**算法复杂度**

时间复杂度：程序在执行时耗费时间的长度的量度。

空间复杂度：程序在执行时占用存储空间的大小的量度（包括栈空间）。

**堆**

建立：从最后一个结点的父结点开始对每个结点进行下滤。（时间复杂度为O(N)）

插入：先插入到最后，从其父结点开始进行上滤，直至到达合适位置。

删除：将最后一个元素移到根结点，依次进行下滤，直至到达合适位置。

**哈夫曼树的建立**

给定n个权值，构建n个只有叶节点的二叉树，每次选取两个权值最小的二叉树重新计算权值后放回直至构造成一个二叉树。

**红黑树（RB-Tree）和二叉平衡树（AVL）**

红黑树特点：

1. 每个结点不是红色就是黑色；

2. 根节点是黑色的；

3. 如果一个节点是红色的，则它的两个孩子结点是黑色的；

4. 对于每个结点，其到所有后代叶结点的路径上，均包含相同数目的黑色结点；

5. 每个叶子结点都是黑色的（指空结点）。

二叉平衡树要求高度平衡而红黑树只追求部分平衡，在插入和删除时红黑树最多进行三次旋转操作，而二叉平衡树需要向上更新；查询时由于AVL树的高度平衡所以查询效率很高。均为O(log(N)。

2-3树：节点有一个或两个码，当有一个码时节点有两个子树，一小一大；当有两个码(a, b)时节点有三个子树，一个小于a，一个在a与b之间，一个大于b。所有叶节点都在同一层上。

**跳表**

一种基于有序链表的数据结构，在链表中增加多级索引来加速查找，每一级索引跨度越大，节点数越少。

查找时从最高级索引开始，查找到合适位置时向下级查找。时间复杂度：O(log(N))。

插入时维护平衡性：通过一个随机函数来决定将结点插入到哪几级索引中。（随机出K，就插入1-K级索引）

相比于红黑树的优点：按区间查找数据、迭代输出有序序列。可用于做排行榜。

**B+树**

B树：每个节点存储了更多的键值和数据，对数据范围进行了细分，每一个范围对应一个子节点。

B+树：数据均保存在子节点上，使得非子节点能保存的关键字大大增加，层级更少，查询速度更快，数据有序，全节点遍历更快（叶子节点有双指针指向同级）。用于文件系统（每一页的存储空间有限，且高度越小，磁盘I/O所花时间越少）。

**图相关算法**

**最小生成树算法**

**Prim算法**：从某一顶点出发，选择它的最小权值的边，将其顶点加入顶点集S中，此后从满足“两端点一个在S集中、一个在非S集中”的边中选择出权值最小的，把对应顶点加入到S集中，直到所有的顶点都加入到S集中为止。

**Krustal算法**：初始建立一个有所有原结点但无边的非连通图，每次从边集中选择两端点在T中不连通的权值最小的边加入到T中，直至所有顶点都连通。

**最短路径算法**

**Dijkstra算法**：单源最短路径算法。初始化源点相邻结点的路径值，其余为无穷大，每次从未被访问的顶点中选出路径最短的顶点u，访问它并更新其相邻的未被访问的结点v的路径值为[原值]与[u的路径值与u-v路径之和]之间的最小值，直至所有顶点都被访问。

**Floyd算法**：求任意两顶点之间的最短路径。定义两顶点间最短路径为边长，中间点k遍历所有顶点，则两顶点i和j的最短路径为原路径与i→k→j路径的最小值。

**拓扑排序**

先找到任意一个入度为0的顶点，输出该顶点并将其相连所有边入度减1，重复直至输出所有顶点。

**Morris遍历**

void morrisTraversal(TreeNode\* root) {

TreeNode\* p1 = root;

while (p1) {

if (p1->left) {

TreeNode\* p2 = p1->left;

while (p2->right && p2->right != p1) {

p2 = p2->right;

}

if (p2->right == nullptr) {

p2->right = p1;

// 在这里进行前序遍历操作

p1 = p1->left;

}

else {

p2->right = nullptr;

// 在这里进行中序遍历操作

p1 = p1->right;

}

}

else {

// 在这里进行前序和中序遍历操作

p1 = p1->right;

