马上开始

35tang-C++竞赛系列四阶课程

三五堂

《35tang-C++竞赛系列四阶课程》

竞赛思路

- 严格遵守竞赛要求，确保不会爆0
- 内存占用不能太大，如果数据量太大，可以考虑动态数组等数据结构，如果想不到好的方法，一定要开静态数组，至少满足部分测试用例要求，不要开太大了。
- 考虑算法时间复杂度是否可以得到满分。如果不能也不要放弃，用“笨”方法得到部分分数。
- 排序往往是一个最先考虑的。
- 好的数据结构，比如set，比如map都可以减少时间复杂度。
- 全局变量的命名要小心

我们在学习什么？

- C++的语言和用法
- STL里面的一些特殊的东西，比如sort，set等提供了方便
- 算法：除了基础的模拟枚举之外，算法就是让我们可以在有限的时间内完成大量的数据运算。
- 数据结构：好的数据结构配合好的算法，进一步提高程序的效率。
- 在学习过程中一些算法，数据结构甚至一些比较巧妙的小技巧，一些优化的小方法，一些让我们程序写起来更容易的方法。都应该自己记录并且复习，掌握。
- 学会测试和调试，考虑各种情况
- 很多网站刷题都是cin cout，但是请记住NOIP (CSP)正式竞赛都是文件输入输出，请尽量的多练习文件读写。

作业做不出来怎么办?

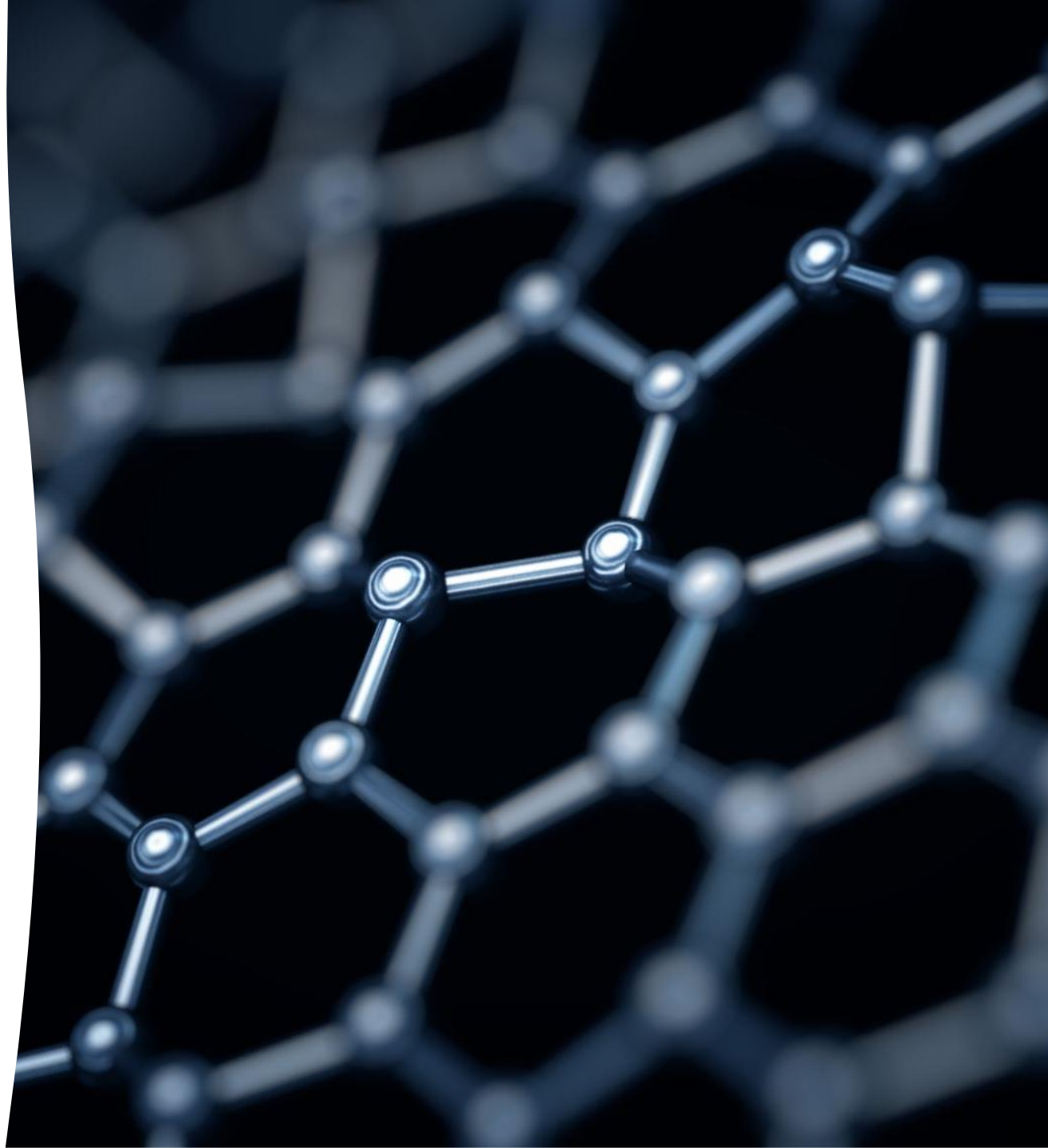
先学习例题

作业一定先尝试思考研究

实在做不出来也没有关系，因为很难，可以尝试用笨方法得到部分分数

可以参考老师答案，网上搜索答案，但是，一定要看懂，理解为什么这么做。然后自己根据学到的思路写出来，注意：不是抄！是学会了以后自己写!

