1 分析算法复杂度

1. $O(1)$
2. $O(n \cdot m)$
3. $O(n^2)$
4. $O(\log n)$
5. $O(n^2)$
6. $O(\sqrt{n})$

2 寻找两个序列的中位数

1. 设计算法

i. 获取序列A的长度 `length`, 推理可知总长度为 `2 * length`, 需要找到的中位数为第length大的元素;
ii. 双指针 `i`, `j` 分别指向序列A和B的起始位置;
iii. 当 `i + j < length` 时, 循环比较 `A[i]` 和 `B[j]` 的大小;
iv. 当 `A[i]` 小时, `i++`;
v. 当 `B[j]` 小时, `j++`;
vi. 当 `i + j == length` 时, 循环结束, 退出循环, 较小的数值就是中位数;

2. 撰写代码

```
#include <iostream>
#include <vector>

using namespace std;

int findMidNum(vector<int> &a, vector<int> &b)
{
    if(a.size() != b.size()) return -1;

    int length = a.size();
    int i=0, j=0;
    // 注意是 length - 1 因为索引从0开始
    while(i + j < length - 1)
    {
        if(a[i] < b[j])
        {
            i++;
        }
        else
        {
            j++;
```

```
        }
    }

    return a[i] < b[j] ? a[i] : b[j];
}

int main()
{
    vector<int> a = {11, 13, 15, 17, 19};
    vector<int> b = {2, 4, 6, 8, 20};

    cout << findMidNum(a, b) << endl;
    // Output: 11
}
```

3. 算法复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(1)$

时间复杂度主要取决于双指针的移动, 由于每次位移为1, 并且终止条件为length, 所以复杂度为O(n);
空间复杂度由于没有增加额外的存储空间, 所以只需要初始数据的存储, 为O(1);

3 重新排列线性表

1. 设计算法

考虑到题目使用的单向链表无法回溯尾指针, 所以这里考虑把链表分成两半, 对后半部分反转再合并;

i. 使用快慢指针法找到链表中点;
ii. 反转后半部分链表;
iii. 按照abab的顺序合并两个表;

2. 撰写代码

```
#include <iostream>
#include "linklist.cpp"

using namespace std;

template <typename T>
void SinglyLinkedList<T>::rearrange() {
    if (head == nullptr || head->next == nullptr) {
        return; // 空链表或只有一个节点,不需要重排
    }
```

```cpp
    // 找到中间节点
    Node* slow = head;
    Node* fast = head;
    while (fast->next && fast->next->next) {
        slow = slow->next;
        fast = fast->next->next;
    }

    // 将链表分为两部分
    Node* mid = slow->next;
    slow->next = nullptr;

    // 反转后半部分
    Node* prev = nullptr;
    Node* curr = mid;
    while (curr) {
        Node* next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }

    // 合并前半部分和反转后的后半部分
    Node* p1 = head->next;
    Node* p2 = prev;
    while (p2) {
        Node* tmp1 = p1->next;
        Node* tmp2 = p2->next;
        p1->next = p2;
        p2->next = tmp1;
        p1 = tmp1;
        p2 = tmp2;
    }
}

int main()
{
    SinglyLinkedList<int> list {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    cout << "Before rearrange:" << endl;
    list.display();
    list.rearrange();
    cout << "After rearrange:" << endl;
    list.display();
}
```

3. 代码复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(1)$

4 演示中缀表达式的计算

计算: 3 * (6 - 5)

