20180414

❁ 散列表查找(哈希表)

散列技术是在记录的存储位置和它的关键字之间建立一个确定的关系 f 使得每个关键字 key 的存储位置=f(key)。查找时，根据 key 计算 f(key)，如果 key 存在集合中，必定在 f(key)位置。

对应关系 f 为散列函数，或哈希(Hash)函数。采用散列技术将记录存储在一块连续的存储空间，这块连续存储空间称为散列表或哈希表(Hash table)。

散列技术既是一种存储方法也是一种查找方法。

如果两个关键字 key1≠key2，但 f(key1) = f(key2)，则为冲突(collision)，key1 和 key2 称为散列函数 f 的同义词(synonym)

冲突可以通过设计散列函数减少，但不能完全避免，所以如何处理冲突很重要。

❁ 好的散列函数设计原则：

① 计算简单

复杂计算耗费时间，特别对于频繁查找，会大大降低效率。所以散列函数的计算时间不应该超过其他查找技术与关键字比较的时间。

② 散列地址分布均匀

存储空间有效利用，并且减少为处理冲突耗费的时间。

❁ 散列函数构造方法

① 直接定址法

对 0~100 岁人口数统计，关键字为年龄，那么可以直接拿年龄的数字作为地址：

f (key) = key

地址	年龄	人数
00	0	500 万
01	1	600 万
02	2	450 万
……	……	……
20	20	1500 万
……	……	……

地址	出生年份	人数
00	1980	1500 万
01	1981	1600 万
02	1982	1300 万
……	……	……
20	2000	800 万
……	……	……

如统计 80 后出生年份的人口数，可以用出生年份减去 1980 作为地址，即

f (key) = key - 1980

这两种都是取关键字的某个线性函数的值作为散列地址，即：

f (key) = a × key + b (a, b 是常数)

线性散列函数简单、均匀，但需要事先知道关键字分别情况，适合查找表较小且连续的情况，实际不常用。

② 数学分析法

如手机号码都是 11 位，前 3 位是接入号，一般对应运营商的子品牌，如 130 是联通如意通、136 是移动神州行、153 是电信等；中间 4 位是 HLR 识别号，表明归属地；后 4 位是用户号。

如果以手机号码作为关键字，同一地方很可能会前 7 位相同；可以选择后 4 位作为散列地址。

130xxxx1234
130xxxx2345
138xxxx4829
138xxxx2396
138xxxx8354

抽取是使用关键字的一部分计算散列地址的方法，散列函数常用手段。
数学分析法适合处理关键字位数比较大的情况，适用于事先知道关键字的若干位分布比较均匀。

③ 平方取中法
比如关键字是 1234，平方是 1522756，抽取中间 3 位是 227；
关键字的 4321，平方是 18671041，抽取中间 3 位是 671 或 710
适用于不知道关键字分布，位数不是很大的情况。

④ 折叠法
从左到右将关键字分成位数相等的几部分，求和并按散列表长度取后几位作为散列地址。比如关键字 9876543210，散列表长度是 3 位，将关键字分为 4 组：987|654|321|0，求和 987+654+321+0=1962，取后 3 位得散列地址为 962。
有时候也可以将某几组反转再求和，取后几位。
折叠法适用于不知道关键字分布和关键字位数较多的情况。

⑤ 除留余数法
f (key) = key mod p (p≤散列表长度)
此方法是最常用的构造散列函数的方法。可以直接对关键字求余数，也可以先折叠、平方取中后求余数。此方法 p 的选择是关键。

如下表 12 个关键字用 *f (key) = key mod 12* 记录的散列表：

下标	0	1	2	3	4	5	6	7	8	9	10	11
关键字	12	25	38	15	16	29	78	67	56	21	22	47

但 p = 12 选择的不好，如 18、30、42 等 mod 12 余数都是 6，都与 78 冲突了。
// 就算 p 选 11，在不知道关键字的情况下怎么能确定 11 比 12 冲突少?
经验是通常 p 取不大于散列表长度的最大质数或不包含小于 20 质因数的合数。

⑥ 随机数法
f (key) = random (key)
random 是随机函数，适用于关键字长度不等时。

对于字符串关键字，可以转为数字处理，如 ASCII 码或 Unicode 码等。
根据不同情况选取散列函数：
1) 计算散列地址需要时间；
2) 关键字长度；
3) 散列表大小；
4) 关键字分布情况；
5) 查找的频率。

