1_complexity.pdf 与 2_list.pdf 详细内容总结

一、1_complexity.pdf：算法复杂度（Asymptotic Complexity）全解析

1. 核心渐近符号（Asymptotic Notations）定义

（1）大O符号（O(g(n))：上界符号）

- **非正式定义**：用于描述算法解决规模为n的问题所需时间T(n)的增长趋势，例如$T(n)=c\log_2 n$即T(n)是O(logn)。
- **正式定义**：O(g(n))是函数集合{f(n)}，存在常数c>0和足够大的n_0，当$n>n_0$时，满足`f(n) < c·g(n)`。也可通过极限表达：`lim_{n→∞} [f(n)/g(n)] ≤ c`。
- **核心含义**：g(n)是f(n)的**渐近上界**，描述算法的**最坏情况**行为，即f(n)的增长速率"不超过"g(n)。

（2）大Ω符号（Ω(g(n))：下界符号）

- **定义**：Ω(g(n))是函数集合{f(n)}，存在常数c>0和足够大的n_0，当$n>n_0$时，满足`f(n) > c·g(n)`。
- **核心含义**：g(n)是f(n)的**渐近下界**，即f(n)的增长速率"不低于"g(n)。

（3）大Θ符号（Θ(g(n))：紧界符号）

- **定义**：Θ(g(n)) = O(g(n)) ∩ Ω(g(n))，即同时满足上界和下界条件。
- **核心含义**：f(n)与g(n)的**增长速率相同**，是对算法复杂度最精确的描述。

2. 大O符号的典型示例与验证

（1）正面示例（可证明为O(g(n))）

函数f(n)	目标g(n)	验证过程（找c和n_0）	结论
2n + 10	n	取c=3，n_0=10：当n≥10时，2n+10 ≤3n	2n+10 = O(n)
7n - 2	n	取c=7，n_0=1：当n≥1时，7n-2 ≤7n	7n-2 = O(n)
$3n^3 + 20n^2 + 5$	n^3	取c=4，n_0=21：$3n^3+20n^2+5 \le 4n^3$（n≥21时成立）	$3n^3+20n^2+5 = O(n^3)$
3logn + 5	logn	取c=8，n_0=2：3log2+5=3+5=8 ≤8log2，n≥2时成立	3logn+5 = O(logn)
10000n + 5	n	取c=1，n_0=10000：10000n ≤1·n（n≥10000时成立）	10000n = O(n)

（2）负面示例（不可证明为O(g(n))）

- **问题**：n^2是否为O(n)?
- **推导**：假设存在c和n_0，使$n^2 \le c\cdot n$（$n\ge n_0$），即n ≤c。但n可无限增大，c为常数，无法满足不等式。
- **结论**：$n^2 \neq O(n)$，但$n^2 = O(n^2)$（取c=1，n_0=1即可）。

3. 渐近符号的关键性质（以大O为核心）

（1）基础性质

1. **忽略常数因子**：对任意k>0，$k\cdot f(n) = O(f(n))$。
   例：$5n^2 = O(n^2)$，$100\log n = O(\log n)$。
2. **高次幂主导低次幂**：若$0\le r\le s$，则$n^r = O(n^s)$。
   例：$n = O(n^2)$，$n^3 = O(n^5)$，但$n^2 \ne O(n)$。
3. **最快增长项主导和运算**：若$f(n) = O(g(n))$，则$f(n)+g(n) = O(g(n))$。
   例：$a\,n^4 + b\,n^3 = O(n^4)$（n^4增长快于n^3）；$\log n + n = O(n)$。
4. **多项式复杂度由最高次项决定**：若f(n)是d次多项式（$f(n)=a_d\,n^d+\dots+a_1n+a_0$），则$f(n) = O(n^d)$。

