数据结构与常用算法

# 五大常用算法

## 分治

待解决的复杂问题能够简化为几个若干个小规模相同的问题，且**各子问题间独立存在，即子问题之间不包含公共的子问题，子问题的解可以合并为该问题的解。**

### 问题特点

1. 子问题之间不包含公共的子问题，即子问题之间相互独立；
2. 子问题的解可以合并为原问题的解。

### 适用问题

1. 二分搜索
2. 合并排序
3. 快速排序
4. 二叉树的生成
5. 大整数乘法
6. Strassen矩阵乘法
7. 棋盘覆盖
8. 线性时间选择
9. 最接近点对问题
10. 循环赛日程表
11. 汉诺塔

## 贪心算法

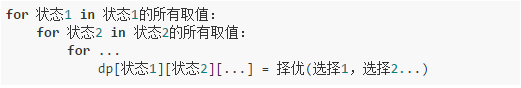
## 动态规划

每次决策依赖于当前状态，又随即引起状态的转移。一个决策序列就是在变化的状态中产生出来的，所以，这种**多阶段最优化决策解决问题**的过程就称为动态规划。

### 问题特点

1. 子问题之间包含公共的子问题，即子问题之间相互不独立；
2. 最优子结构，问题的最优解所包含的子问题的解也是最优的；

### 解题套路



### 适用问题

#### 背包问题

##### 公式

###### 组合问题公式

dp[i] = dp[i] + dp[i - num]

###### 真假问题公式

dp[i] = dp[i] | dp[i - num]

###### 最大最小问题公式

dp[i] = min(dp[i], dp[i - num] + 1)

或

dp[i] = max(dp[i], dp[i - num] + 1)

##### 循环方向

###### 0-1背包问题

数组中的元素不可重复使用，nums放在外循环，target在内循环，且内循环倒序。

*void* ZeroOneBackpack(*int* \**nums*, *int* *numsSize*, *int* *target*)

{

*int* \*dp = (*int* \*)malloc((*target* + 1) \* sizeof(*int*));

    for (*int* i = 0; i < *numsSize*; i++) {

        for (*int* j = *target*; j >= *nums*[i]; j--) {

            dp[j] = dp[j] + dp[j - *nums*[i]];

            dp[j] = dp[j] | dp[j - *nums*[i]];

            dp[j] = fmin(dp[j], dp[j - *nums*[i]] + 1);

        }

    }

    return;

}

###### 完全背包问题

数组中的元素可重复使用，nums放在外循环，target在内循环。且内循环正序。

*void* CompleteBackpack(*int* \**nums*, *int* *numsSize*, *int* *target*)

{

*int* \*dp = (*int* \*)malloc((*target* + 1) \* sizeof(*int*));

    for (*int* i = 0; i < *numsSize*; i++) {

        for (*int* j =  *nums*[i]; j <= *target*; j++) {

            dp[j] = dp[j] + dp[j - *nums*[i]];

            dp[j] = dp[j] | dp[j - *nums*[i]];

            dp[j] = fmin(dp[j], dp[j - *nums*[i]] + 1);

        }

    }

    return;

}

###### 组合问题

需考虑元素之间的顺序，需将target放在外循环，将nums放在内循环。

*void* CompleteBackpack(*int* \**nums*, *int* *numsSize*, *int* *target*)

{

*int* \*dp = (*int* \*)malloc((*target* + 1) \* sizeof(*int*));

    dp[0] = 0;

    for (*int* j =  1; j <= *target*; j++) {

        for (*int* i = 0; i < *numsSize*; i++) {

            dp[j] = dp[j] + dp[j - *nums*[i]];

            dp[j] = dp[j] | dp[j - *nums*[i]];

            dp[j] = fmin(dp[j], dp[j - *nums*[i]] + 1);

        }

    }

    return;