这些难题的学习一方面是新的算法，一方面是一些新的技巧，这些都是我们学习的新思路，需要借鉴。



霍夫曼编码huffman coding

- 在计算机资料处理中，霍夫曼编码使用变长编码表对源符号（如文件中的一个字母）进行编码，其中变长编码表是通过一种评估来源符号出现几率的方法得到的，出现几率高的字母使用较短的编码，反之出现几率低的则使用较长的编码，这便使编码之后的字符串的平均长度、期望值降低，从而达到无损压缩数据的目的。

字符串“to be or not to be“

一般情况下用ASCII码表示字符（计算机内部只有0和1，每一个字符会对应一个整数，和字符一一映射，这个整数就是这个字符的ASCII码），一个字符就是用一个字节（8bit可以表示8位的二进制数）来表示，比如字符'A'就是65，8个bit位是01000001。“to be or not to be“算上空格一共18个字符，那么在计算机内部需要18个字节来存储和表示这个字符串，也就是18*8=144个bits来存储和表示这个字符串。

如何压缩才能用更少的bits来表示这个字符串？比如你有一篇英文的文章，一共1000个字符，如果要存起来，每一个字符用ASCII码，这篇文章就要占用1000个字节的存储。

压缩思路：和ASCII码类似，还是编码，但是：对当前需要处理的一个字符串（或者一篇文章）中出现的字符，不需要用8bit去表示一个字符，而是更少bit位。

定长压缩?

“to be or not to be“

一共出现了7个不同字符，朴素的思路：我们知道3位的二进制数可以表示8个不同的含

- 't' --> 000
- 'o' --> 001
- ' ' --> 010

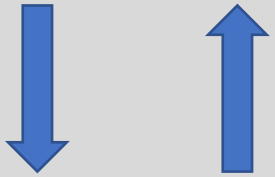
还有更好的办法么？我们发现，其实给定字符串中每一个字符出现的次数（频率）是不一样的，那么编码是不是应该出现多的用长度小的编码，出现少的用长度多的编码？会不会更节省？

按照出现频率给每一个字符（包括空格）新的编码，这样的话出现多的字母占用的bit就比较少，最终的结果肯定是长度最短的（占用的总bit位最少）

“to be or not to be“

- 't' 出现了3次 --> 110
- 'o' 出现了4次 --> 00
- ' ' 出现了5次 --> 10
- 'b' 出现了2次 --> 010
- 'e' 出现了2次 --> 011
- 'r' 出现了1次 --> 1110
- 'n' 出现了1次 --> 1111

to be or not to be



110 00 10 010 011 10 00 1110 10 1111 00 110 10 110 00 10 010 011

- 't' 出现了3次 --> 110
- 'o' 出现了4次 --> 00
- ' ' 出现了5次 --> 10
- 'b' 出现了2次 --> 010
- 'e' 出现了2次 --> 011
- 'r' 出现了1次 --> 1110
- 'n' 出现了1次 --> 1111

不会出现歧义

- 其实这些01中间不需要分隔，因为不会出现歧义，例如
- 11000 是 to，拆开1 1000， 11 000， 1100 0都找不到对应得编码，只有110 00 可以对应to
- 所以整个编码串就是

1100010010011100011101011110011010110010010011

- 一共47个bits，比前面得144个bits的ASCII编码节省了好多空间。

- 't' 出现了3次 --> 110
- 'o' 出现了4次 --> 00
- ‘ ’ 出现了5次 --> 1
- 'b' 出现了2次 --> 010
- 'e' 出现了2次 --> 011
- 'r' 出现了1次 --> 1110
- 'n' 出现了1次 --> 1111