（2）进阶性质

1. **传递性**：若$f(n)=O(g(n))$且$g(n)=O(h(n))$，则$f(n)=O(h(n))$。
   例：$n=O(n\log n)$，$n\log n=O(n^2)$，故$n=O(n^2)$。
2. **乘积性质**：若$f(n)=O(g(n))$且$h(n)=O(r(n))$，则$f(n)\cdot h(n)=O(g(n)\cdot r(n))$。
   例：$n=O(n)$，$\log n=O(\log n)$，故$n\log n=O(n\log n)$。
3. **增长速率排序（核心结论）**：
   指数函数（b^n，$b>1$）> 多项式函数（n^k，$k\ge0$）> 对数函数（$\log_\beta n$，$b>1$）。
   - 例：$n^{20} = O(1.05^n)$（指数快于多项式）；$\log n = O(n^{0.5})$（对数慢于多项式）。
4. **对数函数等价性**：所有底数>1的对数增长速率相同，即$\log_\beta n = O(\log_d n)$（$b,d>1$）。
   例：$\log_2 n = O(\log_{10} n)$，因$\log_2 n = (\log_{10} n)/(\log_{10} 2)$，常数因子可忽略。

4. 算法复杂度的实际分析方法

（1）基础代码结构的复杂度

代码结构	复杂度	说明
简单语句序列（$S_1;S_2;\dots;S_k$）	$O(1)$	k为常数，执行次数固定
单循环（for(i=0;i<n;i++){S;}）	$O(n)$	循环n次，S为O(1)操作
嵌套循环（双层for循环）	$O(n^2)$	外层n次×内层n次，共n^2次操作
多层嵌套循环（k层）	$O(n^k)$	每层循环n次，总操作次数为n^k

（2）特殊循环结构的复杂度

1. **指数增长循环（以2倍增长为例）**：

```
h=1;
while(h <=n){
    S; // O(1)
    h=2*h;
}
```

   - **分析**：h取值为$1,2,4,\dots,2^k$，直到$2^k>n$，循环次数为$1+\log_2 n$。
   - **复杂度**：$O(\log n)$。

2. **内循环次数依赖外循环的嵌套循环**:

```
for(j=0;j<n;j++)
    for(k=0;k<j+1;k++){S;}
```

   - **分析**：内循环执行次数为1+2+...+n = n(n+1)/2（等差数列求和）。
   - **复杂度**：$O(n^2)$（忽略常数因子1/2）。

5. 算法效率分类与实例对比

（1）算法效率分类

- **多项式时间算法**：复杂度为$O(n^d)$（d为整数），被认为是"高效算法"，可在合理时间内解决大规模问题。
- **难解算法（Intractable）**：无已知多项式时间算法，通常复杂度为指数级（如$O(2^n)$），大规模问题无法在实际时间内求解。

（2）实例：硬件提速对问题规模的影响

- **场景**：程序P（$O(n^3)$）和程序Q（$O(3^n)$）均能在1小时内解决n=50的问题，硬件提速729倍后，可解决的问题规模为：
  1. 程序P（$O(n^3)$）：$n^3 = 50^3 \times 729 \rightarrow n = 50 \times \sqrt[3]{729} = 50\times9=450$。
  2. 程序Q（$O(3^n)$）：$3^n = 3^{50} \times 729 = 3^{50} \times 3^6 \rightarrow n=50+6=56$。
- **结论**：多项式算法随硬件提速收益显著，指数算法收益极小，凸显复杂度对算法实用性的决定性作用。

二、2_list.pdf：列表（List）数据结构全解析

1. 前置基础概念

（1）数据结构与算法的定义

- **数据结构**：计算机中存储和组织数据的特定方式，需适配应用场景（如检索、插入等需求），核心目标是"高效使用数据"。
- **算法**：解决问题的有限步、明确指令集合，可含随机输入，分为两类：
  - 决策过程：输出"是/否"（如判断一个数是否为素数）。
  - 计算过程：输出具体解决方案（如排序数组）。
  - 例：数学公式、计算机程序均为算法。

（2）抽象数据类型（ADT）

- **核心定义**：
  1. 数学模型：描述一类数据结构的共同行为（如"栈"的push/pop操作）。
  2. 接口导向：仅定义"能做什么"（操作及约束），不定义"怎么做"（实现细节）。
  3. 多实现性：同一ADT可通过不同数据结构实现（如"列表"可通过数组或链表实现）。
- **ADT与数据类型的区别**：
  - 数据类型：含"值集合+表示方式+操作"（如C语言的int类型）。
  - ADT：仅含"值集合+操作"，屏蔽表示方式（实现无关性）。
- **常见ADT**：列表（List）、栈（Stack）、队列（Queue）、集合（Set）等。

