Noah Parker
Chapter 14 Study Sheet

The selection sort algorithm sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in each array.

Merge sort is a “divide and conquer” algorithm wherein we first divide the problem into subproblems. When the solutions for the subproblems are ready, we combine them together to get the final solution to the problem.

This is one of the algorithms which can be easily implemented using recursion as we deal with the subproblems rather than the main problem.

```
class SelectionSort
{
  void sort(int arr[])
  {
    int n = arr.length;
    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n-1; i++)
    {
      // Find the minimum element in unsorted array
      int min_idx = i;
      for (int j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
          min_idx = j;
      // Swap the found minimum element with the first
      // element
      int temp = arr[min_idx];
      arr[min_idx] = arr[i];
      arr[i] = temp;
    }}
```

```
public static void mergeSort(int[] a, int n) {
  if (n < 2) {
    return;
  }
  int mid = n / 2;
  int[] l = new int[mid];
  int[] r = new int[n - mid];
  for (int i = 0; i < mid; i++) {
    l[i] = a[i];
  }
  for (int i = mid; i < n; i++) {
    r[i - mid] = a[i];
  }
  mergeSort(l, mid);
  mergeSort(r, n - mid);
  merge(a, l, r, mid, n - mid);
}
```

The algorithm can be described as the following 2 step process:

1. Divide: In this step, we divide the input array into 2 halves, the pivot being the midpoint of the array. This step is carried out recursively for all the half arrays until there are no more half arrays to divide.
2. Conquer: In this step, we sort and merge the divided arrays from bottom to top and get the sorted array.

Linear search is used to search a key element from multiple elements.

Linear search is less used today because it is slower than binary search and hashing.

Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster-searching comparison to Linear search.

```
public class LinearSearchExample{
public static int linearSearch(int[] arr, int key){
    for(int i=0;i<arr.length;i++){
      if(arr[i] == key){
        return i;
      }
    }
    return -1;
  }
  public static void main(String a[]){
    int[] a1= {10,20,30,50,70,90};
    int key = 50;
    System.out.println(key+" is found at index:
"+linearSearch(a1, key));
  }
}
```

Calculating an Algorithm's Running Time The acronym Big-O is used to categorize the temporal complexity of algorithms. A good algorithm not only discovers a solution, but it does so rapidly and effectively. We evaluate the time complexity of an algorithm rather than the exact number of operations it will complete. Time complexity is a measure of how much longer an algorithm will take to run (in number of operations). (in number of operations) as the size of the input increases. This only examines the proportional time of the largest components of the algorithm.

The table blew describes all the way.

O(1) - Constant Time	O(log n) - Logarithmic Time	O(n) - Linear Time	O(n log n) - Linearithmic time	O(n2) - Quadratic Time	O(2n) - Exponential Time	O(n!) - Factorial Time
The algorithm performs a constant number of operations regardless of the size of the input. Examples: Access a single number in an array by index Add 2 numbers together.	As the size of input n increases, the algorithm's running time grows by log(n). This rate of growth is relatively slow, so O(log n) algorithms are usually very fast. As you can see in the table below, when n is 1 billion, log(n) is only 30.	The running time of an algorithm grows in proportion to the size of input. Examples: Search through an unsorted array for a number Sum all the numbers in an array Access a single number in a LinkedList by index	log(n) is much closer to n than to n2, so, this running time is closer to linear than to higher running times (and no one actually says "Linearithmic"). Examples: Sort an an array with QuickSort or Merge Sort. When recursion is involved, calculating the running time can be complicated. You can often work out a sample case to estimate what the running time will be. See Quick Sort for more inf	The algorithm's running time grows in proportion to the square of the input size and is common when using nested loops. Examples: Printing the multiplication table for a list of numbers Insertion Sort	Table of n vs. 2n (reverse of log(n) table) n 2n 1 2 10 ~1000 20 ~1M 30 ~1B	This algorithm's running time grows in proportion to n!, a really large number. O(n!) becomes too large for modern computers as soon as n is greater than 15+ (or upper teens). This issue shows up when you try to solve a problem by trying out every possibility, as in the traveling salesman problem. Examples: Go through all Permutations of a string Traveling Salesman Problem (brute-force solution)

This tables information came from https://www.learneroo.com/modules/106/nodes/559