**Introduction to Algorithms Notes**

**Chapter 1:** Foundations

* 1. Algorithms
* Algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
* The **instance** of a problem consists of the input needed to compute a solution to the problem.
* A **data structure** is a way to store and organize data in order to facilitate access and modifications.

**Chapter 2:** Getting Started

2.1 Insertion sort

* The first algorithm we will be studying is insertion sort, which solves the sorting problem.
* The numbers that we would like to sort are known as **keys**.
* The input comes as an array with n elements.
* Insertion sort is efficient for sorting a small number or elements.
* Here is a visual representation of insertion sort:

A picture containing diagram, font

Description automatically generated

* Here is pseudocode for insertions sort:

A picture containing text, font, receipt, white

Description automatically generated

2.2 Analyzing algorithms

* Has come to mean predicting the resources that algorithm requires. Computational time is the main thing we want to measure.
* For this book, we assume a generic one-processer, random-access machine model of computations.
* The time taken by the insertion sort procedure depends on the input. You could have 5 inputs or 1000 inputs.
* Also, the time insertion sort takes depends also on how sorted the list is. Therefore, lists of the same size can have different execution times.
* Here is an example of the analysis of the insertion sort algorithm, where is the number of times the while loop on line 5 executes for that value of .

A picture containing text, font, receipt, screenshot

Description automatically generated

* The best case for insertion sort is when the array is already sorted.
* The worst case is when the array is reverse sorted.
* The **average case** is often roughly as bad as the worst case.

**Chapter 12:** Binary Search Trees

* The search tree data structure supports many dynamic-set operations, including search min, max, predecessor, successor, insert, and delete.
* Basic operations on a binary search tree take time proportional to the height of the tree.
  + For a tree with n nodes, such operations run in worst-case time.
  + If the tree is a linear chain of n nodes, however, the same operations take worst-case time.
* What is a binary search tree?
  + Each node is an object.
  + Each node contains a key, left, right, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate attribute contains the value NIL. The root node is the only node in the tree whose parent is NIL.
  + The keys in a binary search tree are always stored in such a way as to satisfy the **binary-search-tree property:**
    - Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key <= x.key. if y is a node in the right subtree of x, then y.key >= x.key.