1 排序算法

1.1 选择排序

```
public static void selectSort(int[] number){
    int n = number.length;
    for(int i=0;i<n-1;i++){
        int min = i;
        //每次选择最小的数值放在下标为i的位置
        for(int j=i+1;j<n;j++){
            if(number[min]>number[j]){
                min = j;
            }
        }
        int temp = number[i];
        number[i] = number[min];
        number[min] = temp;
    }
}
```

1.2 冒泡排序

```
public static void bubbleSort(int[] number){
    int n = number.length;
    for(int i=0;i<n-1;i++){
        for(int j=0;j<n-1-i;j++){
            if(number[j]>number[j+1]){
                int temp = number[j];
                number[j] = number[j+1];
                numebr[j+1] = temp;
            }
        }
    }
}
```

1.3 插入排序

```
public static void insertSort(int[] number){
    for(int i=1;i<n;i++){
        int temp = number[i];
        //将number[i]插入到前i-1个数中的合适位置
        for(int j=i-1;j>=0;j--){
            if(number[j]<temp){
                number[j+1] = number[j];
            }else{
                break;
            }
        }
        number[j+1] = temp;
    }
}
```

1.4 希尔排序

```
public static void shellSort(int[] arr){
    //分组的插入排序
    int gap = arr.length;
    while(gap>1){
        gap = gap/3+1;
        for(int i = gap;i<arr.length;i++){
            int temp = arr[i];
            //每个分组的间隔为gap
            for(int j=i-gap;j>=0;j-=gap){
                if(arr[j]>temp){
                    arr[j+gap] = arr[j];
                }else{
                    break;
```

```
                }
            }
            arr[j+gap] = temp;
        }
    }
}
```

1.5 堆排序

```
public static void heapAdjust(int[] arr,int parent,int length){
    int child = parent*2+1;
    int temp = arr[parent];
    while(child<length){
        //找到子节点中最大值放到父节点上，升序需要构建最大堆
        if(child+1<length&&arr[child+1]>arr[child]){
            child++;
        }
        if(temp>arr[child]){
            break;
        }
        //用较大子节点的值替换父节点的值
        arr[parent] = arr[child];
        //父节点继续向下调整
        parent = child;
        child = parent*2+1;
    }
    arr[parent] = temp;
}
public static void heapSort(int[] number){
    int n = number.length;
    //只需要调整有子节点的节点就可以了
    for(int i=n/2-1;i>=0;i--){
        heapAdjust(arr,i,n);
    }
    //将大根堆的堆顶放到数组的最后一个位置，然后调整根节点使其继续满足大根堆
    for(int i=n-1;i>=0;i--){
        int temp = number[0];
        number[0] = number[i];
        number[i] = temp;
        heapAdjust(arr,0,i);
    }
}
```

1.6 归并排序

```
public static void mergeSort(int[] arr,int start,int end){
    int mid = (start+end)/2;
    while(start<end){
        //分割数组
```

```
        mergeSort(arr,start,mid);
        mergeSort(arr,mid+1,end);
        i = start;
        j = mid+1;
        //创建临时存储数据
        int[] temp = new int[end-start+1];
        int idx = 0;
        while(i<=mid&&j<=end){
            if(arr[i]<arr[j]){
                temp[idx++] = arr[i];
            }else{
                temp[idx++] = arr[j];
            }
        }
        while(i<=mid){
            temp[idx++] = arr[i++];
        }
        while(j<=end){
            temp[idx++] = arr[j++];
        }
        idx = 0;
        for(int i=start;i<=end;i++){
            arr[i] = temp[idx++];
        }
    }
}
```

1.7 快排

```
public static void quickSort(int[] arr,int left,int right){
    int l = left;
    int r = right;
    int temp = arr[l];
    while(l<=r){
        while(l<=r&&arr[r]<temp)r--;
        arr[l]  = arr[r];
        while(l<=r&&arr[l]>temp)l++;
        arr[l] = arr[r];
    }
    arr[l] = temp;
    quickSort(arr,left,l);
    quickSort(arr,l+1,right);
}
```