20180415

❁ 处理散列冲突的方法

散列函数设计的再精妙也不可能完全避免冲突。

① 开放定址法

一旦发生冲突就寻找下一个空的散列地址

$f_i(key) = (f(key) + d_i) \bmod m$ (d_i=1,2,3,…,m-1, m 为散列表长度)

例如关键字集合[12,67,56,16,25,37,22,29,15,47,48,34]，表长 12

散列函数 f (key) = key mod 12

下标	0	1	2	3	4	5	6	7	8	9	10	11
关键字	12	25	37		16			67	56			

前 5 个数都没有冲突，key=37 时 f(37)=1 与 key=25 位置冲突，根据上面公式得 f(37) = (f(37) + 1) mod 12 = 2，就如同要买的房子被别人买了于是买了下一间房子的做法，37 填入下标为 2。后面 22,29,15,47 都没有冲突：

下标	0	1	2	3	4	5	6	7	8	9	10	11
关键字	12	25	37	15	16	29	48	67	56		22	47

key=48 时，f(48)=0 与 key=12 位置冲突，于是 d 从 1 开始试，直到 d=6，f(48) = (f(48)+6) mod 12 = 6 才有空位，存入。

此种解决冲突的开放定址法称为线性探测法。

本来不是同义词却要争夺一个地址的情况称为堆积。如 f(48)=0 和 f(37)=1 不是同义词，但是 f(48)与 f(12)冲突，需要开放地址探测，于是途中与 47 争夺。

最后一个 key=34，f(key)=10，与 key=22 位置冲突，需要不断累加求余直到下标为 9，但是下标 9 就在下标 10 左边，这样找效率太低，可以考虑双向寻找。

$di=1^2,-1^2,2^2,-2^2,\ldots,q^2,-q^2,(q\le m/2)$，于是对于 key=34 只需要取 di=-1 即可。

增加平方运算是为了不让关键字都聚集在某一块区域。

此种方法称为二次探测法。

另一种对于位移量 di 采用随机函数计算得到，称为随机探测法。

此处的随机数是伪随机数，设置随机种子相同，不断调用随机函数可以生成不重复的数列，查找时同样的随机种子，得到的数列是相同的，相同的 di 得到的散列地址自然相同。

② 再散列函数法

$f_i(key) = RH_i(key)$

RH_i 是不同的散列函数，比如除留余数、折叠、平方取中等。每当冲突时，换一个散列函数计算，总会解决冲突。此方法使得关键字不聚集但增加计算时间。

③ 链地址法

将关键字为同义词的记录存储在一个单链表——同义词子表，散列表只存储同义词子表的头指针。

绝不会出现找不到地址的情况，查找时增加了遍历单链表的性能损耗。

④ 公共溢出区法

为冲突的关键字建立一个公共的溢出区存放，如图孤儿院一般。

查找时，先计算出散列地址，若此位置值与给定的关键字相等，则查找成功；反之则到溢出表去顺序查找。如果冲突数据很少，此方法性能将很高。

20180417

- 散列表查找算法 C 语言实现

```c
#include <stdio.h>
#include <stdlib.h>
#define SUCCESS 1
#define UNSUCCESS 0
#define NULLKEY -32768
int HASHSIZE = 8; // 散列表长度，初始化为 8
typedef int Status;
typedef struct {
    int* arr;
    int cnt; // 当前元素个数. 貌似根本没用到
} HashTable; // 哈希表
void init(HashTable* ph, int size){
    if (HASHSIZE < size){
        HASHSIZE += (size - HASHSIZE) / 4 * 4 + 3;
    }
    ph->arr = (int*)malloc(HASHSIZE*sizeof(int));
    for (int i = 0; i < HASHSIZE; ++i){
        ph->arr[i] = NULLKEY; // 初始化全为 NULLKEY
    }
}
int hash(int key){ // 散列函数
    return key % HASHSIZE; // 除留余数法
}
void insert(HashTable* ph,int key){
    int index = hash(key); // 散列地址
    while (ph->arr[index] != NULLKEY){ // 不为空则冲突
        index = (index + 1) % HASHSIZE; // 线性探测开放地址法
    }
    ph->arr[index] = key; // 有空位则插入
}
Status search(HashTable h, int key, int* index){
    *index = hash(key); // 散列地址
    while (h.arr[*index] != key){ // 如果不相等,则可能存在冲突
        *index = (*index + 1) % HASHSIZE; // 线性探测
        // 如果遇到空位或循环了一圈则没找到
        if (h.arr[*index] == NULLKEY || *index == hash(key)){
            return UNSUCCESS;
        }
```

```
    }
    return SUCCESS;
}
void print_arr(int* arr, int size){
    // 格式化打印数组
    printf("[");
    if (size == 0){
        printf("]\n");
        return;
    }
    for (int i = 0; i < size - 1; ++i){
        printf("%d, ", arr[i]);
    }
    printf("%d]\n", arr[size - 1]);
}
int main(int argc, char const *argv[]){
    int arr[] = {40, 42, 96, 66, 13, 34, 98, 30, 67, 74};
    HashTable ht;
    int length = sizeof(arr) / sizeof(arr[0]);
    init(&ht, length);
    for (int i = 0; i < length; ++i){
        insert(&ht,arr[i]);
    }
    print_arr(ht.arr, HASHSIZE); // [66, 34, 13, 30, 67, 74, -32768,
40, 96, 42, 98]
    return 0;
}
```