OPTR	OPND
	3
*	3
* (	3
* (	3 6
* (-	3 6
* (-	3 6 5
* (	3 1
*	3 1
	3

p.s. 以字符串前部为栈底，字符串尾部为栈顶

5 回文串判断

```
#include <iostream>
#include <string>
#include <cctype> // 用于tolower函数

using namespace std;

bool isPalindrome(const string& str) {
    // 创建两个指针，分别指向字符串的开头和结尾
    int start = 0;
    int end = str.length() - 1;

    // 循环直到两个指针相遇
    while (start < end) {
        // 跳过非字母字符
        while (start < end && !isalnum(str[start])) {
            start++;
        }
        while (start < end && !isalnum(str[end])) {
            end--;
```

```
        }

        // 检查两个字符是否相同(忽略大小写)
        if (tolower(str[start]) != tolower(str[end])) {
            return false; // 如果不同,则不是回文
        }

        start++;
        end--;
    }

    return true; // 如果两个指针相遇且所有字符都相同,则是回文
}

int main() {
    string str1 = "A man a plan a canal Panama";
    string str2 = "Hello, World!";

    cout << str1 << " is " << (isPalindrome(str1) ? "" : "not ") << "
    cout << str2 << " is " << (isPalindrome(str2) ? "" : "not ") << "

    return 0;
}
```

这个代码稍微更近一步的忽略了大小写和非字母和数字元素。

6 求`next`与`nextval`数组

t	a	b	c	a	a	b	b	a	b	c	a	b
next	0	1	1	1	2	2	3	1	2	3	4	5
nextval	0	1	1	0	2	1	3	0	1	1	0	5

7 从广义表中取出指定元素

```
L = (apple, (orange, (strawberry, (banana)), peach), pear)

L.Tail() = ((orange, (strawberry, (banana)), peach), pear)
.Head() = (orange, (strawberry, (banana)), peach)
.Tail() = ((strawberry, (banana)), peach)
.Head() = (strawberry, (banana))
.Tail() = ((banana))
.Head() = (banana)
.Head() = banana
```

综上，L.Tail().Head().Tail().Head().Tail().Head().Head() = banana

8 寻找未出现的最小正整数

1. 设计算法

算法一

对于这个题目，由于要求时间高效，而对空间不做要求，因此可以利用一个Hash表来记录是否出现某个数，而此处的Hash函数只需要设计为x-1就行。考虑到最差情况为n个不相同的正整数输入，此时未出现的最小正整数为n+1，因此空间复杂度为O(n)。

1. 记数组为 `num`，长度为 `n`，建立一个长度为 `n` 的数组 `hash` 初始化为0；
2. 遍历数组 `num`，将 `hash[num[i]-1]` 置为1；
3. 遍历数组 `hash`，找到第一个值为0的元素，即为未出现的最小正整数；
4. 如果没有找到，则返回n+1。

算法二

我又去问了下Claude看看他有什么想法，他给出了一种原地更新的方法，这种方法本质上是对算法一的一种进步，将额外需要的空间转化为输入的数组本身。

给出算法的基本设计思想：

对于这个问题,我们可以利用原地算法(In-place algorithm)的思想,将数组元素进行重新排列,使得数组中的正整数和它的位置建立一一对应关系,然后遍历数组找出第一个不符合这种对应关系的位置,即对应的就是未出现的最小正整数。具体步骤如下:

1. 遍历数组,将所有小于等于0的数过滤掉,同时记录下数组中正整数的最大值 `max_val`。
2. 如果 `max_val` 大于数组长度n,说明未出现的最小正整数一定在1到n+1之间。否则,未出现的最小正整数可能大于n+1。
3. 使用这种方法后,只需要检查1到 `max_val` 之间的数是否出现在数组中即可。
4. 将数组中每个正整数x,映射到 `index=x-1` 的位置上,同时用负值来标记已访问过的数字。
5. 再次遍历数组,如果当前位置index的值是正数,就表示对应的 `index+1` 是未出现的最小正整数。
6. 如果遍历完整个数组都没有找到,那么就返回 `max_val+1` 作为未出现的最小正整数。