}

#### 子序列子数组问题

#### 股票问题

## 回溯法

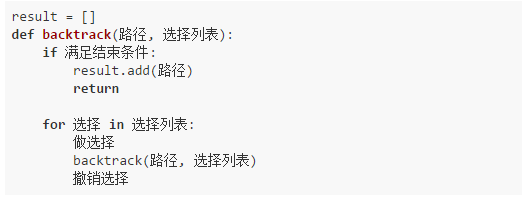
回溯法是一种搜索算法，从根节点出发，按照**深度优先搜索的策略**进行搜索，到达某一节点后 ，探索该节点是否包含该问题的解，如果包含则进入下一个节点进行搜索，若是不包含则回溯到父节点选择其他支路进行搜索。**回溯法一般来说是遍历整个解空间，获取问题的所有解。**

DFS 是一个劲的往某一个方向搜索，而回溯算法建立在 DFS 基础之上的，但不同的是在搜索过程中，达到结束条件后，恢复状态，回溯上一层，再次搜索。因此回溯算法与 DFS 的区别就是**有无状态重置。**

### 问题特点

### 解题套路

* 1. 路径：记录已经做出的选择。
  2. 选择列表：当前可以做的选择。
  3. 结束条件：到达决策树底层，无法再做选择的条件。



### 适用问题

#### 子集问题

#### 组合问题

#### 排列问题

#### 八皇后问题

#### 迷宫问题

#### 数独问题

### 注意事项

1. 定义宏不能太大或者太小；
2. 去重需要先排序，并传入下标的start值；
3. 子集问题，没有结束条件；

#define M 10000

bool \*g\_selected;

int g\_count;

int \*g\_path;

int g\_numsSize;

void dfs(int \*nums, int \*\*ans, int curDepth)

{

    if (curDepth == g\_numsSize) {

        memcpy(ans[g\_count], g\_path, g\_numsSize \* sizeof(int));

        (g\_count)++;

        return;

    }

    for (int i = 0; i < g\_numsSize; i++) {

        if (g\_selected[i] == 0) {

            g\_selected[i] = 1;

            g\_path[curDepth] = nums[i];

            dfs(nums, ans, curDepth + 1);

            g\_selected[i] = 0;

        }

    }

}

int\*\* permute(int \*nums, int numsSize, int\* returnSize, int\*\* returnColumnSizes)

{

    \*returnSize = 0;

    if (nums == NULL || numsSize <= 0) {

        \*returnColumnSizes = NULL;

        return NULL;

    }

    int \*\*ans = (int \*\*)malloc(M \* sizeof(int \*));

    if (ans == NULL) {

        \*returnColumnSizes =  NULL;

        return NULL;

    }

    for (int i = 0; i < M; i++) {

        ans[i] = (int \*)malloc(numsSize \* sizeof(int));

    }

    g\_path = (int \*)malloc(numsSize \* sizeof(int));

    if (g\_path == NULL) {

        \*returnColumnSizes =  NULL;

        return NULL;

    }

    g\_selected = (int \*)malloc(numsSize \* sizeof(int));

    if (g\_selected == NULL) {

        \*returnColumnSizes =  NULL;

        return NULL;

    }

    g\_count = 0;

    g\_numsSize = numsSize;

    memset(g\_selected, 0, numsSize \* sizeof(int));

    memset(g\_path, 0, numsSize \* sizeof(int));

    dfs(nums, ans, 0);

    \*returnSize = g\_count;

    \*returnColumnSizes = (int \*)malloc((\*returnSize) \* sizeof(int));

    if ((\*returnColumnSizes) == NULL) {

        free(ans);

        return NULL;

    }

    for (int i = 0; i < \*returnSize; i++) {

        (\*returnColumnSizes)[i] = numsSize;

    }

    return ans;

}

## 分支限界法

和回溯法相似，也是一种搜索算法，但回溯法是找出问题的许多解，而分支限界法是**找出原问题的一个解**。或是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解

在当前节点（扩展节点）处，先生成其所有的儿子节点（分支），然后再从当前的活节点（当前节点的子节点）表中选择下一个扩展节点。为了有效地选择下一个扩展节点，加速搜索的进程，在每一个活节点处，计算一个函数值（限界），并根据函数值，从当前活节点表中选择一个最有利的节点作为扩展节点，使搜索朝着解空间上有最优解的分支推进，以便尽快地找出一个最优解。