神奇的霍夫曼编码

- 左侧的修改，就失去了唯一性，比如 你遇到一个编码1110，到底是字符r 1110呢？ 还是1 110空格加字符t？

霍夫曼编码的关键

是一种压缩技术

给每一个字符一个新的01bit组成的编码，组成一个编码表

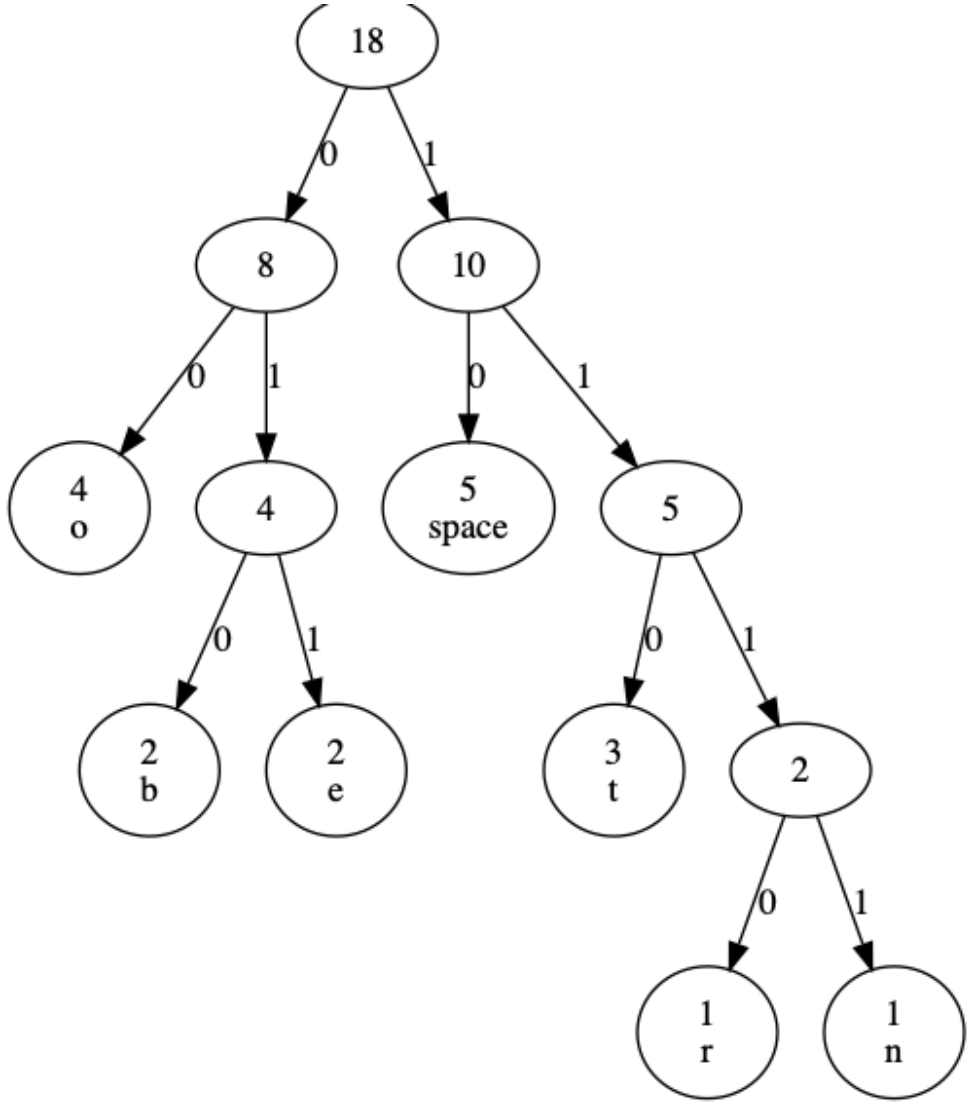
把每一个字符用新的编码，而不是ASCII码，组成新的01bits串

反过来这个新的01bits串在我们需要还原原始字符串的时侯，可以根据编码表唯一的解码成源字符串

由于编码表按照字符出现的频率决定每一个字符对应编码的长度，出现多的编码短，出现少的编码长，所以可以起到压缩的作用。

怎么才能做出编码表---霍夫曼树

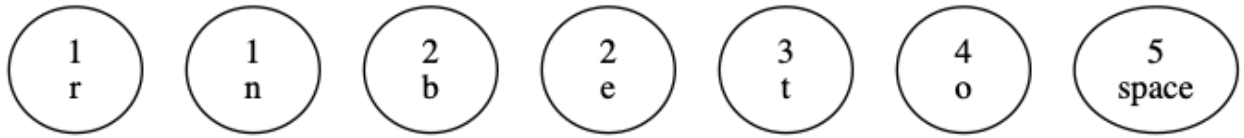
- 如果我们可以构造这么一棵二叉树，每一个叶子节点对应一个字符，该字符的编码就是从根节点到这个叶子节点的路径，路径中每一个节点的左节点就是0，右节点就是1，把经过的路径的01组合起来就是这个叶子节点对应字符的霍夫曼编码了。
- 例如字符n，从根节点开始，编码就是1101
- 出现频率小的叶子节点层次就多一些，编码也长一些