这种算法的时间复杂度为O(n),空间复杂度为O(1),因为我们只利用了原有的数组空间进行原地操作。

2. 撰写函数

```
#include <iostream>
#include <vector>

using namespace std;

int findMissing(vector<int> &nums)
{
    int n = nums.size();
    int* hash = new int[n];      // 构造hash表
```

```
    for (int i = 0; i < n; i++)
        if(nums[i] > 0 && nums[i] <= n)     // 正整数置为1
            hash[nums[i]-1] = 1;
    for (int i = 0; i < n; i++)
        if (hash[i] == 0)          // 找到第一个为0的元素，即为缺失的数
            return i + 1;
    return n + 1;                  // 如果没找到，返回n+1
}

int main()
{
    vector<int> nums = {-1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    cout << findMissing(nums) << endl;
    return 0;
}
```

3. 代码复杂度

- 时间复杂度：O(n)
- 空间复杂度：O(n)

p.s. 如果利用原地更新则空间复杂度可以变为O(1)

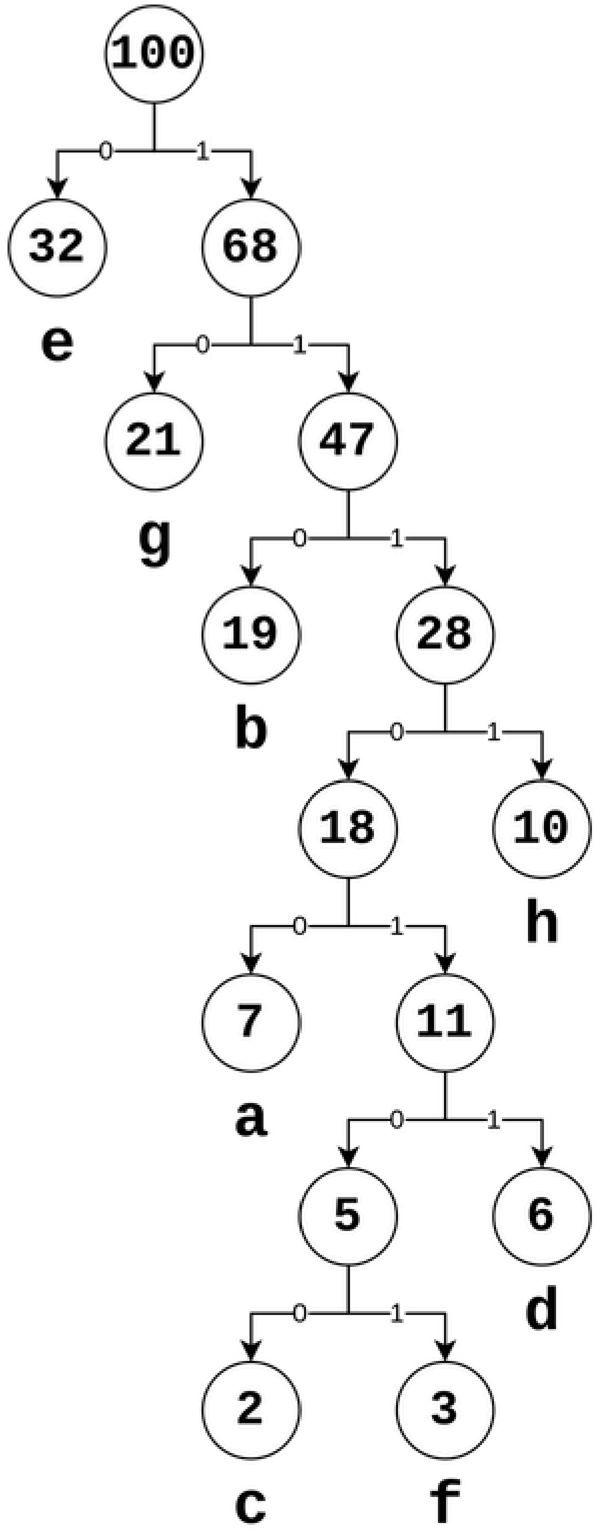
9 设计字符编码

不妨假设8个字符为a~h:

字符	a	b	c	d	e	f	g	h
频率	7	19	2	6	32	3	21	10

1. 设计Huffman编码

构造Huffman树：



从而构造出Huffman编码：

字符	编码
e	0
g	10
b	110
h	1111
a	11100
d	111111

字符	编码
c	1111101
f	1111110

2. 等长编码

直接编号为0~7，二进制编码为3位，$2^3-1=7$，因此可以得到如下编码：

字符	编码
a	000
b	001
c	010
d	011
e	100
f	101
g	110
h	111

3. 比较优缺点

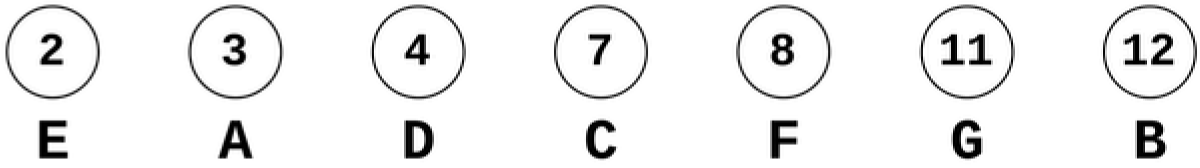
Huffman - 不等长编码:
优点:

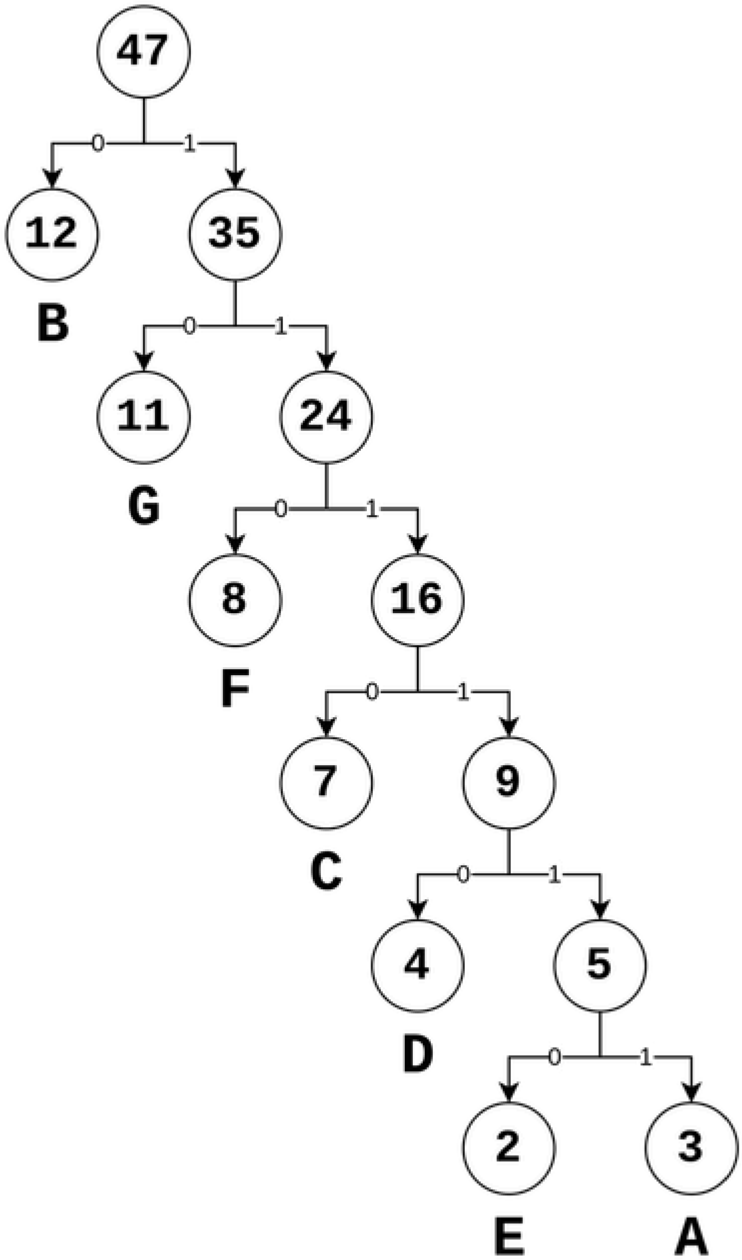
- 根据概率分布，可以得到最优编码。
- 压缩率较高。
- 压缩率随字符个数增加而增加。 缺点:
- 需要存储编码表。
- 需要获取概率分布。
- 算法比较复杂。
- 不等长编码对解码不友好，需要存储编码表。

