算法与数据结构学习笔记

——udc577

第一部分：基础篇

01、基础中的基础

算法分析之中最基础的概念，如时间复杂度、空间复杂度以及它们的粗略表示。

一些约定：符号、类型、写法等的约定。

最基础的数据结构的表示：顺序表和链表。后续的数据结构和算法都应用它们，或者对其进行变体和延伸。

时间复杂度与空间复杂度分析

主要是大 O 表示法：略。

一些约定

MAX_NUM、MAX 等：一般表示数据结构里元素数的最大限制。

DT：即 Data Type，数据类型，在实际的源码中替换成满足 DT 所要求的特征的实际数据类型即可（C++则可以用模板技术）。

在不影响理解的前提下，过于简单的（伪）代码不再列举，多数代码也仅列出重点、难点。

顺序表（擅长快速随机访问）

可简单地认为是数组的变体，支持随机访问。

可与 std::vector 进行类比以深化学习。

数据结构以及最小操作接口以下所示：

```
Struct seq_list {
    Int MAX_NUM;
    Int n;
    DT *elements;
};
```

std::vector

vector::capacity()：调整之前

::max_size()：调整之后

vector::size()

|-- create_null_list(int m) {	vector 默认构造函数
| MAX_NUM = m; n = 0;	
| }	
|	
|-- bool is_empty(seq_list *);	vector::empty()
|	
|-- int locate(seq_list *l, DT x) {	None
| for_each(i : n) {	
| if (l->elements[i] == x)	
| return i;	
| }	
| }	
|	
|-- insert_prev, insert_post	
| ↓	
|-- insert(seq_list *l, int pos, DT x)：	vector::insert()
| 原 pos 及其后位置的元素均后移一位。	
|-- delete_by_value(seq_list *l, DT x);	None
`-- delete_by_pos(seq_list, int pos);	vector::erase()
这两个 delete 函数执行后，被删元素之后的元素均前移一位。	

插入移动数：	
|-- 最少：0。在尾部插入。O(1)。	vector::push_back()
|-- 最多：n。在头部插入。O(n)。	
`-- 平均：sum(i = 0, n, (n - i)*P_i)	// n - i 为待移动元素数，P_i 为在位置 i 插入的概率
= sum(i = 0, n, (n - i) / (n + 1))	// 注：已有 n 个元素，再插一个，则新元素有 n+1 个位置可选择，故 P_i = 1 / (n + 1)。
= (sum(i = 0, n, n) - sum(i = 0, n, i)) / (n + 1)	
= ((n + 1)n - n(n + 1) / 2) / (n + 1)	
= n / 2。随机插入。O(n)。	vector::insert()
总结：顺序表适合按下标进行随机访问的操作，不适合频繁的插入和删除。	

删除移动数：

|-- 最少：0。在尾部删除。O(1)。 ⟷ vector::pop_back()

|-- 最多：n-1。在头部删除。O(n)。

`-- 平均：sum(i = 0, n - 1, (n - 1 - i)*P_d)

= sum(i = 0, n - 1, (n - 1 - i) / n)

= (n - 1) / 2。随机删除。O(n)。

// n - 1 - i 为待移动元素数，P_d 为删除第 i 个元素的概率

// 注：每个元素被删的可能性一样，则有 n 种可选的待删元素，故 P_d = 1 / n。

搜索比较数：

|-- 最少：1。搜索首元素。O(1)。

|-- 最多：n。搜索尾元素。O(n)。

`-- 平均：sum(i = 0, n - 1, (i + 1)*P_s)

= sum(i = 0, n - 1, (i + 1) / n)

= (n + 1) / 2。随机访问。O(n)。

若表中元素有序，则可用二分法，加速为 O($\log_2$n)。

链表（擅长频繁快速插入和删除）

单链表、循环单链表：略。

双链表（部分版本的 std::list 用此实现）

```
typedef struct doubly_linked_list_node {
      DT info;
      dllnode *prev;
      dllnode *next;
} dllnode;

typedef struct doubly_linked_list {
      dllnode *head;
      dllnode *tail;
}dllist;
```

```
|-- create_empty_list();          <------------>  list 默认构造函数
|
|-- is_empty(dllist *);           <------------>  list::empty()
|
|-- locate(dllist *l, DT x);      <------------>  None
|   逐个比较，与顺序表相似，只是结点遍历方式不同。O(n)。
|-- insert_prev, insert_post
|                ↓
|-- insert(dllist *l, dllnode *pos, DT x) ;  <------------>  list::insert(pos, elem)
|   先查找值等于 pos->info 的结点,再调整其 prev 或 next 指向即可。   注：pos 是迭代器，下同
|-- delete_by_value(dllist *l, DT x);        <------------>  list::remove(value)
`-- delete_by_pos(dllist *l, dllnode *pos);  <------------>  list::erase(pos)
    先找到目标，调整其前后结点 prev 和 next 指向，再 free 本结点。
```