2 最短路径算法

2.1 Dijkstra

```
public static int[] DijKstra(int[][] graph,int s){
    int nodeNumber = graph[0].length;
    int[] vis = new int[nodeNumber];  //标记当前节点是否已经被使用过
    int[] dis = new int[nodeNumber];   //所有点距离s点的距离
     //初始化所有点之间的距离为无穷大
    for(int i=0;i<nodeNumber;i++){
        dis[i] = Integer.MAX_VAlUE;
    }
    int dis[s] = 0;
    for(int i=0;i<nodeNumber;i++){
        int node = -1;
        int temp = Integer.MAX_VALUE;
        for(int j=0;j<nodeNumber;j++){
            if(vis[j]==0&&dis[j]<temp){
                temp = dis[j];
                node = j;
            }
        }
        vis[node] = 1;
        //第一步的话将s点加入初始化所有点到s的距离
        for(int j=0;j<nodeNumber;j++){
            if(vis[j]==0&&(dis[node][j]+temp<dis[j])){
                dis[j] = temp+dis[node][j];
            }
        }
    }
    return dis;
}
```

2.2 Floyd

```
public static int[][] Floyd(int[][] graph){
    int[] nodeNumber = graph[0].length;
    int[][] dis = new int[nodeNumer][nodeNumber];
    //初始化所有点之间的距离为无穷大
    for(int i=0;i<nodeNumber;i++){
        for(int j=0;i<nodeNumber;j++){
            dis[i][j] = Integer.MAX_VALUE;
        }
    }
    for(int k=0;k<nodeNumber;k++){
        for(int i=0;i<nodeNumber;i++){
            for(int j=0;j<nodeNumber;j++){
                dis[i][j] = Math.min(dis[i][j],dis[i][k]+dis[k][j])
            }
        }
    }
    return dis;
}
```

3 二叉树遍历

3.1 递归

3.2 非递归

3.3 层次遍历

4 动态规划

4.1 01背包

```
//w物品重量，v物品价值，c背包容量
public static int maxPackage(int[] w,int[] v,int c){
    int n = w.length;
    int[][] dp = new int[n+1][c+1];
    //i表示前几个物品装入背包，j表示当前背包容量，dp表示前i个物品装入容量为j的背包的最大值
    for(int i=1;i<=n;i++){
        for(int j=1;j<=c;j++){
            //i-1是第i个物品的下标
            if(j<w[i-1]){
                dp[i][j] = dp[i-1][j];
            }else{
                dp[i][j] = Math.max(dp[i-1][j],dp[i-1][j-w[i-1]]+v[i-1]);
            }
        }
    }
    return dp[n][c];
}
```

4.2 最小编辑距离

5 设计模式

5.1 单例

5.1.1 静态常量（饿汉式）

```
public class Singleton{
    private static final Singleton INSTANCE =  new Singleton();
    //让构造函数为 private，这样该类就不会被实例化
    private Singleton(){}
    public static Singleton getInstace(){
        return INSTANCE;
```

```
    }
}
```

5.1.2 静态代码块（饿汉式）

```
public class Singleton{
    private static Singleton INSTANCE;
    private Singleton(){}
    static{
        INSTANCE = new Singleton();
    }
    public static Singleton getInstance(){
        return INSTANCE;
    }
}
```

5.1.3 双重检验double check（懒汉式）

```
public class Singleton{
    private static volatile Singleton INSTANCE;
    private Singleton(){}
    public static Singleton getInstance(){
        if(INSTANCE==null){
            synchronized(Singleton.class){
                if(INSTANCE == null){
                    INSTANCE = new Singleton();
                }
            }
        }
        return INSTANCE;
    }
}
```

5.1.4 静态内部类（懒汉式）

```
public class Singleton{
    private Singleton(){}
    private static class SingletonInstance(){
        private static final Singleton INSTANCE = new Singleton();
    }
    public static Singleton getInstance(){
        return SingletonInstance.INSTANCE;
    }
}
```

5.2 工厂模式