}

}

}

**散列查找/哈希算法**

根据数据对象的关键字通过一个函数关系对应到一个函数值，将其作为对象的存储地址进行存放。

处理冲突的方法：

① 线性探测：每次加1
。

② 平方探测：增量序列，+1，-1，+4，-4，+9，-9。

③ 双重哈希：在冲突发生时对键应用第二个哈希函数，增量序列(hash1(key) + i \* hash2(key)) % TABLE\_SIZE
。

③ 分离链表法：将冲突对象放到一个单链表中。

字符串密码哈希算法：MD5、SHA等。

**排序算法**

**冒泡排序**：从前往后进行遍历，若连续两元素反序，则交换顺序，若这次遍历中没有交换过顺序，则停止，否则排除最后一个元素继续遍历。

**选择排序**：依次从未排序的序列中选择最小的元素和序列首位元素交换，直至最后一个元素。

**插入排序**：初始状态已排序序列仅包含第一个元素，此后将未排序序列中元素逐一插入到已排序序列中。

**希尔排序**：将待排序一组元素按一定间隔分为若干序列，分别进行插入排序。开始时间隔较大，每轮逐步减至为1。

**归并排序**：将大小为N的序列看成N个长度为1的子序列，接下来将相邻子序列两两进行归并操作，形成N/2个长度为2的有序子序列，然后再继续进行相邻子序列两两归并操作，循环直至剩下一个长度为N的序列。

**快速排序**：将未排序元素根据一个作为基准的主元分为两个序列，其中一个子序列均小于主元而另一个序列均大于主元。然后递归地对两个序列进行排序。

**堆排序**：根据序列建立堆，依次输出堆顶元素。



<https://leetcode.cn/problems/sort-an-array/>

// 快速排序

void quickSort(vector<int>& nums, int l, int r) {

if (l >= r) return;

swap(nums[l], nums[rand() % (r - l + 1) + l]);

int i = l + 1, k = l + 1, j = r;

while (i <= j) {

if (nums[i] < nums[l]) {

swap(nums[i], nums[k]);

i++;

k++;

}

else if (nums[i] > nums[l]) {

swap(nums[i], nums[j]);

j--;

}

else {

i++;

}

}

swap(nums[l], nums[k - 1]);

quickSort(nums, l, k - 2);

quickSort(nums, i, r);

}

// 归并排序

void mergeSort(vector<int>& nums, vector<int>& tmp, int l, int r) {

if (l >= r) return;

int m = (l + r) / 2;

mergeSort(nums, tmp, l, m);

mergeSort(nums, tmp, m + 1, r);

int i = l, j = m + 1;

int k = l;

while (i <= m && j <= r) {

if (nums[i] <= nums[j]) {

tmp[k++] = nums[i++];

}

else {

tmp[k++] = nums[j++];

}

}

while (i <= m) tmp[k++] = nums[i++];

while (j <= r) tmp[k++] = nums[j++];

for (k = l; k <= r; k++) nums[k] = tmp[k];

}

// 堆排序

void percDown(vector<int>& nums, int l, int r) {

int data = nums[l];

int parent = l, child = l \* 2 + 1;

while (child < r) {

if (child + 1 < r && nums[child + 1] > nums[child]) child++;

if (nums[child] <= data) break;

nums[parent] = nums[child];

parent = child;

child = parent \* 2 + 1;

}

nums[parent] = data;

}

void heapSort(vector<int>& nums) {

int n = nums.size();

for (int i = n / 2 - 1; i >= 0; i--) {

percDown(nums, i, nums.size());

}

for (int i = n - 1; i > 0; i--) {

swap(nums[i], nums[0]);

percDown(nums, 0, i);

}

}

**二分查找**

旋转数组的最小值：

<https://leetcode.cn/problems/xuan-zhuan-shu-zu-de-zui-xiao-shu-zi-lcof/>

// 旋转数组的最小值

int minArray(vector<int>& nums) {

int l = 0, r = nums.size() - 1;

while (l < r) {

int m = (l + r) / 2;

if (nums[m] < nums[r]) {

r = m;

}

else if (nums[m] > nums[r]) {

l = m + 1;

}

else r--;

}

return nums[r];

}

搜索旋转数组是否存在目标：

<https://leetcode.cn/problems/search-in-rotated-sorted-array-ii/>

bool search(vector<int>& nums, int target) {

int l = 0, r = nums.size() - 1;

while (l <= r) {

int m = (l + r) / 2;

if (nums[m] == target) return true;

if (nums[l] == nums[m] && nums[m] == nums[r]) {

l++;

r--;

}

else if (nums[m] <= nums[r]) {

if (nums[m] < target && nums[r] >= target) {

l = m + 1;

}

else r = m - 1;

}

else {

if (nums[m] > target && nums[l] <= target) {

r = m - 1;

}

else l = m + 1;

}

}

return false;

}

搜索旋转数组并返回最小索引

<https://leetcode.cn/problems/search-rotate-array-lcci/description/>

int search(vector<int>& nums, int target) {

int l = 0, r = nums.size() - 1;

while (l <= r) {

if (nums[l] == target) {

return l;

}

int m = (l + r) / 2;

if (nums[m] == target) {

r = m;

}

else if (nums[l] == nums[m] && nums[m] == nums[r]) {

l++;

r--;

}

else if (nums[m] <= nums[r]) {

if (nums[m] < target && nums[r] >= target) {

l = m + 1;

}

else r = m - 1;

}

else {

if (nums[m] > target && nums[l] < target) {

r = m - 1;

}

else {

l = m + 1;

}

}

}

return -1;

}

**字符串匹配KMP算法**

对较短字符串进行提前处理，建立一个字符串大小的next数组。

初始化next[0] = 0，若pattern[next [j – 1]] == pattern[j]，则next [j] = next [j – 1] + 1，否则next [j] = 0.