总结：

插入和删除都不需要移动结点，只需调整结点的指针即可，故时间复杂度为 O(1)。

搜索：同顺序表，为 O(n)。

缺点：存储密度小，每个结点需要额外的指针，而顺序表无此问题。

02、栈和队列

栈（元素后进先出，Last In First Out，LIFO）<———> std::stack

顺序表示：

```
seq_stack {
    int MAX_NUM;
    int top;
    DT *elements;
};
```

链接表示：

```
stack_node {
    DT info;
    stack_node *prev;
};
linked_stack {
    stack_node *top;
};
```

|-- push(stack *s, DT x)：入栈（插栈顶）。

|　顺序：top++; elem[top] = x;　　链接：malloc() for x; 调整 top 指向;

|-- pop(stack *s)：出栈（减栈顶）。

|　顺序：top--;　　链接：调整 top 指向并释放无用结点。

`-- top(stack *s)：取栈顶元素，栈本身无任何变化。

顺序：NULL or elem[top];　　链接：top->info;

总结：两种表示的各项操作时间复杂度均是 O(1)，但由于顺序结构预先分配好内存，故实际耗时应优于链接表示（除非空间用尽需要重新调整）。

递归转非递归算法可用栈来辅助。

队列（元素先进先出，First In First Out，FIFO）⟷ std::queue

顺序表示：

```
seq_queue {
    int MAX_NUM, n;
    Int head, tail;
    DT *elements;
};
```

链接表示：

```
// queue_node 结构同 stack_node，prev 变 next
linked_queue {
    queue_node *head;
    queue_node *tail;
};
```

|-- push(queue *q, DT x)：入队（插队尾）

顺序：
```
|   顺序：if (is_full()) error();
|         tail = (tail + 1) % MAX_NUM;
|         elem[tail] = x;
|         n++;
|
```

链接：
```
链接：if (is_empty())
          head = x_ptr;
      else
          tail->next = x_ptr;
      tail = x_ptr;
```

|-- pop(queue *q)：出队（删队头）

顺序：
```
|   顺序：if (is_empty()) return;
|         head = (head + 1) % MAX_NUM;
|         n--;
|
|
|
```

链接：
```
链接：if (is_empty()) return;
      p = head;
      head = p->next;
      free(p);
```

|-- front(queue *q)：取队头元素，队列本身不变。

| 顺序：if (is_empty()) return NULL; 链接：if (is_empty()) return NULL;

| return elem[head]; return head->info;

|-- is_empty()：判断队列是否为空。

| 顺序：return (0 == n); 链接：return (NULL == head);

`-- is_full()：判断队列是否已满（仅限于顺序结构）

顺序：return (n >= MAX_NUM); 链接：/

注：实际应用中，应加入动态调整容量的操作，使之不会满。

总结：两种表示的各项操作时间复杂度均是 O(1)，但由于顺序结构预先分配好内存，故实际耗时应优于链接表示（除非空间用尽需要重新调整）。

广度优先搜索算法可用队列来辅助。

03、字符串操作（重点介绍匹配操作）

顺序表示：

```
seq_string {
    int MAX_NUM, n;
    char *c;
};
```

链接表示：

```
string_node {
    char c;
    string_node *next;
};
typedef string_node* linked_string;
```

注：链接表示的存储密度低，且按下标访问字符效率低，故不推荐。以下操作只用顺序结构。

|-- // 其它操作

`-- int index(string main/* or target */, string sub/* or pattern */)：查找子串 sub（或模式 pattern）在主串 main（或目标串 target）中首次出现的位置，又叫模式匹配。

朴素模式匹配：逐个且有回溯的匹配，暴力模式，时间复杂度为 O(m x n)。

无回溯的模式匹配：以 KMP 算法为例：在比较 p[i]与 t[j]不等后，将 p 串右移若干位，并用新字符 p[k]（显示 k<i）和 t[j]（甚至 t[j+1]）进行比较，这个 k 值正是加速的关键。经验证，k 值不仅存在，

而且仅依赖于模式串 p 本身，与目标串 t 无关，每个 p[i]的 k 值一般不同，它们的值可组成一个数组，可称之为 next 数组，即表示当匹配不等时，“下次”应偏移的字符量。KMP 算法主要由利用 next 数组进行匹配和构造 next 数组两大部分组成。

KMP 算法：利用 next 数组进行匹配

```
int kmp_match(string t, string p, int *next) {
    int i = 0, j = 0;

    while (i < p->n && j < t->n) {
        if (-1 == i || p->c[i] == t->c[j]) {
            i++; j++;
            continue;
        }
        i = next[i];
    }

    if (i >= p->n)
        return j - p->n + 1;

    return -1;
}
```

