Chapter 39: Searching

Section 39.1: Binary Search

Introduction

Binary Search is a Divide and Conquer search algorithm. It uses `O(log n)` time to find the location of an element in a search space where `n` is the size of the search space.

Binary Search works by halving the search space at each iteration after comparing the target value to the middle value of the search space.

To use Binary Search, the search space must be ordered (sorted) in some way. Duplicate entries (ones that compare as equal according to the comparison function) cannot be distinguished, though they don't violate the Binary Search property.

Conventionally, we use less than (<) as the comparison function. If a < b, it will return true. if a is not less than b and b is not less than a, a and b are equal.

Example Question

You are an economist, a pretty bad one though. You are given the task of finding the equilibrium price (that is, the price where supply = demand) for rice.

Remember the higher a price is set, the larger the supply and the lesser the demand

As your company is very efficient at calculating market forces, you can instantly get the supply and demand in units of rice when the price of rice is set at a certain price `p`.

Your boss wants the equilibrium price ASAP, but tells you that the equilibrium price can be a positive integer that is at most `10^17` and there is guaranteed to be exactly 1 positive integer solution in the range. So get going with your job before you lose it!

You are allowed to call functions `getSupply(k)` and `getDemand(k)`, which will do exactly what is stated in the problem.

Example Explanation

Here our search space is from `1` to `10^17`. Thus a linear search is infeasible.

However, notice that as the `k` goes up, `getSupply(k)` increases and `getDemand(k)` decreases. Thus, for any `x > y`, `getSupply(x) - getDemand(x) > getSupply(y) - getDemand(y)`. Therefore, this search space is monotonic and we can use Binary Search.

The following psuedocode demonstrates the usage of Binary Search:

```
high = 100000000000000000      <- Upper bound of search space
low = 1                        <- Lower bound of search space
while high - low > 1
    mid = (high + low) / 2     <- Take the middle value
    supply = getSupply(mid)
    demand = getDemand(mid)
    if supply > demand
        high = mid              <- Solution is in lower half of search space
```

```
    else if demand > supply
        low = mid              <- Solution is in upper half of search space
    else                       <- supply==demand condition
        return mid             <- Found solution
```

This algorithm runs in `~O(log 10^17)` time. This can be generalized to `~O(log S)` time where S is the size of the search space since at every iteration of the **`while`** loop, we halved the search space (*from [low:high] to either [low:mid] or [mid:high]*).

C Implementation of Binary Search with Recursion

```
int binsearch(int a[], int x, int low, int high) {
    int mid;

    if (low > high)
      return -1;

    mid = (low + high) / 2;

    if (x == a[mid]) {
        return (mid);
    } else
    if (x < a[mid]) {
        binsearch(a, x, low, mid - 1);
    } else {
        binsearch(a, x, mid + 1, high);
    }
}
```

Section 39.2: Rabin Karp

The Rabin–Karp algorithm or Karp–Rabin algorithm is a string searching algorithm that uses hashing to find any one of a set of pattern strings in a text.Its average and best case running time is O(n+m) in space O(p), but its worst-case time is O(nm) where n is the length of the text and m is the length of the pattern.

Algorithm implementation in java for string matching

```
void RabinfindPattern(String text,String pattern){
    /*
    q a prime number
    p hash value for pattern
    t hash value for text
    d is the number of unique characters in input alphabet
    */
    int d=128;
    int q=100;
    int n=text.length();
    int m=pattern.length();
    int t=0,p=0;
    int h=1;
    int i,j;
//hash value calculating function
    for (i=0;i<m-1;i++)
            h = (h*d)%q;
        for (i=0;i<m;i++){
        p = (d*p + pattern.charAt(i))%q;
        t = (d*t + text.charAt(i))%q;
        }
//search for the pattern
```

```
    for(i=0;i<end-m;i++){
        if(p==t){
//if the hash value matches match them character by character
            for(j=0;j<m;j++)
                if(text.charAt(j+i)!=pattern.charAt(j))
                break;
            if(j==m && i>=start)
                System.out.println("Pattern match found at index "+i);
        }
        if(i<end-m){
            t =(d*(t - text.charAt(i)*h) + text.charAt(i+m))%q;
            if(t<0)
                t=t+q;
        }
    }
}
```

While calculating hash value we are dividing it by a prime number in order to avoid collision.After dividing by prime number the chances of collision will be less, but still ther is a chance that the hash value can be same for two strings,so when we get a match we have to check it character by character to make sure that we got a proper match.

t =(d*(t - text.charAt(i)*h) + text.charAt(i+m))%q;

This is to recalculate the hash value for pattern,first by removing the left most character and then adding the new character from the text.

Section 39.3: Analysis of Linear search (Worst, Average and Best Cases)

We can have three cases to analyze an algorithm:

1. Worst Case
2. Average Case
3. Best Case

```
#include <stdio.h>

// Linearly search x in arr[].  If x is present then return the index,

// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
        if (arr[i] == x)
         return i;
    }

    return -1;
}
```

/* Driver program to test above functions*/

```
int main()
```

```
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}
```

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be Θ(n)

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\theta((n+1)*(n+2)/2)}{(n+1)}$$

$$= \Theta(n)$$

Best Case Analysis (Bogus)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1) Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information. The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs. The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

For some algorithms, all the cases are asymptotically same, i.e., there are no worst and best cases. For example, Merge Sort does Θ(nLogn) operations in all cases. Most of the other sorting algorithms have worst and best cases. For example, in the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occur when the pivot elements always divide array in two halves. For insertion sort, the worst case occurs when the array is reverse sorted and the best case

occurs when the array is sorted in the same order as output.

Section 39.4: Binary Search: On Sorted Numbers

It's easiest to show a binary search on numbers using pseudo-code

```
int array[1000] = { sorted list of numbers };
int N = 100;  // number of entries in search space;
int high, low, mid; // our temporaries
int x; // value to search for

low = 0;
high = N -1;
while(low < high)
{
    mid = (low + high)/2;
    if(array[mid] < x)
        low = mid + 1;
    else
        high = mid;
}
if(array[low] == x)
    // found, index is low
else
    // not found
```

Do not attempt to return early by comparing array[mid] to x for equality. The extra comparison can only slow the code down. Note you need to add one to low to avoid becoming trapped by integer division always rounding down.

Interestingly, the above version of binary search allows you to find the smallest occurrence of x in the array. If the array contains duplicates of x, the algorithm can be modified slightly in order for it to return the largest occurrence of x by simply adding to the if conditional:

```
while(low < high)
    {
        mid = low + ((high - low) / 2);
        if(array[mid] < x || (array[mid] == x && array[mid + 1] == x))
            low = mid + 1;
        else
            high = mid;
    }
```

Note that instead of doing `mid = (low + high) / 2`, it may also be a good idea to try `mid = low + ((high - low) / 2)` for implementations such as Java implementations to lower the risk of getting an overflow for really large inputs.

Section 39.5: Linear search

Linear search is a simple algorithm. It loops through items until the query has been found, which makes it a linear algorithm - the complexity is O(n), where n is the number of items to go through.

Why O(n)? In worst-case scenario, you have to go through all of the n items.

It can be compared to looking for a book in a stack of books - you go through them all until you find the one that you want.

Below is a Python implementation:

```
def linear_search(searchable_list, query):
    for x in searchable_list:
        if query == x:
            return True
    return False

linear_search(['apple', 'banana', 'carrot', 'fig', 'garlic'], 'fig') #returns True
```