在将主串和短串进行匹配时，主串比较位置不需要回退，只需要回退短串，大大提高了效率。

<https://leetcode.cn/problems/find-the-index-of-the-first-occurrence-in-a-string/>

int kmp(const string& s, const string& p) {

int sn = s.size();

int pn = p.size();

vector<int> next(pn, 0);

int i = 1, j = 0;

while (i < pn) {

if (p[i] == p[j]) {

next[i++] = ++j;

}

else if (j > 0) {

j = next[j - 1];

}

else i++;

}

i = j = 0;

while (i < sn && j < pn) {

if (s[i] == p[j]) {

i++;

j++;

}

else if (j > 0) {

j = next[j - 1];

}

else i++;

}

return j == pn ? i - pn : -1;

}

**树状数组**

一种可以动态维护序列前缀和的数据结构，可以解决区间问题。

提供方法：**单点更新update**和**区间查询query**。

注意下标从1开始。

<https://leetcode.cn/problems/count-of-range-sum/>

class BIT {

public:

int n;

vector<int> tree;

BIT(int \_n) : n(\_n), tree(\_n + 1) {}

int lowbit(int x) {

return x & (-x);

}

void update(int x, int v) {

while (x <= n) {

tree[x] += v;

x += lowbit(x);

}

}

int query(int x) {

int res = 0;

while (x > 0) {

res += tree[x];

x -= lowbit(x);

}

return res;

}

};

**A\*算法**

A\*是比较经典的寻路算法，它的重点就是启发函数：总代价F=当前点到起点的代价G+当前点到终点的代价H，代价可以是距离或者权重等。算法的核心是两个列表，开启列表和关闭列表，未走过的节点放入开启列表，已走过的节点放入关闭列表。

启发函数可以是：曼哈顿距离（只允许上下左右）、欧几里得距离（直接距离）、对角线距离（先对角线到同一列或同一行再直线）。

改进：对启发函数（H）添加权重系数、搜索邻域优化（舍齐掉反方向邻域）、双向搜索（1从起点出发而2从终点出发，1以2的开启列表中F最小的节点为终点，2同理，直至两者都到达终点，即两者相遇）。

**指数循环节**

对指数进行降幂，其中φ(C)表示欧拉函数。欧拉函数：小于n的正整数中与n互质的数的数目。

截图.png

**十进制与二进制的转换**

整数部分：除二取余法，然后将结果倒序。

小数部分：乘二取整法。

**最大公约数/最小公倍数**

<https://leetcode.cn/problems/find-greatest-common-divisor-of-array/>

int gcd(int a, int b) {

return b == 0 ? a : gcd(b, a % b);

}

int lcm(int a, int b) {

return a / gcd(a, b) \* b;

}

**计数质数**

<https://leetcode.cn/problems/count-primes/>

int countPrimes(int n) {

vector<int> isPrime(n, true);

int res = 0;

for (int i = 2; i < n; i++) {

if (isPrime[i]) {

res++;

for (int j = i \* i; j < n; j += i) {

isPrime[j] = false;

}

}

}

return res;

}

**实现计算器**

实现只有"+"、"-"、"("、")"的计算器。

[https://leetcode.cn/problems/basic-calculator/](https://leetcode.cn/problems/basic-calculator/description/)

int calculate(string s) {

int n = s.size();

stack<int> ops;

ops.emplace(1);

int cur = 1;

int res = 0;

int num = 0;

for (int i = 0; i < n; i++) {

if (s[i] >= '0' && s[i] <= '9') {

num = num \* 10 + (s[i] - '0');

}

if ((s[i] < '0' || s[i] > '9') && s[i] != ' ' || i == n - 1) {

res += num \* cur;

num = 0;

switch (s[i]) {

case '+':

cur = ops.top();

break;

case '-':

cur = -ops.top();

break;

case '(':

ops.emplace(cur);

break;

case ')':

ops.pop();

break;

}

}

}

return res;

}

实现只有"+"、"-"、"\*"、"/"的计算器。

[https://leetcode.cn/problems/basic-calculator-ii/](https://leetcode.cn/problems/basic-calculator-ii/description/)

int calculate(string s) {

int n = s.size();

int lnum = 0, rnum = 0, num = 0;

char op = '+';

for (int i = 0; i < n; i++) {

if (s[i] >= '0' && s[i] <= '9') {

num = num \* 10 + (s[i] - '0');

}

if ((s[i] < '0' || s[i] > '9') && s[i] != ' ' || i == n - 1) {

switch (op) {

case '+':

lnum += rnum;

rnum = num;

break;

case '-':

lnum += rnum;

rnum = -num;

break;

case '\*':

rnum \*= num;

break;

case '/':

rnum /= num;

break;

}

num = 0;

op = s[i];

}

}

return lnum + rnum;

}