等长编码:
优点:

- 在概率分布相近时较好。
- 方便解码，每次读入的位数相等。 缺点：
- 所有字符等长，常用字符占用空间多。

10 写出Huffman树





11 前缀特性编码

1. 二叉树;
2. 二叉树的左右路径分布标记为0与1, 叶子节点标记上对应的字符;
    i. 从根节点开始, 遇到0向左, 遇到1向右;
    ii. 遇到叶子节点, 输出字符, 并返回根节点;
    iii. 重复执行上述步骤, 遍历完编码字符串;
3. 构造对应的二叉树, 并检查有没有冲突的叶子节点;
    i. 从根节点开始, 遇到0向左, 遇到1向右, 如果对应结果不存在则添加节点;
    ii. 遍历完当前字符编码后, 如果当前的叶子节点已经存在, 则返回`false`;
    iii. 重复检查所有的字符编码;

iv. 如果没有发现冲突，则满足前缀特性编码，返回`true`；

12 计算带权路径长度

1. 算法的基本设计思想

求解二叉树的带权路径长度(WPL)问题,我们可以采用递归的思路。对于每个节点,我们需要计算从根节点到该节点路径上所有节点的权重之和,然后将这个和乘以该节点所在层的叶子节点数量,得到该节点对WPL的贡献值。最终,我们将所有节点的贡献值相加即可得到整棵树的WPL。

具体步骤如下:

1. 定义一个递归函数,该函数接受当前节点和从根节点到当前节点的路径权重之和为参数。
2. 对于叶子节点,将路径权重之和乘以1(因为只有一个叶子节点),即为该节点对WPL的贡献值。
3. 对于非叶子节点,分别计算左子树和右子树对WPL的贡献值,然后将它们相加即为该节点对WPL的贡献值。
4. 在计算左子树和右子树的贡献值时,需要将当前节点的权重加到路径权重之和中。
5. 最终,我们将所有节点的贡献值相加,即得到整棵树的WPL。

2. 二叉树节点的数据类型定义

```
struct TreeNode {
    TreeNode* left;
    int weight;
    TreeNode* right;
    TreeNode(int w) : weight(w), left(nullptr), right(nullptr) {}
};
```

3. C++算法实现

```
int calculateWPL(TreeNode* root, int pathWeight = 0) {
    if (!root) {
        return 0;  // 空树,WPL为0
    }

    // 计算当前节点到根节点的路径权重之和
    pathWeight += root->weight;

    // 如果是叶子节点,直接返回路径权重之和
    if (!root->left && !root->right) {
        return pathWeight;
    }

    // 递归计算左子树和右子树的贡献值
    int leftWPL = calculateWPL(root->left, pathWeight);
```

```
    int rightWPL = calculateWPL(root->right, pathWeight);

    // 返回当前节点的贡献值,即左子树和右子树贡献值之和
    return leftWPL + rightWPL;
}
```

在主函数中,我们可以这样调用该函数:

```
int main() {
    // 构建一棵二叉树
    TreeNode* root = new TreeNode(5);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(1);
    root->left->right = new TreeNode(4);

    int wpl = calculateWPL(root);
    cout << "Weighted Path Length (WPL) of the binary tree: " << wpl

    return 0;
}
```

这个算法的时间复杂度为$O(n)$,其中n是二叉树中节点的数量,因为我们需要访问每个节点一次。空间复杂度为$O(h)$,其中h是二叉树的高度,因为在最坏情况下(树完全倾斜),递归调用堆栈的深度为h。