注：考虑到“或”逻辑的短路操作，-1 == i 必须放在前面，否则当 i 等于-1 会导致数组越界。

j 不变，i 减小，即 p 右移。若 next[i]为-1，则表示（并导致）（应）重新用 p[0]进行比较，且是右移一位与 t[j+1]比较。

循环中 j 只增不减，故时间复杂度为 O(n)。next 数组的求解见后面。

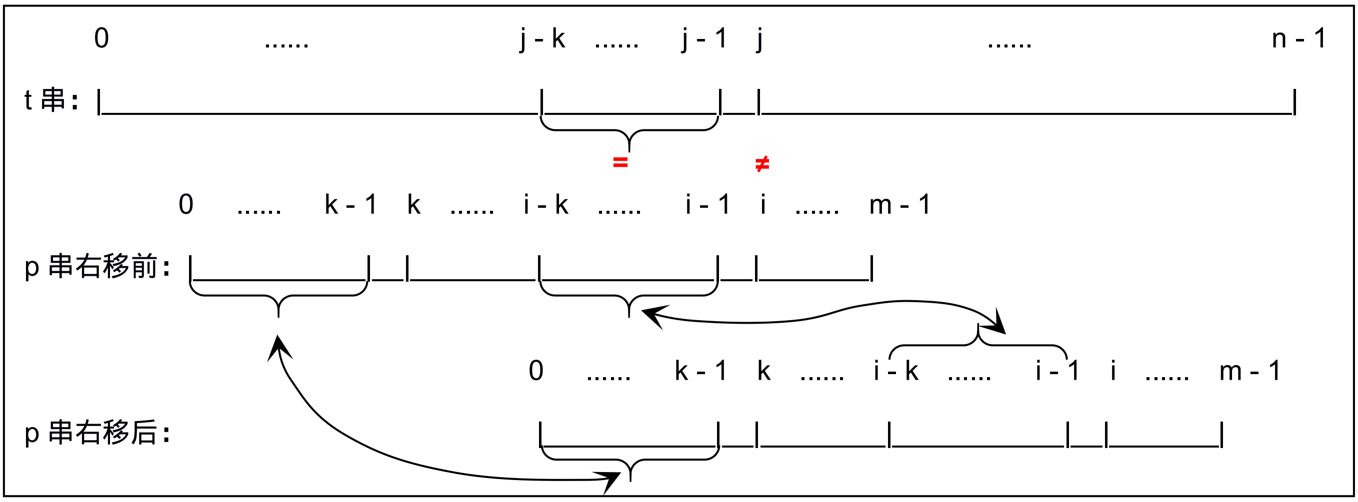
KMP 算法：构造 next 数组

假设比较操作进行到 p[i]与 t[j]的对比时发现不等，且假设需要将 p 右移若干位让 p[k]（0 ≤ k ≤ i-1）与 t[j]重新进行比较即可，则可推导出右移量为 i-k，且 p[0]到 p[k-1]这一段是不必重复比较的，与 t[j-k]至 t[j-1]段相等，而 t 串的这一段又与 p 串右移前的 p[i-k]至 p[i-1]（令 i-1-x+1=k，则 x=i-k 为该段起点）相同，最后得出 p[0]至 p[i-1]存在一个相同的最大前后缀（不包括 p[0]至 p[i-1]本身，但允许空串），这个前后缀的长度为 k，则 next[i]=k（这种取值方法还可改进，见后面）。从这里可以看出，next[i]只与 p 串有关，通过在 p 串内部进行头尾子串的匹配即可得到 k，这实际上也是一个小范围的模式匹配问题，求 k 值实质上是找最大相同前后缀的长度。

特别地，next[0]总为-1，因为若 p[0]与 t[j]比较不等时，只能让 p[0]与 t[j+1]继续比较，不存在一

个 k 值可让 p[k]继续与 t[j]比较，也可理解为 p[-1]与 t[j]比较，但无实际意义，不过却可导致 p[0]与 t[j+1]对齐，即 p 串右移一位继续从头比较。

论证最大相同前后缀的示意图如下：



代码实现：

```
void make_next_array(string p, int *next) {
    int i = 0, k = -1;


    next[i] = k; //  初始化 next[0]

    while (i < p->n - 1) {
        while (k >= 0 && p->c[i] != p->c[k]) {

            k = next[k]; //  其实与 k--类似

        }
        i++; k++;
        next[i] = (p->c[i] != p->c[k]) ? k : next[k];
    }
}
```

