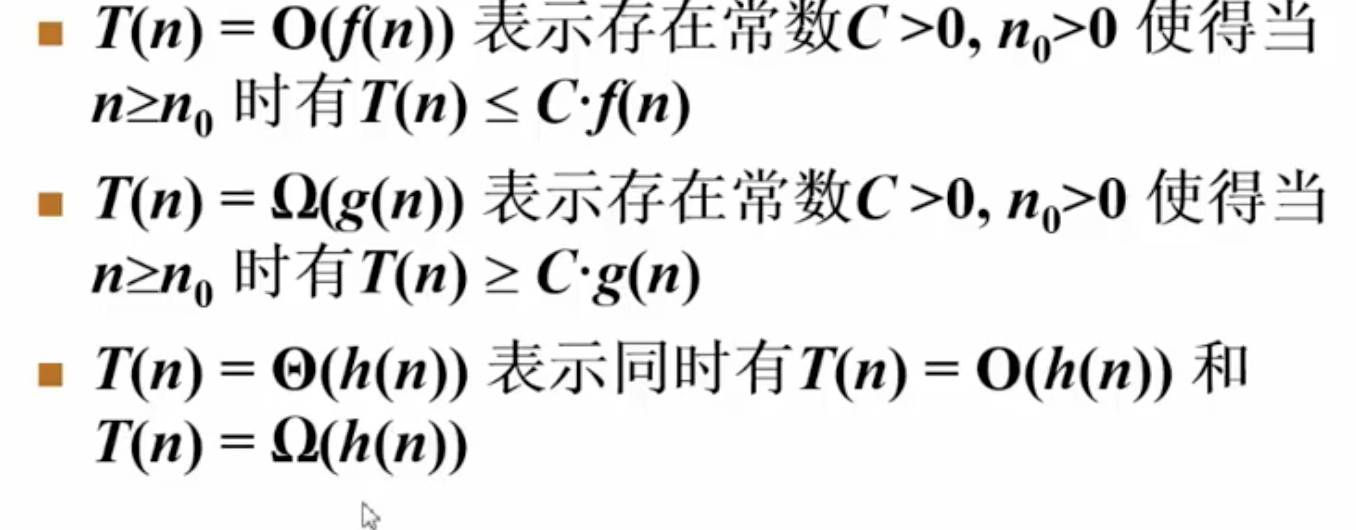
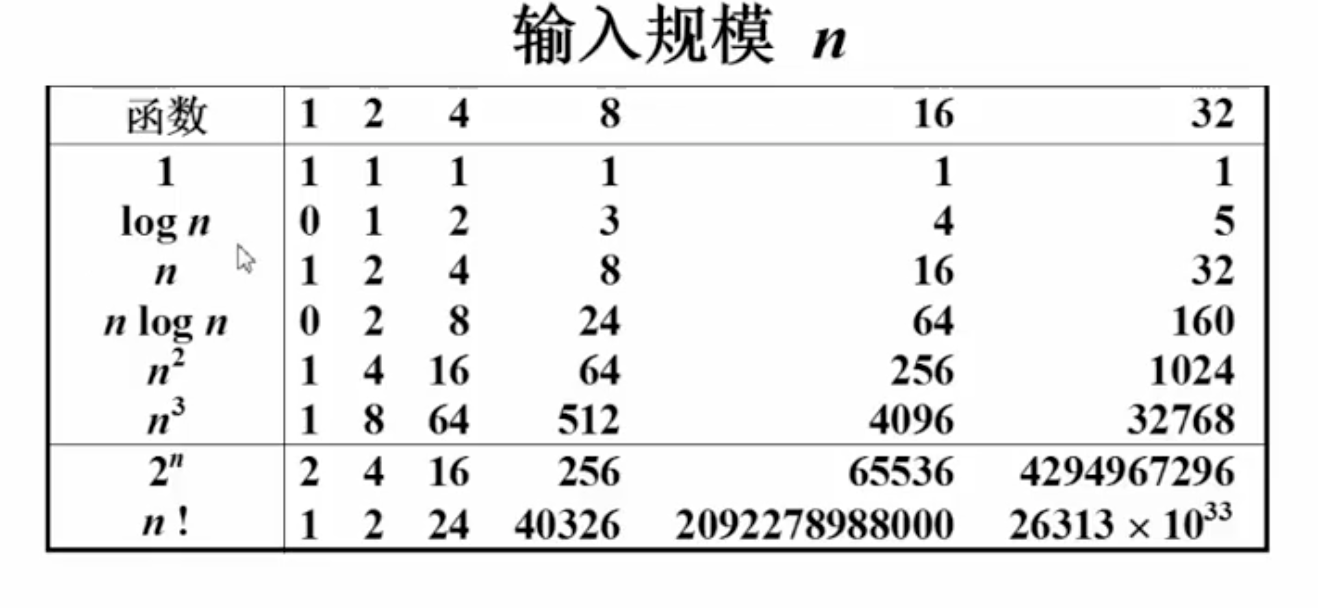
# 数据结构

