Data Structure & Algorithm

- Linear Data Structure
- Tree 69
- Graph 7
- recursion 2
- DFS/BFS
  - DFS template
    - iteration

```
func dfs(node Node) {
    if node == nil {
        return
    }
    stack := []Node{node}
    visited := map[int]bool{}

    for len(stack) > 0 {
        top := stack[len(stack)-1]
        stack = stack[:len(stack)-1]
        if visited[top.Val] { continue }
        visited[top.Val] = true
        for _, c := range top.Children {
            stack = append(stack, c)
        }
    }

}
```

    - recursion

```
func dfs( node Node, visited map[int]bool) {
    // terminator
    if node == nil || visited[node.Val] {
        return
    }
    // process current level
    visited[node.Val] = true
    // drill down to children
    for _, c := range node.Children {
        dfs(c, visited)
    }
}
```

  - BFS template

```
func bfs(node Node) {
    if node == nil {
        return nil
    }
    queue := []Node{node}
    visited := map[int]bool{}

    for len(queue) > 0 {
        // process current level
        for i := 0; i < len(queue); i++ {
            n :  = queue[i]
            for _, c := range n.Children {
                queue = append(queue, c)
            }
        }
        queue = queue[size:]
    }
}
```

- Go
  - String
    - byte: unit8 — ASCII
    - rune: int32
      - a more broader set of Unicode encoded in UTF-8 format
      - default type of character
  - stack or heap — Compiler decides where to allocate
    - bases on the scope and size
    - After returns, if the variable still has reference, it will allocate on the heap
    - otherwise, if the variable has small size, it will be allocate on the stack. If the size is large, it will allocate on the heap
      - explicit variable max stack size: 10 * 1024 * 1024
      - implicit: 64 * 1024
- Binary Search
  - LeetCode Problems
    - 69. Sqrt(x)
      - binary search

```
func sqrt(x float64) float64 {
    l := 0
    r := x
    //  6 decimal places
    for math.Abs(l - r) > 1e-6 {
        mid := l + (r - l) / 2
        // avoid overflow
        tmp := x / mid
        if math.Abs(tmp - mid) <= 1e-6 {
            return mid
        } else if mid < tmp {
            // since it is float, we cannot use + 1
            l = mid
        } else {
            r = mid
        }
    }
    return r
}
```

      - math
        - next = (prev + y0/ prev) / 2

```
func sqrt(x float64) float64 {
    r := x
    for math.Abs(x / r - r ) > 1e-6 {
        r = (r + x / r) / 2
    }
    return r
}
```

  - template

```
func bs(nums []int, target int) int {
    l, r := 0, len(nums) - 1
    for l <= r {
        mid := l + (r - l) / 2
        if nums[mid] == target {
            return mid
        } else if nums[mid] < target {
            l = mid + 1
        } else {
            r = mid - 1
        }
    }
}
```

- divide & conquer 9
- Greedy
  - we may not get final optimal result by taking optimal solution for each sub-problem — get use the optimal solution for each sub-problem
  - we does not store the result of sub-problems. Therefore, we does not backtrack
  - However, for DP, we store the results of sub-problems and will backtrack to find the optimal solution
- backtracking
  - leetcodeProblems 44