（3）ADT的核心优势

1. **模块化**：将程序拆分为独立功能单元，便于调试、维护和团队协作。
2. **可重用性**：同一ADT可在多个程序中复用，无需重复实现。
3. **实现透明性**：修改ADT的实现（如列表从数组改为链表）不影响上层使用，降低耦合。

2. 列表（List）ADT的定义与操作

（1）列表ADT的形式化定义

- **结构**：有序元素序列$A_1,A_2,...,A_n$（n为列表大小），满足：
  - 空列表：n=0。
  - 前驱/后继：A_{i-1}是A_i的前驱，A_{i+1}是A_i的后继（i=2..n-1）。
  - 首尾元素：A_1为“头”（head），A_n为“尾”（tail）。
  - 位置：A_i的位置为i。

（2）列表ADT的核心操作

操作名	功能描述
MakeEmpty	将列表置为空
PrintList	打印列表所有元素
Find	查找指定元素的位置，未找到返回特殊值
FindKth	查找第k个位置的元素
Insert	在指定位置插入元素
Delete	删除指定位置或指定值的元素
Next	返回指定元素的后继元素位置
Previous	返回指定元素的前驱元素位置

（3）操作示例（列表：34,12,52,16,12）

- Find(52) → 3（元素52在位置3）。
- Insert(20,4) → 34,12,52,20,16,12（位置4插入20）。
- Delete(52) → 34,12,20,16,12（删除元素52）。
- FindKth(3) → 20（位置3的元素为20）。

3. 列表的两种核心实现方式

（1）数组实现（静态实现）

① 数组的基本特性

- **定义**：固定大小、同类型元素存储于**连续内存**的静态数据结构，通过“索引”访问元素。
- **分类**：
  - 按维度：一维数组（单索引）、多维数组（多索引，如二维数组int a[3][4]）。

    - 按内存分配：静态数组（编译期分配内存，大小固定）、动态数组（运行期分配内存，大小可调整但需手动管理）。
- **C语言示例（一维静态数组）**：

```c
int b[5]; // 定义容量为5的int型数组
int b[5] = {19,68,12,45,72}; // 初始化数组
```

② 数组实现列表的操作细节

- **存储结构**：用数组存储元素，用变量`size`记录实际元素个数，`capacity`记录数组容量（size ≤ capacity）。
  例：数组`[2.3,7.1,0.3,9.5,0.0,...,0.0]`（capacity=10，size=4）表示列表[2.3,7.1,0.3,9.5]。

- **关键操作的实现逻辑**：

    1. **插入（Insert）**：
        - 无序列表：直接在`size`位置插入，`size++`（O(1)）。
        - 有序列表：先找到插入位置，将该位置后所有元素后移1位，再插入，`size++`（平均O(N)，需移动半数元素）。
    2. **删除（Delete）**：
        - 无序列表：用最后一个元素覆盖待删元素，`size--`（O(1)）。
        - 有序列表：删除元素后，将后续元素前移1位，`size--`（平均O(N)）。
    3. **查找（Find）**：线性搜索（遍历数组对比元素），O(N)。
    4. **FindKth**：直接通过索引访问（`array[k-1]`），O(1)。
    5. **PrintList**：遍历数组打印前`size`个元素，O(N)。

③ 数组实现的优缺点

- **优点**：随机访问效率高（O(1)），实现简单。
- **缺点**：
    - 大小固定：需提前预估容量，高估浪费内存、低估导致溢出。
    - 插入/删除效率低：有序列表中需移动大量元素（O(N)）。

（2）链表实现（动态实现）

① 链表的基本特性

- **定义**：由非连续节点组成的动态数据结构，每个节点含两部分：
    - 数据域（data）：存储元素值。
    - 指针域（next）：存储下一个节点的内存地址（尾节点指针为NULL）。
- **C语言节点定义（单链表）**：

```c
typedef struct _node{
    int data; // 数据域
    struct _node* next; // 指针域（指向同类型节点）
} Node;
```