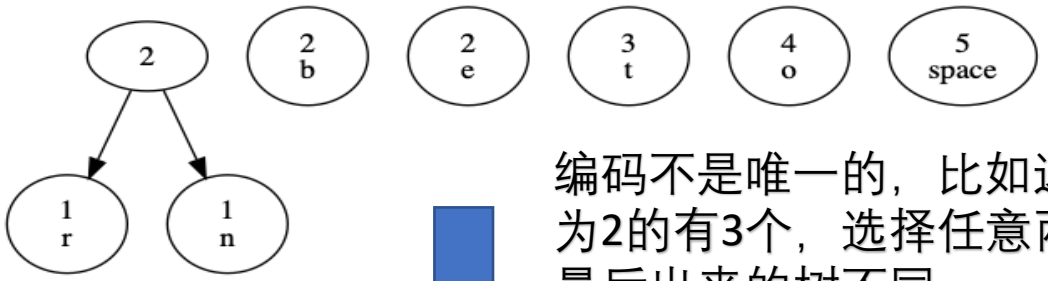


to be or not to be

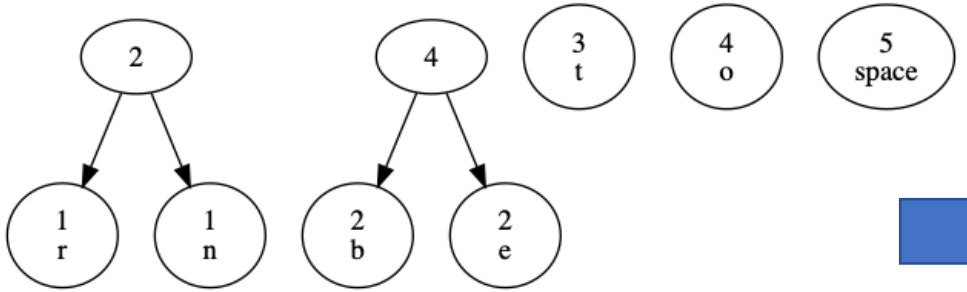
如何构造霍夫曼树（构造出了树就有了霍夫曼编码）

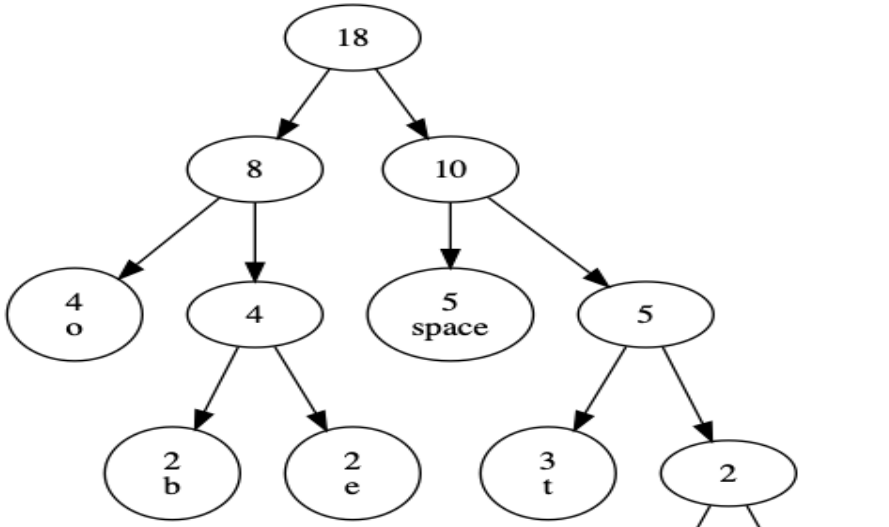
- 刚开始每一个字符都是叶子节点，每一个叶子节点记录该字符以及该字符出现的次数(权值)。然后就是不停的合并节点（出现次数最小的两个节点增加一个父亲节点），合并后新的父亲节点的出现次数（权值）等于合并的两个子节点的出现次数的和。一直到所有节点组成一棵树。

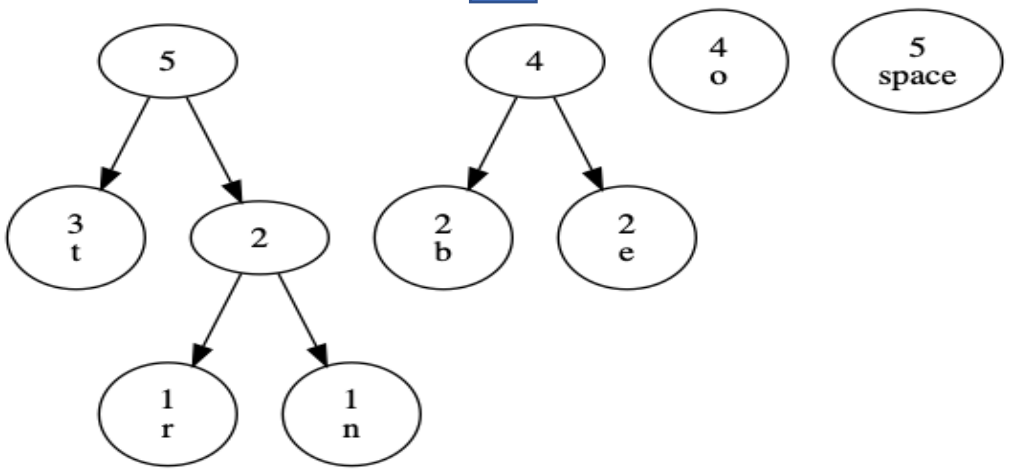




编码不是唯一的，比如这里权值为2的有3个，选择任意两个合并，最后出来的树不同







用数组结构生成树：数组下标就是节点编号

```
struct node {          // 树的节点类型
    char val;          //节点代表的字符，只有叶子节点才有意义
    int key;                 // key，也就是权值（次数）
    int lchild;        // 左孩子指針(左孩子节点在forest数组的数组下标)
    int rchild;        // 右孩子指針(右孩子节点在forest数组的数组下标)
};
vector <node> forest; //保存所有树节点
int frequency[300] = {0};//统计出现字符的频率，如果都是字符，数组大小不会超过255
```

例如字母r出现了1次，这是一个叶子节点，该节点在forest数组的下标如果是2（或者说这个叶子节点的节点编号是2），那么

forest[2]是一个node类型的元素，

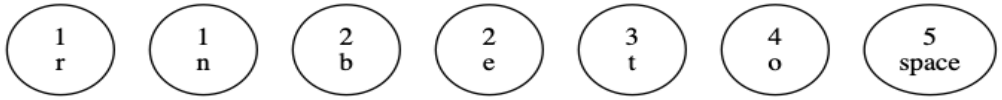
forest[2].val就是’r’;

forest[2].key 就是1;

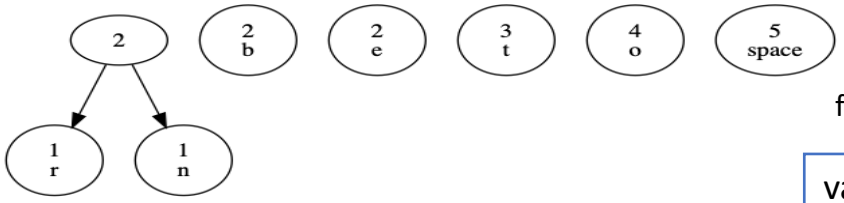
forest[2].lchild 就是-1; 表示没有子节点

forest[2].rchild 就是-1;表示没有子节点

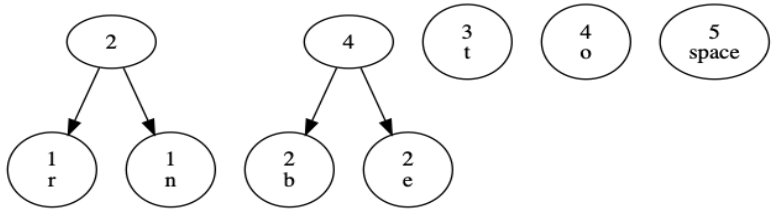
初始化所有出现的字符为一个叶子节点，比如这里出现了7个字符，那么初始化在forest数组中放入7个叶子节点。



forest[0]	forest[1]	forest[2]	forest[3]	forest[4]	forest[5]	forest[6]
val: ' ' key: 5 lchild: -1 rchild: -1	val: 'b' key: 2 lchild: -1 rchild: -1	val: 'e' key: 2 lchild: -1 rchild: -1	val: 'n' key: 1 lchild: -1 rchild: -1	val: 'o' key: 4 lchild: -1 rchild: -1	val: 'r' key: 1 lchild: -1 rchild: -1	val: 't' key: 3 lchild: -1 rchild: -1