如果没有冲突，散列查找时间复杂度为 O(1)。但实际应用中，冲突不可避免。
散列查找平均查找长度(ASL)取决于：
① 散列函数是否均匀
不同散列函数对同一组随机对的关键字产生冲突的可能性相同，可以忽略其对 ASL 的影响。
② 处理冲突的方法
比如线性探测处理冲突可能产生堆积，没有二次探测法好；链地址法不会产生任何堆积，具有更好的平均查找性能。
③ 散列表的装填因子
装填因子 α=填入表的关键字个数/散列表长度，标志散列表的装满长度。填入表关键字越多，α 越大，产生冲突的可能性越大。
将散列表的空间设置的比查找集合大可以解决这一问题。虽然浪费点空间，但查找效率大大提升。

20180419

❁ 排序的稳定性

排序前 $k_i=k_j$ 且排序前 k_i 在 k_j 前面(i<j)，如果排序后仍然是 k_i 在 k_j 前面，则排序算法是稳定的；反之，排序后 k_j 在 k_i 前面则排序算法不稳定。

❁ 内排序和外排序

内排序：在排序过程中待排序记录全部被放置在内存中；
外排序：排序的记录个数太多，不能同时放置在内存，整个排序过程需要在内外存之间多次交换数据。
目前只考虑内排序。

❁ 排序算法性能影响因素

① 时间性能

是衡量排序算法好坏的最重要标准。在内排序主要进行比较和移动两个操作。
比较：关键字之间比较；移动：记录从一个位置移动到另一个位置。
高效的内排序算法具有尽可能少的比较次数和尽可能少的记录移动次数。

② 辅助空间

辅助存储空间是除了执行算法时待排序记录所占存储空间之外的其他存储空间。

③ 算法的复杂性

指算法本身的复杂度而不是算法的时间复杂度。
根据排序过程借助的主要操作，内排序分为：插入排序、交换排序、选择排序、归并排序。这些都是比较成熟的排序算法，很多都封装到了编程代码中。
学习排序的目的不是为了实现排序算法，而是为了提高编写算法的能力。

❁ 主要排序算法

以下排序统一为从小到大。

① 冒泡排序 (Bubble Sort)

比较相邻记录的关键字，如果顺序不对则交换，直到顺序全部正确。

```
function randint(a, b) {
    // 获取[a,b]的随机一个整数
    if (arguments.length === 0) {
        a = 10;
        b = 99;
    } else if (arguments.length === 1) {
        b = a;
        a = 0;
    }
    return Math.floor(Math.random() * (b - a + 1) + a);
}
function get_random_arr(n) {
    // 获取 n 个 10~99 随机数的数组, 默认 10 个
    n = n || 10;
    let arr = [];
```

```
    for (let i = 0; i < n; i++) {
        arr.push(randint(10, 99));
    }
    return arr;
}
function swap(arr, i, j) {
    // 交换数组下标为 i 和 j 的元素
    let tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}
function bubble_sort(arr) {
    // 冒泡排序
    for (let i = arr.length - 1; i > 0; i--) {
        for (let j = 0; j < i; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr, j, j + 1);
            }
        }
    }
}
let arr = get_random_arr();
print(arr); // [71, 20, 58, 64, 98, 54, 24, 56, 47, 26]
bubble_sort(arr);
print(arr); // [20, 24, 26, 47, 54, 56, 58, 64, 71, 98]
```