代码分析如下：

k 值的含义是关键！它最初的含义是当 p[i]与 t[j]不等时 p 串向右移动所产生的一个可继续与 t[j]比较的字符的下标值（-1 则表示 p[0]与 t[j+1]比较），**可能取** p[0] ~ p[i-1]的**最大相同前后缀长度值**，**也可能不取**，不过 k 越小，p 串向右偏移越大，匹配速度越快，但也要建立在无漏匹配的前提下。

最初 next[i] = k < i，故当 k ≥ 0 时，next[k] < k，

$$\text{又有}\begin{cases}\text{p[k] = p[i]时：p[k]没必要与 t[j]比较} \\ \text{p[k] != p[i]时：p[k]必须与 t[j]比较}\end{cases}\text{，则 next[i] = }\begin{cases}\text{next[k]} \\ \text{k}\end{cases}\text{。}$$

代码思路是利用前面已求出的 next[0]至 next[i]值，来求 next[i+1]，而且假定先取最大相同前后缀长度值，则 p[0]至 p[i+1]对应的 p[0]至 p[i]子串的最大相同前后缀，肯定比 p[0]至 p[i]对应的 p[0]至 p[i-1]子串的最大相同前后缀至少长一位，所以先在内层循环里令 k 取 next[k]，跳出循环之后再加 1，得到新 k 值后，再进一步比较 p[i]和 p[k]（下标是新 k 值）是否相等，最后决定是否可采用值更小的 next[k]（同样，下标是新 k 值）。

时间复杂度分析：由于有两层循环，很容易误解为 $O(m^2)$，但实际上外层循环的 k++最多执行 m-1 次，而且 k 不能小于-1，所以推断出内层循环的 k 值递减也不可能超过 m-1 次（是总的次数，不是每轮外层循环），因此总的循环次数最大是 m-1 次，则时间复杂度为 O(m)。与前面利用 next 数组匹配过程的时间复杂度结合，总的 KMP 算法时间复杂度是 O(m+n)。并进一步可推断，当 n>>m（意为 n 远大于 m）时，总的时间复杂度实际上趋近 o(n)，且 next 数组只需构建一次而多次使用，对性能的提升是非常巨大的。但当 n 与 m 非常接近，则有回溯的朴素匹配算法可能更省时间（构造 next 数组是有耗时的）。

04、树与二叉树

重要概念

层数/层次（level）：规定**根结点的层次为 0**（与数组的起点相同。但也可规定起点为 1，只是后续依赖于此的计算表达式有所不同而已），**其余结点的层数则在其父结点的层数加 1**。

深度（depth）/高度（height）：**树中的最大结点层数**叫树的深度/高度。

边：相邻结点之间的连线。

度：结点的**非空子树的个数**为该结点的度数，树中的最大结点度数为该树的度数。

路径及**路径长度**：起始结点到终止结点的连线，或者结点序列，叫路径。相邻层次两个结点之间的边数之和，或者结点序列数减一，就是路径长度。

叶结点及**分支结点**：无子结点（即其下再无分支延伸出去）的结点叫叶结点，类似于实际中树叶的概念，有时也叫外部结点。有子结点的结点叫分支结点，有时也叫内部结点。

斜树：结点单侧生长的树，跟一个链表的形象相似。

满二叉树：无统一定义。此处约定非叶结点都有左右子树，且所有叶结点都在同一层（底层）。

一定是 ↓↑ 不一定是

完全二叉树：满二叉树从底往上、从右往左依次逐个去掉结点的每个形态即是完全二叉树。

数据结构

树的表示法之一：长子右兄弟表示法：

```
struct tree {
    DT data;
    struct tree *first_child, *right_sibling;
    // 如有必要可再增加 parent 结点指针以快速查找父结点
};
```

优点：可以很容易地将复杂的树转化为简单的二叉树。

其余表示法以及树的各种运算、接口，见进阶篇中树的章节详细介绍，下面重点介绍二叉树。

二叉树的链接表示：每个结点最多有两个子结点，可完全列举，如下：

```
struct binary_tree {
    DT data;
    binary_tree *left_child, *right_child;
}; // 与树的长子右兄弟表示法结构上相同。
```

二叉树的主要性质

性质 1：第 i 层最多有 2^i 个结点。

性质 2：深度为 k 的二叉树最多有 $\sum_{i=0}^{k} 2^i = 2^{k+1}-1$ 个结点。

性质 3：叶结点个数 n_0 **=** 度为 2 的结点个数 n_2 **+ 1**。证明如下：

等式 1：$N = n_0 + n_1 + n_2$；其中 N 是二叉树的结点总数，n1 显然是度为 1 的结点个数

等式 2：$B = N - 1$；其中 B 是二叉树的总边数

等式 3：$B = n_1 + n_2$；只有有度数的结点才能向下延伸出边

三式结合即可证明。

性质 4：n 个结点的**完全二叉树**的**深度为$\lfloor \log_2 n \rfloor$**（$\lfloor x \rfloor$即不大于 x 的最大整数）。利用性质 2、对数转换和不等式性质即可证明。