分支限界法：

1）FIFO分支限界法

3）优先队列分支限界法：按照优先队列中规定的优先级选取优先级最高的节点成为当前扩展节点。

示例：装载问题，旅行售货员问题

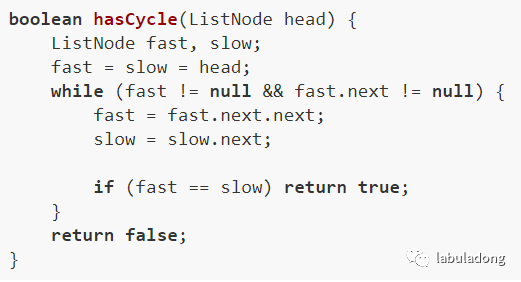
# 数据结构

## 数组与链表

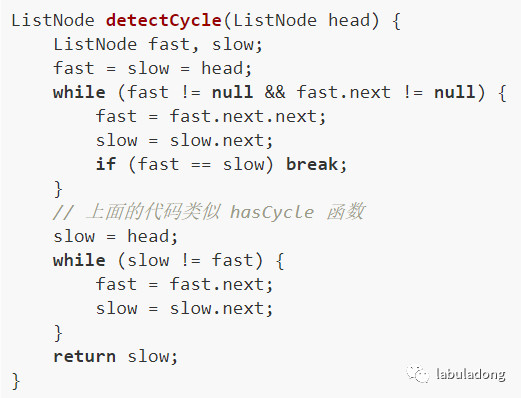
### 左右指针

### 快慢指针——成环问题

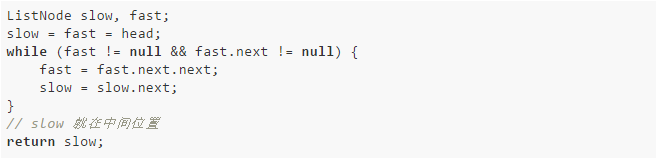
#### 判定链表中是否含有环



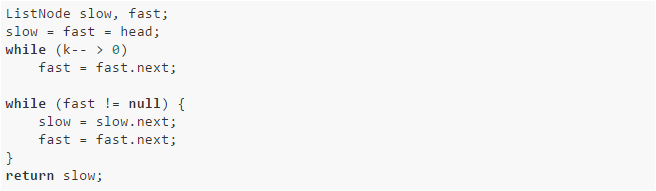
#### 已知链表中含有环，返回这个环的起始位置



#### 寻找链表的中点



#### 寻找链表的倒数第 k 个元素



### 滑动窗口——子串问题

#define M 128

bool IsSatisfy(int \*need, int \*window)

{

    for (int i = 0; i < M; i++) {

        if (need[i] > window[i]) {

            return false;

        }

    }

    return true;

}

bool checkInclusion(char \* s1, char \* s2)

{

    if (s1 == NULL || s2 == NULL) {

        return false;

    }

    int \*need = (int \*)malloc(M \* sizeof(int));

    int \*window = (int \*)malloc(M \* sizeof(int));

    memset(need, 0, M \* sizeof(int));

    memset(window, 0, M \* sizeof(int));

    for (int i = 0; i < strlen(s1); i++) {

        need[s1[i]]++;

    }

    int left = 0;

    int right = 0;

    while (right < strlen(s2)) {

        char c = s2[right];

        right++;

        if (need[c] != 0) {

            window[c]++;

        }

        while (IsSatisfy(need, window) == true) {

            if (right - left == strlen(s1)) {

                return true;

            }

            char c = s2[left];

            left++;

            if (need[c] != 0) {

                window[c]--;

            }

        }

    }

    return false;

