UNIT II

Searching Techniques

Searching is the process of looking through the data contained in a data structure and determining if a specific value is present. (And potentially returning it.) The contains methods of the ArrayList, for example, is a method that searches the list for a given value and returns true of false.

1. Linear Search

The most basic search algorithm is a *linear search*. This is just a fancy name for "start at the first element and go through the list until you find what you are looking for." It is the simplest search. Here is a simple implementation for an array:

```
public static int linearSearch(int[] a, int value) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        if (a[i] == value)
            return i;
    }
    return -1;
}
```

If you look carefully at it, you'll realize that the efficiency is O(N). Is it possible to do better than this, though?

2. Binary Search

Binary search is a searching algorithm that is faster than O(N). It has a catch, though: It requires the array you are searching to already be sorted. If you have a sorted array, then you can search it as follows:

1. Imagine you are searching for element *v*.
2. Start at the middle of the array. Is *v* smaller or larger than the middle element? If smaller, then do the algorithm again on the left half of the array. If larger, then do the algorithm again on the right half of the array.

Basically, with every step, we cut the number of elements we are searching for in half. Here is a simple implementation:

```
public static int binarySearch(int[] a, int value) {
    int lowIndex = 0;
    int highIndex = a.length - 1;
    while (lowIndex <= highIndex) {
        int midIndex = (lowIndex + highIndex) / 2;
        if (value < a[midIndex])
            highIndex = midIndex - 1;
        else if (value > a[midIndex])
            lowIndex = midIndex + 1;
        else // found it!
            return midIndex;
    }
    return -1; // The value doesn't exist
}
```

The efficiency of this is more complicated than the things we've been analyzing so far. We aren't doing N work in the worst case, because even in the worst case we are cutting the array in half each step. When an algorithm cuts the data in half at each step, that efficiency is *logarithmic*. In this case, O(log2 N)). In efficiency, we frequently drop the base 2 from logarithmic efficiency, so we can just write this as O(log N) This is significantly better than an O(N) algorithm.