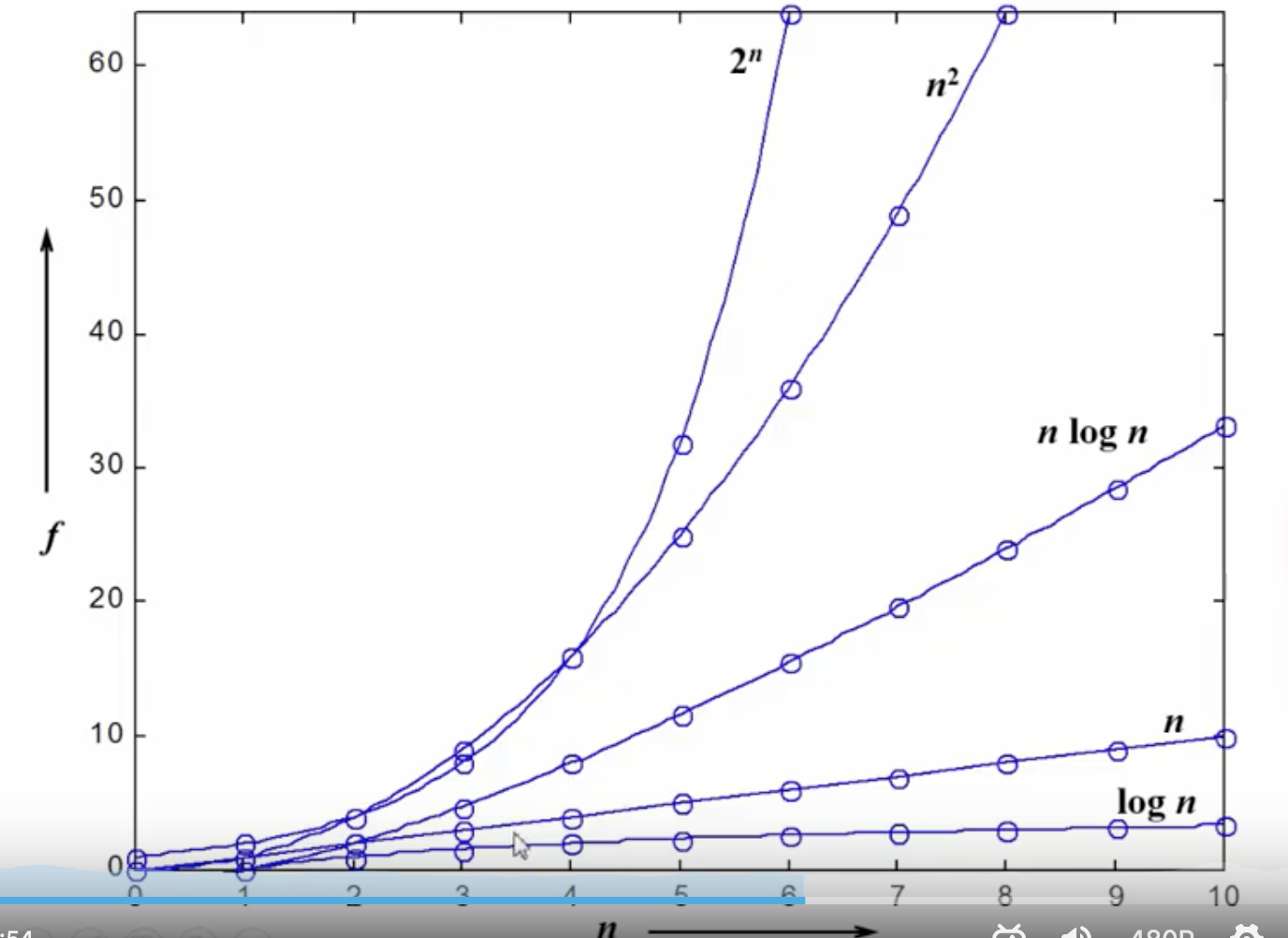
# 第1章 简介

### 1.2.2 什么是好算法

### 1.2.3 复杂度的渐进表示







## 1.3 应用实例

### 1.3.1 算法1&2

题目1

|  |
| --- |
| 例如：{-1, 3, -2, 4, -6, 1, 6, -1},求其中最大连续子列的和 |

算法1

|  |
| --- |
| */\*\*  \* 算法复杂度是T(N)=O(N^3)  \*  \** ***@param*** *arr  \** ***@param*** *n  \** ***@return*** *\*/* private static int maxSubseqSum1(int[] arr, int n) {  int thisSum = 0, maxSum = 0;  for (int i = 0; i < n; i++) { // i是子列左端位置  for (int j = i; j < n; j++) {// j是子列右端位置  thisSum = 0;  // 求子列和  for (int k = i; k <= j; k++) {  thisSum += arr[k];  }  if (thisSum > maxSum) {  maxSum = thisSum;  }  }  }  return maxSum; } |

算法2

|  |
| --- |
| */\*\*  \* 算法复杂度T(n)=O(n^2)  \** ***@param*** *arr  \** ***@param*** *n  \** ***@return*** *\*/* private static int maxSubseqSum2(int[] arr, int n) {  int thisSum = 0, maxSum = 0;  for (int i = 0; i < n; i++) { // i是子列左端位置  thisSum = 0;  for (int j = i; j < n; j++) {// j是子列右端位置  thisSum += arr[j];  if(thisSum > maxSum){  maxSum = thisSum;  }  }  }  return maxSum; } |

### 1.3.2 算法3 (分而治之)

|  |
| --- |