性质 5：描述**完全二叉树**的**父结点与子结点下标的关系**。即若从上到下、从左到右的顺序对完全二叉树所有结点从 0 到 n-1 进行编号，则：

（1）子找父：无父结点（当 $i = 0$ 时）；或父结点的下标为$\lfloor (i - 1) / 2 \rfloor$（当 $i > 0$ 时）。

（2）父找子：左子下标为 $2i + 1$，右子下标为 $2i + 2$，若子下标大于 $n - 1$，则无该子。

性质 6：满二叉树中，叶结点个数比分支结点个数多 1。实质是性质 3 的特例。

性质 7：扩充二叉树（扩充成满二叉树）中，外部结点数（叶结点数）比内部结点数（分支结点数）多 1。可根据性质 6 证明。

性质 8：扩充二叉树的外部路径长度 E 和内部路径长度 I 满足 $\mathbf{E = I + 2n}$，其中 n 是内部结点数。

二叉树的遍历

先根（前序/前缀）遍历

……

中根（中序/中缀）遍历

……

后根（后序/后缀）遍历

……

逐层遍历

……

小结

已知前序序列和中序序列，或已知后序序列和中序序列，即知道前中后序之中相邻的两种序列，可唯一确定一棵二叉树。

线索二叉树

将原结构的 left_child 和 right_child 分别重新命名为 left_node 和 right_node，并让空结点指向前驱或后继，以及在结点中加多一个 left_tag 和 right_tag 分别表示结点（空结点和非空结点）指向的是左（left_tag=0 时）、右（right_tag=0 时）孩子，还是指向前驱（left_tag=1 时）、后继（right_tag=1），则成了线索二叉树。线索二叉树的引入是为了充分利用结点资源，和加速遍历过程。

代码实现：……

哈夫曼树和哈夫曼编码

叶子带权重值，带权路径长度（Weighted Path Length，WPL）最小的二叉树叫哈夫曼树或最优二叉树。主要用于压缩数据。

构造思路：先排好序，权值最小的两个结点作为最初的叶结点，小左右大，并为它们加上父结点，父结点值为它们值之和。然后，将这个父结点与下一个待处理结点以相同的方法重复即可。

哈夫曼编码：与哈夫曼树在权重表示方面有一点区别，暂略。

哈夫曼树和哈夫曼编码代码实现：……

二叉树和树的转换

见进阶篇。

05、图

概念

图：描述多对多关系的一种数据结构，逻辑上用集合来表示。G(V, E)：V 即 Vertex（顶点），

表示一系列顶点的集合，必须有穷且非空；E 即 Edge（边），表示顶点之间的联系，可以为空；G 即 Graph，是由顶点和顶点之间关系组成的集合。

无向边：无方向的边，用圆括号表示，例如(v1, v2)。

有向边：有方向的边，又叫弧（arc），用尖括号，例如<v1, v2>从 v1 指向 v2，v1 叫弧尾，v2 叫弧头（箭头所在之处）。

简单图：边不重复，且不存在某个顶点到其自身的边。**与之相对的是复杂图**。

无向完全图：任意两点之间都有边的无向图，边数为 $C_n^2 = n(n-1)/2$。

有向完全图：任意两点之间都有方向相反的两条弧的有向图，边数为无向完全图的 2 倍，即 $n(n-1)$。

稀疏图与**稠密图**：相对概念，看 V 和 E 的多少。

网：边带权重的图。

顶点的度：与该顶点连接的边的数目。如果是有向图，还区分入度和出度。

连通图：任意两点间有直接或间接路径的无向图。

连通分量：无向图中的**极大**（不是最大）连通子图。

强连通图：任意两点间都有直接或间接双向路径的有向图。

强连通分量：有向图的**极大**（不是最大）强连通图。

生成树：包含一个图的所有 N 个顶点，但只有足以构成一棵树的 N-1 条边。从这点也可推导出 N-1 条边是该图有无环的边界条件。

数据结构

领接矩阵（Adjacency Matrix）表示：

```
struct graph {
    VertexType vexs[MAX_NUM];
    EdgeType arcs[MAX_NUM][MAX_NUM];
    int vertex_num, edge_num;
};
```

不足：对边数少的情况会造成较大的资源浪费。

邻接表（Adjacency List）表示（结合数组和链表）：

```
struct EdgeNode {
    int adjvex_index;
    EdgeType weight;
    struct EdgeNode *next;
};

struct VertexNode {

    VertexType data;
    EdgeNode *first_edge;
};

struct graph {
    VertexNode adj_list[MAX_NUM];
    int vertex_num, edge_num;
};
```

