Algorithm

1. Short notes on Optimization

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand a smaller number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types: machine independent and machine dependent.

2. Different algorithms that I know.

Bubble Sort [Best: O(N), Worst, (N^2)]

Starting on the left, compare adjacent items and keep “bubbling” the larger one to the right (it’s in its final place). Bubble sort the remaining N -1 items.

- Though “simple” I found bubble sort nontrivial. In general, sorts where you iterate backwards (decreasing some index) were counter-intuitive for me. With bubble-sort, either you bubble items “forward” (left-to-right) and move the endpoint backwards (decreasing), or bubble items “backward” (right-toleft) and increase the left endpoint. Either way, some index is decreasing.
- You also need to keep track of the next-to-last endpoint, so you don’t swap with a non-existent item.

Selection Sort [Best/Worst: O(N^2)]
Scan all items and find the smallest. Swap it into position as the first item. Repeat the selection sort on the remaining N-1 items.

- I found this the most intuitive and easiest to implement — you always iterate forward (i from 0 to N-1), and swap with the smallest element (always i).

Quicksort [Best: O (N Lg N), Avg: O (N Lg N), Worst(N^2)]
There are many versions of Quicksort, which is one of the most popular sorting methods due to its speed (O (N lgN) average, but O(N^2) worst case). Here’s a few:

Using external memory:

- Pick a “pivot” item
- Partition the other items by adding them to a “less than pivot” sub list, or “greater than pivot” sub list
- The pivot goes between the two lists
- Repeat the quicksort on the sub lists, until you get to a sub list of size 1 (which is sorted).
- Combine the lists — the entire list will be sorted

3. **Why I am learning so many algorithms**

Algorithms are clearly specified means to solve problems.

You want to know two things about an algorithm:

1. Does it solve the problem?
2. Does it use resources efficiently?

If you write code that does not solve the problem, or if it solves the problem but uses resources inefficiently (for example, it takes too long or uses too much memory), then your code doesn't really help.

That's why we study algorithms. We want to know that our code is based on ideas that solve the problem and that we're using resources efficiently. And we want to know that our solution is correct and efficient for all possible situations, or at least to know that the cases in which our algorithm fails to meet these criteria are rare.

Even if you intend to just call functions in APIs and not design algorithms yourself, you should know about the algorithms and data structures used in implementing these APIs. No data structure is the best choice for every situation, and so you need to know the strengths and weaknesses of each.

4. Show analysis of a recursive algorithm

Example: Factorial

$n! = 1•2•3...n$ and $0! = 1$ (called initial case) So,

the recursive definition $n! = n•(n-1)!$

Algorithm $F(n)$ if n = 0 then return 1 //

base case else F(n-1) •n //

recursive call

Basic operation? multiplication during the recursive call

Formula for multiplication

$M(n) = M(n-1) + 1$ is a recursive

formula too. This is typical.

We need the initial case which corresponds to the base case M

$(0) = 0$

There are no multiplications

Solve by the method of *backward substitutions*

$M(n) = M(n\text{-}1) + 1$

$= [M(n\text{-}2) + 1] + 1 = M(n\text{-}2) + 2$ substituted $M(n\text{-}2)$ for $M(n\text{-}1)$

$= [M(n\text{-}3) + 1] + 2 = M(n\text{-}3) + 3$ substituted $M(n\text{-}3)$ for $M(n\text{-}2)$

... a pattern evolves

$= M(0) + n$

$= n$

Therefore $M(n)\ \varepsilon\ \Theta(n)$

5. Design an iterative and recursive algorithm and prove that your algorithm works

 Iterative (Binary Search) class

```
Main
{
        // find out if a key x exists in the sorted array A
// or not using binary search algorithm          public
static int binarySearch(int[] A, int x)
        {
                // search space is A[left..right]
int left = 0, right = A.length - 1;
```

```
                    // till search space consists of at-least one element
while (left <= right)
                    {
                              // we find the mid value in the search space and
                              // compares it with key value
          int mid = (left + right) / 2;
                              // overflow can happen. Use:
                              // int mid = left + (right - left) / 2;
                              // int mid = right - (right - left) / 2;
   // key value is found    if (x == A[mid])
{    return mid;
                              }
                              // discard all elements in the right search
space                                   // including the mid element
else if (x < A[mid]) {                                      right = mid - 1;
                              }
                              // discard all elements in the left search space
                              // including the mid element
                              else {
                                        left = mid + 1;
                              }
                    }
  // x doesn't exist in the array   return -1;
          }
          public static void main(String[] args)
```

```
    {
        int[] A = { 2, 5, 6, 8, 9, 10 };
        int key = 5;

        int index = binarySearch (A, key);

        if (index != -1) {
            System.out.println("Element found at index " + index);
        } else {
            System.out.println("Element not found in the array");
        }
    }
}
```

- **Recursive (Binary Search)**

```
class Main
{
    // Find out if a key x exists in the sorted array          //
A[left..right] or not using binary search algorithm          public
static int binarySearch(int[] A, int left, int right, int x)
    {
        // Base condition (search space is exhausted)
        if (left > right) {
            return -1;
        }
```

```
            // we find the mid value in the search space and
            // compares it with key value

            int mid = (left + right) / 2;

            // overflow can happen. Use below
            // int mid = left + (right - left) / 2;

  // Base condition (key value is found)   if (x ==
A[mid]) {
                return mid;
            }

            // discard all elements in the right search space
            // including the mid element             else if
(x < A[mid]) {                          return
binarySearch(A, left, mid - 1, x);
            }
            // discard all elements in the left search space
            // including the mid element
            else {
                return binarySearch(A, mid + 1, right, x);
            }
    }
    public static void main(String[] args)
```

```
    {
            int[] A = { 2, 5, 6, 8, 9, 10 };              int
key = 5;           int left = 0;            int right =
A.length - 1;              int index = binarySearch(A,
left, right, key);
            if (index != -1) {
                    System.out.println("Element found at index " + index);
            } else {
                    System.out.println("Element not found in the array");
            }
    }
}
```

- **Algorithm Prove**

We know that at each step of algorithm, our search space reduces to half. That means if initially our search space reduces to half. That means if initially our search space contains n elements, then after one iteration it contains n/2 then n/4 and so on ...
n - > n/2 -> n/4 …. ->1

Suppose after K steps our search apace is executed.

n/2k =1 n=
2k
k = long2n

There for time complexity of binary search algorithm is O(log2n) which is very efficient. Auxiliary space used by it is O(log2n) for recursive implementation due to call stack.