对于[1, 2, 3, 4, 5]这种排序好的，第一圈发现没有交换元素，说明已经排序好了，此时可以直接结束，而不需要下一圈继续判断。

冒泡排序优化：

```
function bubble_sort(arr) {
    // 冒泡排序改进版
    for (let i = arr.length - 1; i > 0; i--) {
        let exchange = false;
        for (let j = 0; j < i; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr, j, j + 1);
                exchange = true;
            }
        }
        if (!exchange) {
            break;
        }
    }
}
```

最好情况，数组是有序的，只需比较 n-1 次，没有数据交换，时间复杂度为 O(n)；
最坏情况，数组是逆序的，需要比较 n-1+n-2+…+2+1 = n(n-1)/2 次，并移动相等数量级的次数，总体时间复杂度为 O(n^2)。

② 选择排序 (Selection Sort)
第 i 趟排序，通过 n-i 次关键字的比较，从 n-i+1 个记录中选出最小的，并和第 i 个(本趟排序第 1 个)记录交换。

```
function select_sort(arr) {
    // 选择排序
    for (let i = 0; i < arr.length - 1; i++) {
        let min_index = i;
        for (let j = i + 1; j < arr.length; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j; // 从 i~len-1 选择最小元素的下标
            }
        }
        swap(arr, min_index, i);
    }
}
```

选择排序最大特点是交换移动次数相当少。但无论什么情况比较次数都是一样的，第 i 趟排序，需要 n-i 次比较，也就是 n-1+n-2+…+2+1 = n(n-1)/2 次。虽然交换次数少，但总体时间复杂度也是 O(n^2)。
选择排序与冒泡排序都是 O(n^2)，但性能上选择排序略优。

③ 插入排序 (Insertion Sort)
将第 i+1 个元素插入到排序好的前 i 个元素的有序数组中。

```
function insert_sort(arr) {
    // 插入排序，从 1 开始，一个元素肯定有序
    for (let i = 1; i < arr.length; i++) {
        // 将第 i 个元素插入前面的有序表中，找到位置直接 break，没必要往前比较了
        for (let j = i; j >= 1; j--) {
            if (arr[j] < arr[j - 1]) {
                swap(arr, j, j - 1);
            } else {
                break;
            }
        }
    }
}
```

最好情况，数组本身有序，每趟比较一次，总共 n-1 次，时间复杂度 O(n)；最坏和平均都是 O(n^2)。插入排序性能比比冒泡和选择稍好。

④ 希尔排序 (Shell Sort)
D.L.Shell 于 1959 年提出的一种排序算法。

将相距某个增量的记录组成一个子序列，对子序列直接插入排序，缩小增量重复，直至增量为 1。希尔排序也叫缩小增量排序。
增量序列是递减的一个序列，其最后一个值必须为 1。

如数组为[9, 1, 5, 8, 3, 7, 4, 6, 2]，增量 increment = increment / 3 + 1
开始 increment = length / 3 + 1 = 4

9	3	2
1	7	
5	4	
8	6	

2	3	9
1	7	
4	5	
6	8	

得到[2, 1, 4, 6, 3, 7, 5, 8, 9]
increment = 4 / 3 + 1 = 2

2	4	3	5	9
1	6	7	8	

2	3	4	5	9
1	6	7	8	

得到[2, 1, 3, 6, 4, 7, 5, 8, 9]
increment = 2 / 3 + 1 = 1
就是简单的插入排序，得：[1, 2, 3, 4, 5, 6, 7, 8, 9]
可以看出序列从无序变得基本有序，再到最后的有序。
// 这个例子增量到后面一直是 1，岂不是还要判断使其只执行一次?

```
function shell_sort(arr) {
    // 原始希尔排序，增量序列为每次除 2 取整
    for (let gap = parseInt(arr.length / 2); gap > 0; gap = parseInt(gap / 2)) {
        for (let i = gap; i < arr.length; i++) { // 插入排序，将 1 变为增量 gap
            for (let j = i; j >= gap; j -= gap) {
                if (arr[j] < arr[j - gap]) {
                    swap(arr, j, j - gap);
                } else {
                    break;
                }
            }
        }
    }
}
```

