**Chapter 7 Summary**

**Insertion Sort:** Insertion sort ensure that that element in position 0 through p is sorted in order. It also makes use of the fact that elements in position 0 through p – 1 are already known to be in sorted order. This is efficient when the size of the input is small.

**Inversion:** An inversion in an array of numbers is any ordered pair having the property that I < j but a{i} > a{j}.

**Shell Sort:** One of the first algorithms to break the quadratic time barrier. This works by comparing elements that are distant; the distant between the comparisons decreases as the algorithm runs until the last phase.

**Heap Sort:** During the first phase we are to build a heap, which uses less than 2N comparisons. In the second phase, the ith deleteMax uses at most less than 2[log(N – I + 1)] comparisons. The goal is to show that there very heaps that have small cost sequences.

**Merge Sort:** Runs in O(NLogN) worst-case running time, and the number of comparison is optimal. The fundamental operation in this algorithm is merging two sorted lists. Since the list are sorted it can be done in one pass through the input. When the input list is exhausted, the remainder of the other list is copied to C.

**Quick Sort:** It’s a fast sorting algorithm that is recursive. The analysis requires solving a recurrence formula. The running time of quicksort is equal to the running time of two recursive calls plus the linear time spent in the partition.

**Decision Tree:** Is an abstraction used to prove lower bounds. This is a binary tree. Each node represents a set of possible orderings, consistent with comparisons that have been made, among the elements. This type of lower bound argument is also known as an information-theoretic lower bound.

**Bucket sort**: The input must consist of only positive integers smaller than M. If this is the case, then the algorithm is simple: Keep an array called count, of size M, which is initialized to all 0’s. After all the input is read, scan the count array, printing out a representation of the sorted list.

**Radix Sort:** A non-[comparative](https://en.wikipedia.org/wiki/Comparison_sort) [integer](https://en.wikipedia.org/wiki/Integer_sorting) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm) that sorts data with integer keys by grouping keys by the individual digits which share the same [significant](https://en.wikipedia.org/wiki/Significant_figures) position and value.

**Counting Radix Sort**: An alternative implementation of radix sort that avoids using Array Lists. We maintain a count of how many items would go in each bucket; this information can go into an array count.

**Multiway Merge**: Merging two runs is done by winding each input tape to the beginning of each run. The smaller element is found, placed on an output tape, and the appropriate input tape is advanced.

**Replacement Selection:** M records are read into memory and placed in a priority queue. We perform a deleteMin , writing the smallest record to an output tape. Priority queues are smaller by one element so we are able to store in a dead space.

**Chapter 8 Summary**

**Equivalence Relations:** A relation R that satisfies three properties:

1. (Reflexive) aRa, for all a in element S.
2. (Symmetric) aRb if and only if bRa.
3. (Transitive) aRb and bRc implies that aRc.

**Disjoint:** This representation is when all relations are false. Each set has a different element.

**Union-by-size**: A simple improvement is always to make the smaller tree a subtree of the larger, breaking ties by any method. If we prove that unions are done by size, the depth of any node is never more than log N.

**Union-by-height**: We keep track of the height, instead of the size, of each tree and perform unions by making the shallow tree a subtree of the deeper tree.

**Path Compression:** This is performed during a find operation and is independent of the strategy used to perform unions. The effect of path compression is that every node on the path from x to the root has its parent changed to the root.

**Iterated logarithm:** which represents the number of times the logarithm needs to be iteratively applied until we reach one, is a very slowly growing function.

**Partial Find:** A partial find operation specifies the search item and the node up to which the path compression is performed. The node that will be used is the node that would have been the root at the time the matching find was performed.

**Recursive Decomposition:** Divide each tree into two halves: a top half and a bottom half. We make sure that the top half and bottom half has exactly the same total number of partial find operations. A formula is then used to figure out the total path compression cost in the tree in terms of path compression of the top half and then the bottom half.

**Lemma 8.1 –** When executing a sequence of union instructions, a node of rank r > 0 must have at least one child or rank 0, 1 ,..., r -1.

**Lemma 8.2 –** At any point in the union/find algorithm, the ranks of the nodes on a path from the leaf to a root increase monotonically.

**An Application:** An example of the use of the union/find data structure is the generation of mazes. The union/find data structure I used to represent sets of cells that are connected to each other. The running time is calculated by the union/find cost.