weights may be negative.

* Bellman-Ford algorithm returns a boolean value indicating whether

or not there is a negative-weight cycle that is reachable from the source.

(returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source)

* If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.



* 1. Knapsack:
* Uses Dynamic rather than Greedy

Dynamic Programming:

* We typically apply dynamic programming to *optimization problems*