十字链表表示：针对有向图，在邻接表基础上，同时存储入度和出度的信息。

邻接多重表表示：针对无向图，在邻接表基础上，让边结点用两个结点表示（？？）。

边集数组表示：与邻接矩阵的不同只是边信息用一个一维数组表示，仅列出存在的边，无空间浪费。

遍历方式

深度优先遍历（Depth First Search，DFS）：沿同一方向搜到底。

广度优先遍历（Breadth First Search，BFS）：同一结点所有方向都搜一次，再搜下一结点。

最小生成树算法

生成树定义见前面。

Prim 算法：用于邻接矩阵，以顶点为思路。

Kruskal 算法：用于边集数组，以边为思路。

最短路径（仅讨论针对有权重的边的图，即网）

Dijkstra 算法：按路径长度递增的次序生成最短路径的算法。

Flod 算法：适用于求所有顶点到所有顶点的最短路径的情况。

拓扑排序

AOV 网：Activity On Vertex Network，是一个用顶点来表示活动的网，这是一个无环有向图。

拓扑序列：v_i 到 v_j 的某条路径叫做一条拓扑序列。

拓扑排序即是对一个有向图构造拓扑序列的过程。

构造时，若全部顶点都输出，则是 AOV 网（无环），否则不是 AOV（有环）。

算法思路：从 AOV 网选一个入度为 0 的顶点，输出并删除，再删除以此顶点为尾的弧，重复此步骤，直至输出所有顶点，或不存在入度为 0 的顶点为止。

关键路径

AOE 网：Activity On Edge Network，是一个用边的权值表示活动的持续时间的网。

从源点到汇点（终点）的最长路径叫关键路径。

算法思路：找所有活动的最早开始时间和最晚开始时间，若两者相等，则此活动是关键活动，活动间的路径是关键路径（？？）。

06、查找

顺序查找/线性查找

用 for 循环或 while 循环，逐个比较。代码实现略。

注意 for (i = 0; i <= n; i++)每次循环都需要判断下标是否越界，有一定消耗，可优化为在待查找数组预留一个位置填入“哨兵”值 sentry，然后用 while 循环：while (a[i] != sentry)。但这样做，可能需要改动待查找数组的内存分配，以便增加一个位置放哨兵。可考虑另外设置一个临时变量，把数组首元素或尾元素先换出来，查找结束之后再放回去。

顺序查找适用于小型无序数据集，时间复杂度为 O(n)。

折半查找/二分法查找

属于有序表查找技术之一。针对的是有序的数据集，时间复杂度为 $O(\log_2 n)$。

算法的核心是取中间元素的操作：

mid = (low + high) / 2 = low + (1 / 2) (high - low)

然后根据本次比较的情况，令 high = mid - 1（当 key < a[mid]时）或 low = mid + 1（当 key > a[mid]时），去算下一轮的 mid 值，再进行比较，依此重复。

插值查找

折半查找的引申。取中间元素的操作公式为：

mid = low + (**(key - a[low]) / (a[high] - a[low])**)(high - low)

适用于分布比较均匀的数据集，这种情况下要比折半查找快。但对于极端不均匀的数据集，例如 { 0, 1, 2, 2000, 2001, 999998, 999999 }，则不一定适合。

斐波那契（Fibonacci）查找

利用斐波那契数组生成 high 和 low 与 mid 比较，平均性能比折半查找要好。

线性索引查找

稠密索引查找：索引与记录是一对一关系，适用于小量数据。

分块索引查找：数据分块，块内无序，块间有序。时间复杂度为 $O(\sqrt{n})$（分析待补充）。

倒排索引查找：搜索引擎的做法。单词作为索引，文章和链接等内容作为记录。

二叉排序树查找

动态查找表：查找时会有增删操作的表。

二叉排序树：记录的值大小遵循 左子树 < 根 < 右子树。可看出用的是中序遍历来提高查找和增删的效率。

算法：查找、插入、删除。

删除：删叶结点（直接删），或删只有左子树或右子树的结点（子承父业），或删同时有左右子树的结点（用它的前驱或后继结点直接替换）。

查找效率与二叉树的高度有关，所以应该构造成比较平衡。

↓ 引申出

平衡二叉树（即 AVL 树）

AVL 树是以发明者的首字母命名的一种二叉树。这种二叉树的左、右子树高度差不超过 1，这个高度差也叫平衡因子 BF（Balance Factor），显然 BF = -1, 0 或 1。