原始希尔排序最坏情况 $T = \Theta(n^2)$。
希尔排序的关键在于增量的选取，这是一个数学难题。
Hibbard 增量序列：$2^k - 1$，相邻元素互质，最坏情况 $T = \Theta(n^{3/2})$；猜想：$T_{avg}=O(n^{5/4})$
Sedgewick 增量序列：{1, 5, 19, 41, 109, …} ($9\times4^i-9\times2^i+1$ 或 $4^i-3\times2^i+1$)
猜想：$T_{avg}=O(n^{7/6})$，$T_{worst}=O(n^{4/3})$
由于是跳跃式的移动，希尔排序是不稳定排序算法。

20180420

⑤ 堆排序 (Heap Sort)
利用堆(一般取最大堆)进行排序的方法。待排序序列构造最大堆，堆顶即为最大

结点，将它与堆最后一个结点交换，将剩余 *n*-1 个序列重新构造一个最大堆，再重复直至变为有序。

```
function heap_sort(arr) {
    // 堆排序
    for (let i = Math.floor(arr.length / 2) - 1; i >= 0; i--) {
        heap_adjust(arr, i, arr.length); // 从最后叶结点的父结点开始构建最大堆
    }
    for (let last = arr.length - 1; last > 0; last--) {
        swap(arr, 0, last); // 交换堆顶和堆最后元素
        heap_adjust(arr, 0, last); // 将 0~last-1 元素调整为最大堆
    }

    function heap_adjust(arr, start, length) {
        // 将 start~length-1 元素向下筛选调整为最大堆
        let tmp = arr[start]; // 暂存初始堆顶元素
        let child = 2 * start + 1; // child 初始为 start 的左儿子
        while (child < length) {
            if (child + 1 < length && arr[child + 1] > arr[child]) {
                child++; // 存在右儿子且右儿子比左儿子大
            }
            if (tmp > arr[child]) {
                break; // 找到合适位置
            }
            arr[start] = arr[child]; // 下面的大元素上移
            start = child; // 两个指针向下筛选
            child = 2 * start + 1;
        }
        arr[start] = tmp; // 循环退出时 start 为合适位置，插入 tmp
    }
}
let arr = get_random_arr();
print(arr); // [58, 79, 21, 73, 26, 91, 75, 18, 69, 88]
heap_sort(arr);
print(arr); // [18, 21, 26, 58, 69, 73, 75, 79, 88, 91]
```

构建堆的时间复杂度为 O(*n*)，一次堆调整为 O(log*n*)，需要 n-1 次，重建堆时间复杂度为 O(*n*log*n*)，总体堆排序的时间复杂度也是 O(*n*log*n*)。
堆排序对原始的排序状态不敏感，最好、最坏、平均时间复杂度都是 O(*n*log*n*)。
空间复杂度上，只用 tmp 记录暂存元素。
但由于元素比较和交换是跳跃式进行，堆排序不是稳定的排序。
因为初始构建堆比较次数较多，所以不适合待排序序列个数较少的情况。

20180422

⑥ 归并排序 (Merging Sort)

初始序列 n 个记录，看成 n 个有序子序列(1 个元素当然是有序的)，然后两两归并，得到 ceil(n/2)个长度为 2 或 1 的子序列，再两两归并，重复直至得到一个长度为 n 的有序序列，两两归并称为 2 路归并排序。

```
function merge_sort(arr) {
    // 归并排序，统一函数接口
    // 传入 tmp 数组是为了减少申请和释放临时数组的次数
    msort(arr, [], 0, arr.length - 1);

    function msort(arr, tmp, left, right) {
        if (right - left < 1) { // 小于等于 1 个元素就是有序
            return;
        }
        // 对 left~mid 和 mid+1~right 两边分别归并排序
        let mid = left + Math.floor((right - left) / 2);
        msort(arr, tmp, left, mid);
        msort(arr, tmp, mid + 1, right);
        merge(arr, tmp, left, mid, right); // 合并
    }

    function merge(arr, tmp, left, mid, right) {
        let lp = left, rp = mid + 1, tp = left;
        // 按顺序将 arr 数组的 left~mid 和 mid+1~right 合并到临时数组 tmp
        while (lp <= mid && rp <= right) {
            if (arr[lp] <= arr[rp]) {
                tmp[tp++] = (arr[lp++]);
            } else {
                tmp[tp++] = (arr[rp++]);
            }
        }
        // 其中一个没加完，把剩余继续添加
        while (lp <= mid) {
            tmp[tp++] = (arr[lp++]);
        }
        while (rp <= right) {
            tmp[tp++] = (arr[rp++]);
        }
        // tmp 数组元素倒回 arr 数组
        for (let i = left; i <= right; i++) {
            arr[i] = tmp[i];
        }
    }
}

let arr = get_random_arr();
```

```
print(arr); // [31, 81, 73, 95, 22, 89, 45, 98, 40, 86]
merge_sort(arr);
print(arr); // [22, 31, 40, 45, 73, 81, 86, 89, 95, 98]
```