forest[0]	forest[1]	forest[2]	forest[3]	forest[4]	forest[5]	forest[6]	forest[7]
val: ' ' key: 5 lchild: -1 rchild: -1	val: 'b' key: 2 lchild: -1 rchild: -1	val: 'e' key: 2 lchild: -1 rchild: -1	val: 'n' key: max lchild: -1 rchild: -1	val: 'o' key: 4 lchild: -1 rchild: -1	val: 'r' key: max lchild: -1 rchild: -1	val: 't' key: 3 lchild: -1 rchild: -1	val: key: 2 lchild: 5 rchild: 3



forest[0]	forest[1]	forest[2]	forest[3]	forest[4]	forest[5]	forest[6]	forest[7]	forest[8]
val: ' ' key: 5 lchild: -1 rchild: -1	val: 'b' key: max lchild: -1 rchild: -1	val: 'e' key: max lchild: -1 rchild: -1	val: 'n' key: max lchild: -1 rchild: -1	val: 'o' key: 4 lchild: -1 rchild: -1	val: 'r' key: max lchild: -1 rchild: -1	val: 't' key: 3 lchild: -1 rchild: -1	val: key: 2 lchild: 5 rchild: 3	val: key: 4 lchild: 1 rchild: 2

一起看看程序

模拟程序：输入一个字符串，生成huffman树，并根据这个树给出出现的每一个字符的huffman编码(用只有01字符的字符串表示)。

注意：

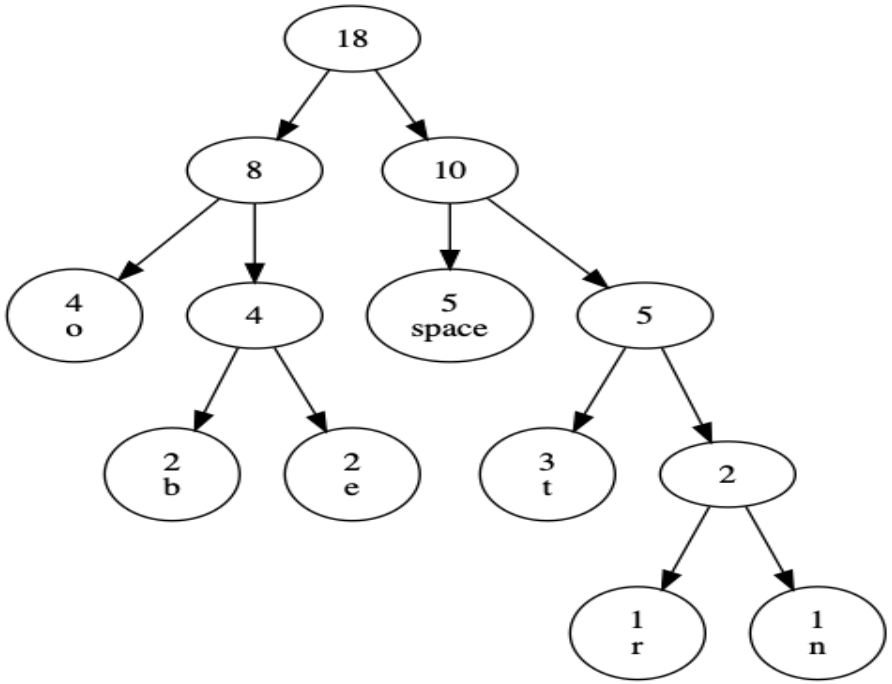
n个叶子节点（不同的字符）一共需要合并n-1次之后就会完成树。

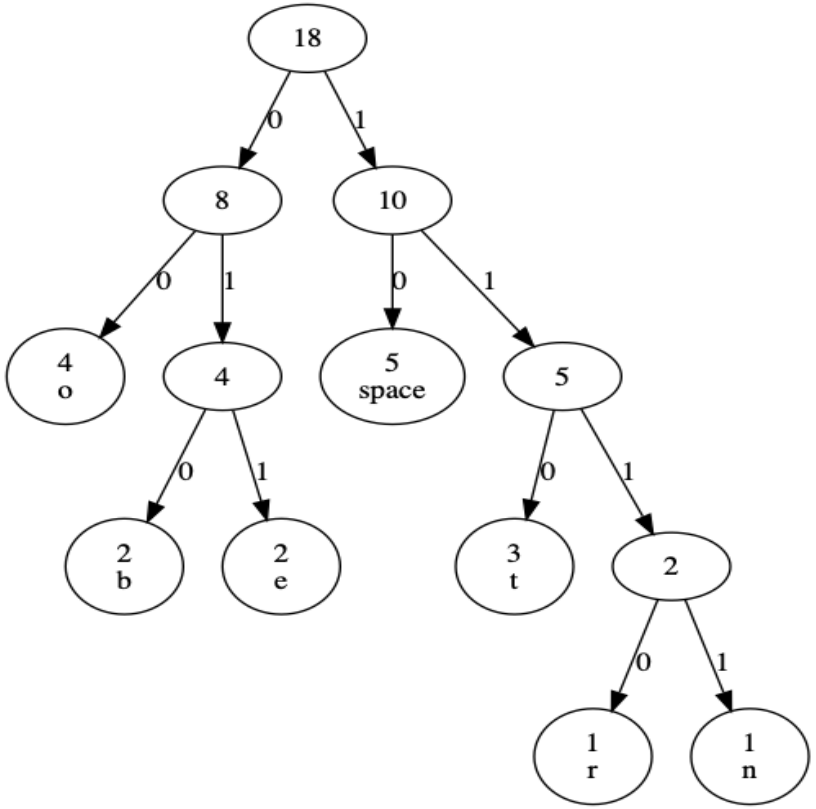
一个节点只会被合并一次。

```
//前面已经准备了n个叶子节点并且放在了forest数组里面。下面合并n-1次。
for (int i = 0; i < n; i++) {//n-1次合并后树就完成了
    node ptr;
    //不能sort，因为现在节点编号就是数组下标，sort就乱了，需要遍历数组找到key第一小和第二小的两个
    //合并过的节点不再使用，修改他们的权值为INT_MAX，下次找最小就找不到了
    int s1=0,s2=0;//当前最小的2个节点编号，也就是数组下标
    for (int i=0;i<forest.size();i++)
            if (forest[i].key<forest[s1].key) s1=i;//打擂找到没有合并过的当前最小节点的数组下标s1
    ptr.key = forest[s1].key;
    forest[s1].key=INT_MAX;  //设置为最大避免再次合并
    for (int i=0;i<forest.size();i++)
            if (forest[i].key<forest[s2].key ) s2=i; //打擂找到没有合并过的当前第二小节点的数组下标s2
    ptr.key+= forest[s2].key;//新节点的key是合并的两个节点s1和s2的key的和
    forest[s2].key=INT_MAX;
     //生成一个新的父亲节点，该节点的左右儿子分别是forest[s1]和forest[s2]这两个节点
    ptr.lchild = s1;
    ptr.rchild = s2;

    forest.push_back(ptr);//插入新节点到数组里面
  }
```