最小不平衡子树：距插入结点最近，且 BF 大于 1 的结点为根的子树。

构造平衡二叉树：插入新结点时先检查是否破坏平衡，若是则找出最小不平衡子树，并**旋转**使之平衡。（示例待补充）

如何旋转：BF 为负则左旋，为正则右旋，这是在最小不平衡子树的根结点与子结点的 BF 符号一致的前提下，若不一致，须先统一再进行旋转。（代码实现待补充）

多路查找树（B 树、B+树等）

专为外存查找而设计，一个结点可存多个 key，可带多个子结点。

2-3 树：每个结点有 0、2 或 3 个孩子。一个 2 结点包含 1 个 key 和 0 或 2 个孩子，且孩子与本结点的大小关系与二叉排序树的一样。一个 3 结点包含一小一大 2 个 key 和 0 或 3 个孩子，且左孩子 < 较小 key < 中间孩子 < 较大 key < 右孩子。

2-3-4 树：2-3 树的扩充。

B 树：一种平衡的多路查找树，以上两种均是它的特例。

m 阶 B 树有如下属性：

（1）若根结点不是叶子，则至少有 2 棵子树。

（2）所有叶子位于同一层次。

（3）每个非根结点均有 k-1 个元素，其中 $\lceil m/2 \rceil$ ≤ k ≤ m，若是非叶结点则还要有 k 个孩子。（$\lceil m/2 \rceil$这个下限怎样推导出？不是因为其它条件而得出，而是要满足这个条件才是 B 树的定义，故可推导出 3 阶 B 树只有 2、3 子结点两种情况，即 2-3 树，4 阶则有 2、3、4 子结点三种情况，即 2-3-4 树，依此类推，阶数越高，子结点类型越多。特别地，没有 2 阶 B 树，因为按定义，2 阶 B 树会出现 1 或 2 个子结点，显然 1 并不是多路）

（4）所有分支结点包含以下数据：{ $n, A_0, K_1, \ldots, A_{n-2}, K_{n-1}, A_{n-1}, K_n, A_n$ }，其中 n 是关键字个数（$\lceil m/2 \rceil$ - 1 ≤ n ≤ m - 1），K_i 是关键字（即 key），A_i 是子树，且 $K_i < K_{i+1}$，A_{i-1} 子树所有结点关键字均小于 K_i，A_i 则大于 K_i。

查找最坏效率：$k \le \log_{\lceil m/2 \rceil}((n + 1) / 2) + 1$

↓ 为解决所有元素遍历的问题、针对文件系统的需求而改造出的数据结构

B+树：B 树的改进版。有 n 棵子树的结点，包含 n 个关键字，且所有关键字在叶子也存一份，叶子之间也有连线，这样遍历时就不必再回溯上层的分支结点。

所有分支结点可看成索引，叶子则记录实际信息。

红黑树

实质上是 2-3 树的简化实现。待补充。

07、排序

主要介绍内排序。如下：

	简单排序	复杂排序		
交换排序	**冒泡排序**	**快速排序**		
选择排序	**简单选择排序**	堆排序		
插入排序	**直接插入排序**	希尔（Shell）排序	二分法插入排序	表插入排序
归并排序		归并排序		
分配排序		基数排序		

其中：简单排序是指思路和代码实现都很简单、直观的排序算法，复杂排序则往往是简单排序的改进版，思路会比较绕、取巧，代码也会比较复杂。有底色的排序是重点中的重点，一定要掌握。

另外要注意的是：交换排序和选择排序，都是通过交换记录来进行的，没有大规模的记录移动；而直接插入排序则除了在合适位置插入目标记录，目标记录后面的记录都要向后挪一位。

所有排序代码均以**整型数组**按不递减（即**从小到大**）的顺序为例进行设计。代码中 SWAP_ARRAY_ELEMENT()为交换数组中两个不同下标元素的操作，可实现为宏，也可实现为（内联）函数。时间和空间复杂度分析中，“比较”主要考虑数组元素之间或数组元素下标之间的比较；“赋值”主要考虑数组元素之间或数组元素与外部变量之间的赋值，“交换”亦是，且一次完整的交换等于 3 次赋值，为了更精细，一般只看赋值次数。

冒泡排序（必会）

Bubble Sort。两两比较相邻记录，若反序则交换，直至无反序的记录为止。类似炒股的短线操作，不断地买进买出（即不断地交换）。时间复杂度如下：

最好：有序，n-1 次比较，无交换，O(n)。

最坏：逆序，$\sum_{i=0}^{n-2}(n-1-i)$ = n(n - 1) / 2 次比较，及 3 倍的赋值操作，O(n^2)。

代码实现：

```
void bubble_sort(int *a, int n)
{
    bool no_swap; // 根据是否有元素交换的特点来决定外层循环是否可提前结束的标志
    for (int i = 0; i < n - 1 && !no_swap; ++i)
    {
        no_swap = true;
        for (int j = n - 1; j > i; --j)
        {
            if (a[j - 1] > a[j])
            {
                SWAP_ARRAY_ELEMENT(a, j - 1, j);
                no_swap = false;
            }
        }
    }
}
```