对于一次归并排序 $T(n) = 2T(n/2) + n$，$O(n) = n\log n$，这是归并排序最好、最坏、平均时间复杂度。至于空间复杂度，临时数组与原数组大小一样，递归深度为 $\log_2 n$ 的栈空间，总体为 $O(n + \log n)$。
归并排序占用内存高，但是稳定、效率高。一般用于外排序而不是内排序。

归并排序的非递归版本：

```
function merge_sort2(arr) {
    // 归并排序非递归版本
    let tmp = [];
    // 初始子序列长度为 1，后面每次 x2
    for (let s = 1; s < arr.length;) {
        // 从 arr 倒到 tmp，再倒回 arr
        merge_pass(arr, tmp, s);
        s *= 2;
        merge_pass(tmp, arr, s);
        s *= 2;
    }

    function merge_pass(arr, tmp, s) {
        // 一次合并
        let i = 0;
        // i~i+2s-1 共 2s 个元素，分为 2 组每组 s 个元素，合并
        for (; arr.length - i >= 2 * s; i += 2 * s) {
            // left=i, right=i+2s-1, mid=i+s-1
            merge(arr, tmp, i, i + s - 1, i + 2 * s - 1);
        }
        // 尾巴不足 2s 个元素
        if (arr.length - i > s) { // 如果尾巴长度大于 s，可以分为两个子列，也去合并
            merge(arr, tmp, i, i + s - 1, arr.length - 1); // 此处 right 一定是 len-1
        } else { // 尾巴只有一个子列，直接添加到后面
            for (let j = i; j < arr.length; j++) {
                tmp[j] = arr[j];
            }
        }
    }

    function merge(arr, tmp, left, mid, right) {
        let lp = left, rp = mid + 1, tp = left;
        // 按顺序加入临时数组 tmp
        while (lp <= mid && rp <= right) {
            if (arr[lp] <= arr[rp]) {
```

```
                tmp[tp++] = (arr[lp++]);
            } else {
                tmp[tp++] = (arr[rp++]);
            }
        }
        // 其中一个没加完，继续添加
        while (lp <= mid) {
            tmp[tp++] = (arr[lp++]);
        }
        while (rp <= right) {
            tmp[tp++] = (arr[rp++]);
        }
        // 此处没有从 tmp 倒回到 arr
    }
}
```

非递归版本避免了递归时 $\log_2 n$ 的栈空间，空间复杂度为 O(n)。

20180423

⑦ 快速排序 (Quick Sort)

C++ STL、Java SDK、.NET FrameWork SDK 等开发工具包中源代码都能看到快速排序的某种实现版本。快速排序算法由图灵奖获得者 Tony Hoare 设计出来，被称为 20 世纪十大算法之一。

希尔排序相当于插入排序的升级；堆排序相当于选择排序的升级；快速排序就是冒泡排序的升级。

基本思想：通过一趟排序将待排序记录根据某个 key 分割成独立两部分，一边全部比 key 小、另一边全部比 key 大，分别对两部分进行排序，直至全部有序。

```
function quick_sort(arr) {
    // 快速排序
    qsort(arr, 0, arr.length - 1);

    function qsort(arr, left, right) {
        if (left >= right) {
            return;
        }
        let pivot = partition(arr, left, right); // 枢轴
        // pivot 左边比 key 小右边比 key 大，对左右递归
        qsort(arr, left, pivot - 1);
        qsort(arr, pivot + 1, right);
    }
    function partition(arr, left, right) {
        let key = arr[left]; // 关键值取第 1 个
        let low = left, high = right;
```

```
        while (low < high) {
            // 与 key 相等时往一侧放，比如此处放右边
            while (arr[high] >= key && low < high) {
                high--;
            }
            swap(arr, low, high);
            while (arr[low] < key && low < high) {
                low++;
            }
            swap(arr, low, high);
        }
        // 循环结束 low=high，对应位置为 key
        return low;
    }
}
```