}

## 字符串

### 模式匹配

#### BF算法

时间复杂度：N\*M

#### KMP算法

## 树

### 深度优先索索

1. 基本情况；

* 做事；
* 递归；

前序遍历

#### Top Down DFS

* 把值通过参数的形式从上往下传；
* 一般dfs()本身不返回值；

#### Bottom Up DFS

* 把值从下往上传；
* 当前递归层利用子问题传上来的值计算当前层的新值并返回；
* 一定会有返回值；

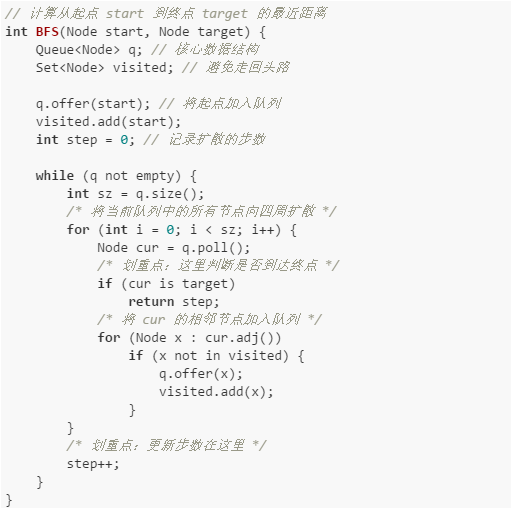
**一般流程**

1. 基本情况；
2. 向子问题要答案；
3. 利用子问题答案构建当前问题答案；
4. 返回答案给父问题。

### 广度优先搜索——最近距离

本质上就是一幅「图」，让你从一个起点，走到终点，问最短路径。

#### 套路



## 图

## 前缀和和查分数组

前缀和的应用场景是，**需要对某个区间[i...j]频繁查询累计和，避免每次查询都遍历这个区间。**

差分数组的应用场景是，**需要对某个区间[i...j]频繁地加或减某一值，避免每次都遍历这个区间。**

# uthash的使用

#include “uthash.h”

## 3个数据结构

### UT\_hash\_handle

### UT\_hash\_table

### UT\_hash\_bucket

## 自定义结构体

typedef *struct* MyNode {

*int* key;

*char* value[10];

    UT\_hash\_handle hh;

} MyNode;

MyNode \*head = NULL;

* key的类型不一样，后面的插入、查找调用的接口函数就不一样，因此要求确保key的类型与uthash的接口函数一致。
* 必须提供UT\_hash\_handle变量hh，无需为其初始化。
* 定义一个hash结构的空指针users，用于指向保存数据的hash表，必须初始化为空，在后面的查、插等操作中，uthash内部会根据其是否为空而进行不同的操作

## 7个函数

### 查找操作HASH\_FIND

#### HASH\_FIND\_INT

#### HASH\_FIND\_STR

#### HASH\_FIND\_PTR

#define HASH\_FIND\_INT(*head*,*findint*,*out*) HASH\_FIND(hh,head,findint,sizeof(*int*),out)

#define HASH\_FIND\_STR(*head*,*findstr*,*out*) HASH\_FIND(hh,head,findstr,strlen(findstr),out)

#define HASH\_FIND\_PTR(*head*,*findptr*,*out*) HASH\_FIND(hh,head,findptr,sizeof(*void* \*),out)

### 添加操作HASH\_ADD

#### HASH\_ADD\_INT

#### HASH\_ADD\_STR

#### HASH\_ADD\_PTR

#define HASH\_ADD\_INT(*head*,*intfield*,*add*) HASH\_ADD(hh,head,intfield,sizeof(*int*),add)

#define HASH\_ADD\_STR(*head*,*strfield*,*add*) HASH\_ADD(hh,head,strfield,strlen(add->strfield),add)

#define HASH\_ADD\_PTR(*head*,*ptrfield*,*add*) HASH\_ADD(hh,head,ptrfield,sizeof(*void* \*),add)

*void* AddNode(*int* *key*, *char* \**value*)

{

    MyNode \*node = NULL;

    HASH\_FIND\_INT(head, &*key*, node);

    if (node != NULL) {

        return;

    }

    node = (MyNode \*)malloc(sizeof(MyNode));

    if (node == NULL) {

        return;

    }

    node->key = *key*;

    strcpy(node->value, *value*);

    HASH\_ADD\_INT(head, *key*, node);

    return;

}

### 替换操作HASH\_REPLACE

#### HASH\_REPLACE\_INT

#### HASH\_REPLACE\_STR

#### HASH\_REPLACE\_PTR

#define HASH\_REPLACE\_INT(*head*,*intfield*,*add*,*replaced*) HASH\_REPLACE(hh,head,intfield,sizeof(*int*),add,replaced)

#define HASH\_REPLACE\_STR(*head*,*strfield*,*add*,*replaced*) HASH\_REPLACE(hh,head,strfield,strlen(add-strfield),add,replaced)