快速排序（必会）

Quick Sort。20 世界十大算法之一，冒泡排序的改进版。

通过一趟排序将记录分成 2 部分，其中一部分的记录均比另一部分的要小，则可分别对这两部分进行排序，最后得到有序的整个序列。

最好、平均时间复杂度：$O(n\log_2 n)$，最坏时间复杂度：$O(n^2)$，此时正序或逆序。

属不稳定排序。

简单选择排序（必会）

Simple Selection Sort。通过 n - i 次比较，从 n - i 个记录中选出最小的记录，和第 i - 1（1 ≤ i < n）个记录交换。相对于冒泡排序，它是找到合适的位置再交换，否则不交换，所以在交换操作方面节省了时间。类似炒股中到时机非常明确才出手的高手。时间复杂度如下：

最好：$\sum_{i=0}^{n-2}((n-1-i)+1)$ = (n + 2)(n - 1) / 2 次比较，0 次赋值，$O(n^2)$。

最坏：比较次数不变，赋值次数则为 3(n-1)次，$O(n^2)$。

因赋值次数较少，故优于冒泡排序。这算是从另一个角度来优化冒泡排序。

代码实现：

```
void simple_selection_sort(int *a, int n)
{
	for (int i = 0; i < n - 1; i++)
	{
		int min = i;

		for (int j = n - 1; j > i; j--)
		{
			if (a[j] < a[min])
				min = j;
		}

		if (i != min)
			SWAP_ARRAY_ELEMENT(a, i, min);
	}
}
```

堆排序

Heap Sort。对简单选择排序的改进。每一趟排序除了找出第 i 小（或大）的元素，也会对其他元素的位置进行调整，从而减少总的比较和移动次数。

堆：即某结点大于等于（大根堆）或小于等于（小根堆）左右结点的完全二叉树。

主要问题：如何构造堆。

最坏、最好、平均时间复杂度均为 $O(n\log_2 n)$，是不稳定排序，不适用于小集合，因为建堆的耗费相对较大。

直接插入排序（必会）

Straight Insertion Sort。将未排序记录逐个插入到已排序的子区间中，子区间逐渐增大直至包含所有记录，则排序完毕。类似从桌上逐张拿扑克牌到手中理顺。注意当新元素插入有序子区间的某位置时，该位置原记录及其后至 i-1 之前的记录需后移一位。**还要注意比较操作和移动（赋值）操作要同时进行**，若拆分开来，可能会导致比较次数较多时移动次数却较少，或比较次数较少时移动次数却

较多。时间复杂度如下：

最好：有序，n-1 次比较，无赋值，O(n)。

最坏：逆序，$\sum_{i=1}^{n-1}(1+i-1)$ = n(n - 1) / 2 次比较，$\sum_{i=1}^{n-1}(1+1+i-1+1)$ = (n + 4)(n - 1) / 2 次赋值，$O(n^2)$。

平均：比较和移动均是 n^2 / 4。

性能比冒泡和简单选择排序要好一些。

适用于基本有序或数据量小的集合。

代码实现（**TODO：看右边是否比左边更优？**）：

```
void straight_insertion_sort(int *a, int n)
{
	for (int i = 1; i < n; i++)
	{
		if (a[i] >= a[i - 1])			//  无此判断

			continue;

		int tmp = a[i];
		int j;					// int j = i - 1;

		a[i] = a[i - 1];			// not needed
		for (j = i - 2; j >= 0 && tmp < a[j]; j--)	// while (j >= 0 && tmp < a[j])
		{
			a[j + 1] = a[j];		//	a[j + 1] = a[j];	j–;
		}
		a[j + 1] = tmp;			// if (j != i - 1) a[j + 1] = tmp;
	}
}
```

希尔排序

Shell Sort。对直接插入排序的改进。分组，每组用直接插入排序，当整个序列**基本有序**时，再对全体进行一次直接插入排序。

分组有一定的策略，不能连续分割，而是将有一定距离的元素分到同一组，这样才能保证每组排序并合并后整个序列基本有序。

平均比较次数和平均移动次数的时间复杂度都是 $O(n^{1.3})$。

归并排序

Merging Sort。多次拆成 m 个子序列，各自排好后再按每 m 个子序列合并，直至最后合并成总序列。

以 2 路归并排序为例进行分析（待补充）。

最坏、最好、平均时间复杂度均为 $O(n\log_2 n)$，是稳定排序。空间复杂度为 $O(n + \log_2 n)$，故效率高却占内存。

小结

参考《大话数据结构》中各种排序的性能总结。

？？、参考资料

《大话数据结构》，程杰

《算法与数据结构——C 语言描述（第 2 版）》，张乃孝

《算法导论（原书第 3 版）》，Thomas 和 Ronald 等著，殷建平等译