快速排序和二叉搜索树本质貌似是一样的，最好时间复杂度就是树基本平衡时，为 O($n$log$n$)；最坏情况就是一颗斜树，时间复杂度为 O(n^2)。
至于空间复杂度，主要是递归时栈空间的使用，最好为 O(log$n$)，最坏为 O(n)。
由于关键字比较和交换是跳跃进行，快速排序是不稳定的排序。

快速排序优化
1) 优化选取枢轴
主要是三数取中法(median-of -three)，取左中右三个数，排序，选取中间数作为枢轴。随机选取三个数因为随机数生成带来额外开销，反而不好。
对于待排序列非常大的情况可以用九数取中法(median-of-nine)。

2) 优化不必要的交换
用 key 记录关键字，后面可以不用 swap()交换而是直接覆盖值，循环结束，在 low 处重新赋值为 key。

partition 函数使用三数取中法和优化交换：

```
    function partition(arr, left, right) {
        let low = left, high = right;
        let mid = left + Math.floor((right - left) / 2);
        // 三数取中法选取枢轴
        if (arr[left] > arr[right]) {
            swap(arr, left, right); // 保证 left<=right
        }
        if (arr[mid] > arr[right]) {
            swap(arr, mid, right); // 保证 mid<=right
        }
        if (arr[mid] > arr[left]) {
            swap(arr, mid, left); // 此时 mid<=left<=right
        }
```

```
    // left 为左中右三个关键字的中间值
    key = arr[left];
    while (low < high) {
        // 与 key 相等时往一侧放，比如此处放右边
        while (arr[high] >= key && low < high) {
            high--;
        }
        arr[low] = arr[high]; // 右侧的小数往左边放，替换比交换效率高
        while (arr[low] < key && low < high) {
            low++;
        }
        arr[high] = arr[low]; // 左侧大数往右边放
    }
    // 循环结束 low=high，对应位置设为 key
    arr[low] = key;
    return low;
}
```

3) 优化小数组的排序方案

大炮打蚊子这种大材小用有时反而不好用。对于小数组，快速排序反而不如插入排序效率高。原因是快速排序使用了递归，大量数据的话，这点性能损失是可以忽略的；对于数组只有几个数据，反而性能不好。

```
let MAX_LENGTH = 7; // 数组长度阈值

function qsort(arr, left, right) {
    if (right - left > MAX_LENGTH) { // 大于阈值使用快速排序
        let pivot = partition(arr, left, right); // 枢轴
        // pivot 左边比 key 小右边比 key 大，对左右递归
        qsort(arr, left, pivot - 1);
        qsort(arr, pivot + 1, right);
    } else { // 小于等于阈值使用插入排序
        insert_sort(arr);
    }
}
```

4) 优化递归操作

递归对性能有一定影响，如 qsort()函数尾部有两次递归。

修改成尾递归形式：

```
function qsort(arr, left, right) {
    if (right - left > MAX_LENGTH) { // 大于阈值使用快速排序
        while (left < right) {
            let pivot = partition(arr, left, right); // 枢轴
            // 对左边递归排序
            qsort(arr, left, pivot - 1);
```

```
            left = pivot + 1;
        }
    } else { // 小于等于阈值使用插入排序
        insert_sort(arr);
    }
}
```

先对左边递归，left=pivot+1，再次循环调用 partition(arr, left, right)相当于 partition(arr, pivot+1, right)，效果等同于 qsort(arr, pivot+1 right)；采用迭代可以缩减栈深度，提高整体性能。

❁ 排序算法总结

排序算法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	O(n^2)	O(n)	O(n^2)	O(1)	稳定
选择排序	O(n^2)	O(n^2)	O(n^2)	O(1)	不稳定
插入排序	O(n^2)	O(n)	O(n^2)	O(1)	稳定
希尔排序	O($n$log$n$)~O(n^2)	O($n^{1.3}$)	O(n^2)	O(1)	不稳定
堆排序	O($n$log$n$)	O($n$log$n$)	O($n$log$n$)	O(1)	不稳定
归并排序	O($n$log$n$)	O($n$log$n$)	O($n$log$n$)	O(n)	稳定
快速排序	O($n$log$n$)	O($n$log$n$)	O(n^2)	O(log$n$)~ O(n)	不稳定

综合各项指标，优化的快速排序性能最好，但不同场合需要考虑不同的算法。