然后遍历生成的树，生成编码：左面路径为0，右面为1





遍历生成的树，生成编码：左面路径为0，右面为1

```
void printCode(int root,string hs) {//hs是从根节点到当前节点经过的‘0’，‘1’字符串
    //dfs遍历树，输出每一个叶子节点（字符）的huffman编码
   if(forest[root].lchild == -1 || forest[root].rchild == -1) {//到了叶子节点，输出字符串
      cout<<forest[root].val<<" "<<hs<<endl;
      return;
   }
   printCode(forest[root].lchild, hs+“0”);//左节点，加上字符’0’走下去
   printCode(forest[root].rchild, hs+“1”); //右节点，加上字符’1’走下去
}
```

主程序调用：

```
printCode(forest.size()-1,“”);
```

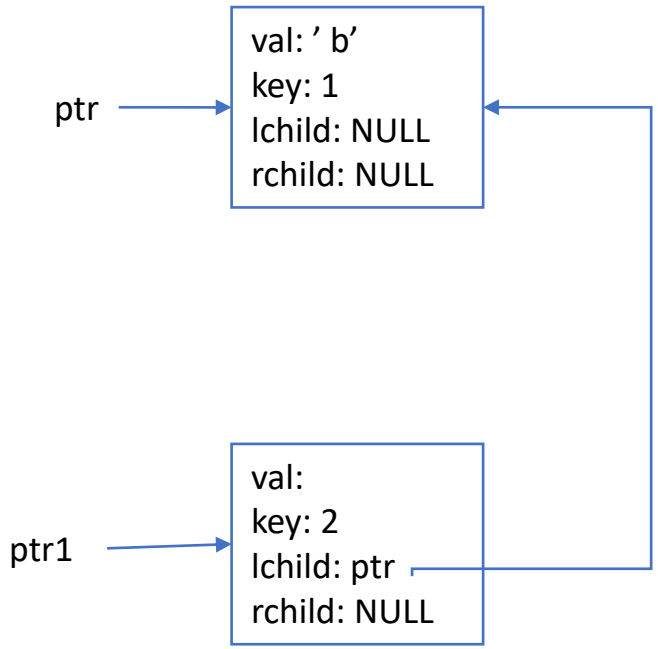
//由于每一次新的节点添加到数组最后，所以n-1次之后，所有的节点合并好的根，也就是最后生成的一个节点在数组的最右面

指针：简单理解就是一个地址，指向内存一块区域

```
struct node {              // 树的节点类型
    char val;              //节点代表的字符，只有叶子节点才有意义
    int key;               // key，也就是权值（次数）
    node *lchild;          // 左孩子指针
    node *rchild;          // 右孩子指针
};
```

```
node *ptr;
ptr=new node;
ptr->val= 'b';
ptr->key= 1;
ptr-> lchild=NULL;
ptr-> rchild=NULL;

node *ptr1;
ptr1=new node;
ptr1->key= 2;
ptr1-> lchild=ptr;
ptr1-> rchild=NULL;
```