② 单链表的核心操作实现

1. **创建节点**：用`malloc`分配内存，初始化数据域和指针域。

```
Node* newNode(int data, Node* next){
    Node* res = (Node*)malloc(sizeof(Node));
    res->data = data;
    res->next = next;
    return res;
}
```

2. **遍历（Traverse）**：从表头指针开始，沿`next`指针扫描至NULL，无随机访问能力。

```
void traverse(Node* L){
    Node* pt = L; // pt指向表头
    while(pt != NULL){
        printf("%d ", pt->data);
        pt = pt->next; // 移动到下一个节点
    }
}
```

3. **插入（Insert）**：
   - 头部插入（O(1)）：

   ```
   Node* insertHead(Node* L, int data){
       Node* X = newNode(data, NULL);
       X->next = L; // 新节点指向原表头
       L = X; // 表头更新为新节点
       return L;
   }
   ```

   - 尾部插入（O(N)，需扫描至尾节点）：

   ```
   void insertTail(Node* L, int data){
       if(L == NULL){ // 空链表特殊处理
           L = newNode(data, NULL);
           return;
       }
       Node* p = L;
       while(p->next != NULL){ // 找尾节点（p->next=NULL）
           p = p->next;
       }
   ```

```
    p->next = newNode(data, NULL); // 尾节点指向新节点
}
```

4. **删除（Delete）**：
   - 头部删除（O(1)）：

```
Node* deleteHead(Node* L){
    if(L == NULL) return NULL;
    Node* temp = L;
    L = L->next; // 表头指向后继节点
    free(temp); // 释放原表头内存
    return L;
}
```

   - 中间删除（O(N)，需找待删节点的前驱）：

```
Node* deleteMiddle(Node* L, int x){
    Node* current = L;
    Node* previous = NULL;
    // 找待删节点current及前驱previous
    while(current != NULL && current->data != x){
        previous = current;
        current = current->next;
    }
    if(current == NULL) return L; // 未找到
    if(previous == NULL) return deleteHead(L); // 待删节点为表头
    previous->next = current->next; // 前驱指向后继，跳过待删节点
    free(current);
    return L;
}
```

③ 双链表（扩展实现）

- **定义**：节点额外含前驱指针（`previous`），支持双向遍历和高效的中间插入/删除。

```
typedef struct _doubleNode{
    struct _doubleNode* previous; // 前驱指针
    int data;
    struct _doubleNode* next; // 后继指针
} DoubleNode;
```

- **中间删除优势**：已知待删节点`current`时，无需找前驱，直接修改指针（O(1)）：

```
void deleteDoubleNode(DoubleNode* current){
    current->previous->next = current->next;
    current->next->previous = current->previous;
    free(current);
}
```

④ 链表实现的优缺点

- **优点**：
  - 动态伸缩：无需提前预估大小，可按需分配/释放内存。
  - 插入/删除高效：头部/尾部插入/删除（O(1)），已知前驱时中间操作（O(1)）。
  - 无需连续内存：避免数组的连续内存分配限制。
- **缺点**：
  - 无随机访问：访问第k个元素需遍历（O(N)）。
  - 空间开销：需额外存储指针域。

4. 两种实现方式的对比分析

对比维度	数组实现	链表实现
内存存储	连续内存	非连续内存
大小灵活性	静态，固定容量	动态，按需伸缩
访问效率	随机访问（O(1)）	顺序访问（O(N)）
插入/删除效率	有序列表O(N)，无序O(1)	头部O(1)，尾部/中间O(N)
空间开销	无额外开销（仅存元素）	有额外开销（存指针）
适用场景	随机访问频繁、大小固定	插入删除频繁、大小动态

5. 列表操作的时间复杂度总结

操作名	数组实现（有序/无序）	链表实现（单链表）
MakeEmpty	O(1)	O(1)
PrintList	O(N)	O(N)
Find	O(N)	O(N)
FindKth	O(1)	O(N)
Insert	有序O(N)，无序O(1)	头部O(1)，其他O(N)
Delete	有序O(N)，无序O(1)	头部O(1)，其他O(N)