| */\*\*  \* 分而治之：时间复杂度T(N)=O(NlogN);  \** ***@param*** *arr  \** ***@param*** *n  \** ***@return*** *\*/* private static int maxSubseqSum3(int[] arr, int n) {  return *divideAndConquer*(arr, 0, n - 1); }  private static int divideAndConquer(int[] arr, int start, int end) {  int maxLeftSum = 0, maxRightSum = 0;  int maxLeftBoardSum = 0, maxRightBoardSum = 0;  int leftBoardSum = 0, rightBoardSum = 0;   // 递归终止条件  if (start == end) {  if (arr[start] > 0) {  return arr[start];  } else {  return 0;  }  }   int center = (start + end) / 2;   maxLeftSum = *divideAndConquer*(arr, start, center);  maxRightSum = *divideAndConquer*(arr, center + 1, end);   for (int i = center; i >= start; i--) {  leftBoardSum += arr[i];  if (leftBoardSum > maxLeftBoardSum) {  maxLeftBoardSum = leftBoardSum;  }  }    for (int i = center + 1; i <= end; i++) {  rightBoardSum += arr[i];  if (rightBoardSum > maxRightBoardSum) {  maxRightBoardSum = rightBoardSum;  }  }   return Math.*max*(maxLeftSum, Math.*max*(maxRightSum, maxRightBoardSum + maxLeftBoardSum)); } |

### 1.3.2 算法4

|  |
| --- |
| private static int maxSubseqSum4(int[] arr, int n) {  int thisSum = 0, maxSum = 0;  for (int i = 0; i < n; i++) { // i是子列左端位置  thisSum += arr[i];  if (thisSum > maxSum) {  maxSum = thisSum;  }  if (thisSum < 0) {  thisSum = 0;  }  }  return maxSum; } |

# 第2章 顺序、链式存储、堆、对列