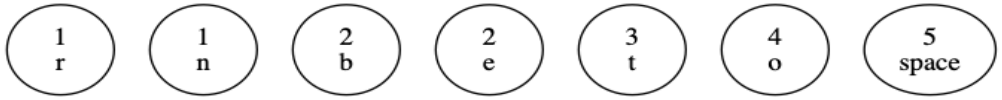
链表数据结构，不是数组，通过指针链接在一起

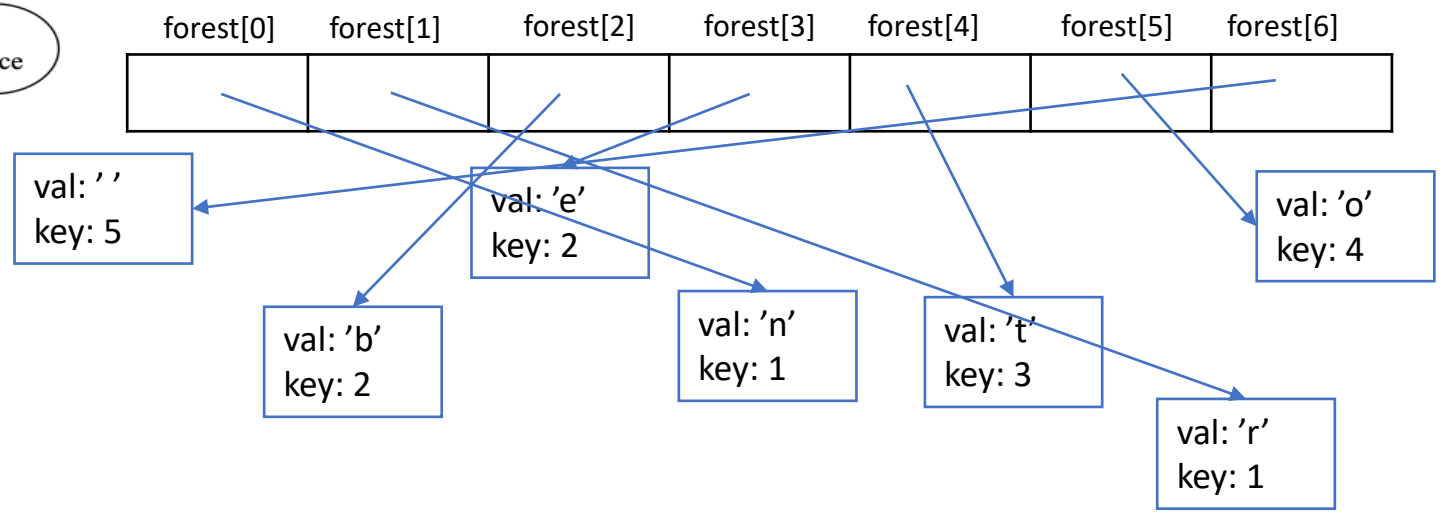
指针实现

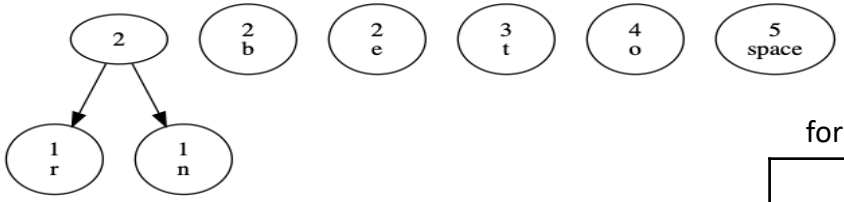
尝试看懂

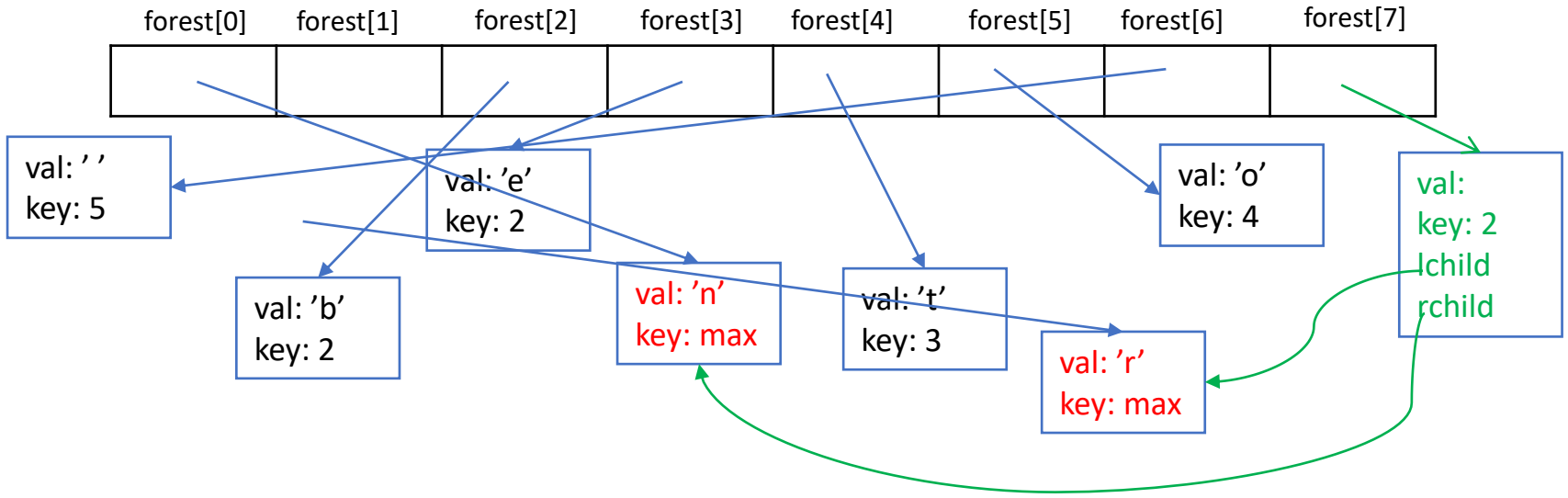
- 输入一个字符串，生成huffman树，并根据这个树给出出现的每一个字符的huffman编码(用只有01字符的字符串表示)。
- 注意：
  - n个叶子节点（不同的字符）一共需要合并n-1次。
  - 使用指针来实现树，指针就是一个地址，就是一个引用，到新的位置，类似链表数据结构，每一个节点是一个struct指针类型，他通过指针直接指向自己的儿子节点（也是struct类型），而不是前面学习的通过数组下标访问儿子节点。
  - 这里：每一个叶子节点是一个结构体指针，把这些指针都放在数组里面，从而方便排序和使用。数组里面放的是指针不是结构体，指针指向一个存放节点信息的结构体。

```
struct node {                                   // 树的节点类型
        char val;                              //节点代表的字符，只有叶子节点才有意义
    int key;                                // key，也就是权值（次数）
    node *lchild;                           // 左孩子指针
    node *rchild;                           // 右孩子指针
};
```