#define HASH\_REPLACE\_PTR(*head*,*ptrfield*,*add*) HASH\_REPLACE(hh,head,ptrfield,sizeof(*void* \*),add,replaced)

### 删除操作HASH\_DEL

#### HASH\_DEL

#define HASH\_DEL(*head*, *delptr*) HASH\_DELETE(hh,head,delptr)

*void* DeleteNode(*int* *key*)

{

    MyNode \*node = NULL;

    HASH\_FIND\_INT(head, &*key*, node);

    if (node == NULL) {

        return;

    }

    HASH\_DEL(head, node);

    free(node);

}

### 得到节点数目HASH\_COUNT

*int* GetNodeCount()

{

    return HASH\_COUNT(head);

}

### 遍历HASH\_ITER

*void* IterAllNode()

{

    MyNode \*curNode = NULL;

    MyNode \*tmpNode = NULL;

    HASH\_ITER(hh, head, curNode, tmpNode) {

        printf("key = %d, value = %s\n", curNode->key, curNode->value);

    }

    return;

}

### 清空HASH\_ITER + HASH\_DEL

*void* DeleteAllNode()

{

    MyNode \*curNode = NULL;

    MyNode \*tmpNode = NULL;

    HASH\_ITER(hh, head, curNode, tmpNode) {

        HASH\_DEL(head, curNode);

        free(curNode);

    }

    return;

}

### 排序HASH\_SORT

*int* CompareByKey(*struct* MyNode \**a*, *struct* MyNode \**b*) {

    return (*a*->key - *b*->key);

}

*int* CompareByValue(*struct* MyNode \**a*, *struct* MyNode \**b*) {

    return strcmp(*a*->value, *b*->value);

}

*void* SortByKey() {

    HASH\_SORT(head, CompareByKey);

}

*void* SortByValue() {

    HASH\_SORT(head, CompareByValue);

}

## 不同类型的key值

### 整数类型int

typedef *struct* MyNode {

*int* key;

*char* value[10];

    UT\_hash\_handle hh;

} MyNode;

MyNode \*head = NULL;

#### HASH\_FIND\_INT(head, &key, node)

#### HASH\_ADD\_INT(head, key, node)

### 字符指针char\*

typedef *struct* MyNode {

*char* \*ptrKey;

*int* value;

    UT\_hash\_handle hh;

} MyNode;

MyNode \*head = NULL;

#### HASH\_ADD\_KEYPTR(hh, head, node->ptrKey, strlen(node->ptrKey), node)

#### HASH\_FIND\_STR(head, stringKey, node)

### 字符数组char[N]

typedef *struct* MyNode {

*char* stringKey[10];

*int* value;

    UT\_hash\_handle hh;

} MyNode;

MyNode \*head = NULL;

#### HASH\_ADD\_STR(head, stringKey, node)

#### HASH\_FIND\_STR(head, stringKey, node)

### 使用地址void\*

#### HASH\_ADD\_PTR

#### HASH\_FIND\_PTR

### 使用结构体

#include <stdlib.h>

#include <stdio.h>

#include "uthash.h"

typedef *struct* {

*char* a;

*int* b;

} MyNodeKey;

typedef *struct* {

    MyNodeKey key;

    /\* ... other data ... \*/

    UT\_hash\_handle hh;

} Node;

*int* main(*int* *argc*, *char* \**argv*[]) {

    Node \*head = NULL;

    Node \*findNode = NULL;

    Node \*curNode = NULL;

    Node \*tmpNode = NULL;

    Node \*node = (Node \*)malloc(sizeof(Node));

    memset(node, 0, sizeof(Node));

    node->key.a = 'a';

    node->key.b = 1;

    HASH\_ADD(hh, head, key, sizeof(MyNodeKey), node);

    MyNodeKey nodeKey = {0};

    memset(&l, 0, sizeof(Node));

    nodeKey.a = 'a';

    nodeKey.b = 1;

    HASH\_FIND(hh, head, &nodeKey, sizeof(MyNodeKey), findNode);

    if (findNode) printf("found %c %d\n", findNode->key.a, findNode->key.b);

    HASH\_ITER(hh, head, curNode, tmpNode) {

      HASH\_DEL(head, curNode);

      free(curNode);

    }

    return 0;

}