1
r
1
n
2
b
2
e
3
t
4
o
5
space


forest[0]
forest[1]
forest[2]
forest[3]
forest[4]
forest[5]
forest[6]
val: ' '
key: 5
val: 'e'
key: 2
val: 'o'
key: 4
val: 'b'
key: 2
val: 'n'
key: 1
val: 't'
key: 3
val: 'r'
key: 1


2
2
b
2
e
3
t
4
o
5
space
1
r
1
n


forest[0]
forest[1]
forest[2]
forest[3]
forest[4]
forest[5]
forest[6]
forest[7]
val: ' '
key: 5
val: 'e'
key: 2
val: 'o'
key: 4
val:
key: 2
lchild
rchild
val: 'b'
key: 2
val: 'n'
key: max
val: 't'
key: 3
val: 'r'
key: max

唯一分解定理

任何一个大于1的整数N，都可以唯一分解成有限个质数的乘积

质数的筛选

- 找出小于正整数n的所有质数的个数或者找到所有小于n的质数。 有很多办法，对吧？最原始的算法，对于每一个小于n的数，判断他是不是质数(从2到平方根尝试整除)。时间复杂度O(n*sqrt(n))
- 但是怎么才能最快呢？筛！根据前面的树，直接筛选（标记）出后面不是质数的数。

埃氏筛法：一个数如果不是质数，他一定是前面一个比他小的质数的倍数，反过来某一个数不是前面所有小于他的质数的倍数，这个数就是质数。O(n*logn)

算法：初始化假设所有数都是质数，然后从前往后从2到n遍历，找到一个质数（还没有标记为合数的就是质数），就把他后面所有他的倍数（*2,*3,*4....）设置为合数（非质数）。

例如，n=13, 找小于13的所有质数

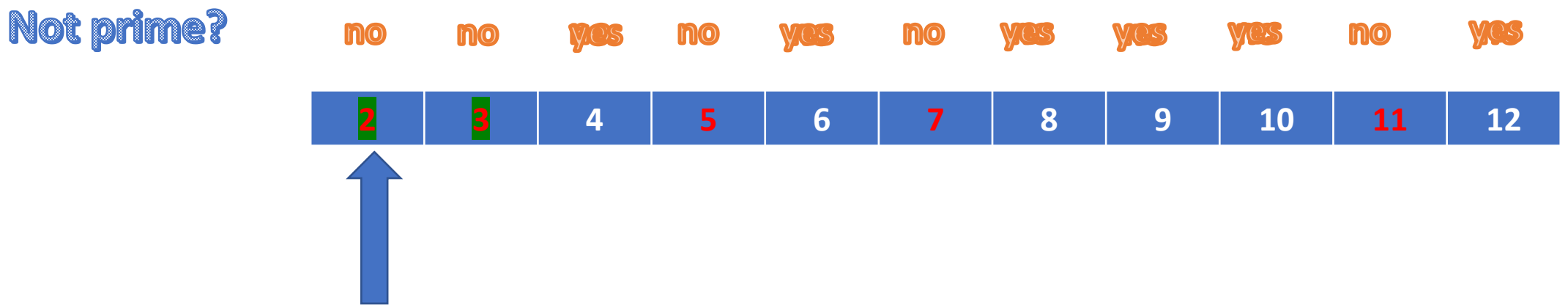
	2	3	4						10	11	12
i=2			合数						合数		合数
i=3											合数
i=4											
i=5									合数		
i=6											
i=7											
i=8,9,1 ,11,12											

重复：例如数字6,12，被2和3都筛了一次；

埃氏筛法优化：时间复杂度还是O(n*logn)，但是快很多。

算法：初始化假设所有数都是质数，然后从前往后从2到**n的平方根**遍历，找到一个质数，就把他后面所有的倍数(优化：**从这个质数的平方开始**)设置为非质数。

例如，n=13, 找小于13的所有质数



2是质数，4不是，n的平方根了

3是质数，4不是，6不是，3是n的平方根了

2*3，2*循环终止，质数时候查表，

所有的非质数都修改了，

剩下的就是质数了

程序

```
#include <bits/stdc++.h>
using namespace std;
bool notprime[5000001]; //notprime[i]为0表示i是质数
int main()
{
        int n;
        cin>>n;
        for (int i = 2; i * i <= n; i++) {
        //从2到sqrt(n)挨个处理
         if (notprime[i]) continue;//不是质数，跳过
         for (int j = i * i; j <=n; j += i) {
          //如果i是质数，把i的倍数从i*i开始全部设置为不是质数
            notprime[j] = true;
         }
        }

        int count = 0;//统计质数个数
        for (int i = 2; i <n; i++)   if (!notprime[i]) count++;
        cout<<count<<endl;
        return 0;
}
```

还可以优化么？

- 前面还是有重复：比如对于一个数30，可分解为2*15或者3*10或者5*6，显然，当循环,2,3,5时都会筛除一次30这个数，而当n很大时，就会出现许多的冗余操作。怎么办？

- 线性筛法： 对于每一个数（无论质数合数）i，筛掉所有小于等于i的最小质因子的质数乘以i的数。比如对于77,它分解质因数是7*11，那么筛掉所有小于等于7的质数*77，筛掉2*77、3*77、5*77，7*77。每次只要筛选小于等于i的第一个(小的那个)质因子的质数与i的乘积，既不会造成重复筛选，又不会遗漏， 时间复杂度O(N)。

线性筛法： 对于每一个数（无论质数合数）i，筛掉（标记为合数）所有小于等于i的最小质因子的质数乘以i的数。

例如， n=13, 找小于13的所有质数

	2	3	4	5	6	7	8	9	10	11	12
i=2			合数								
i=3					合数			合数			
i=4							合数				
i=5									合数		
i=6											合数
i=7,8,9,10,11,12											

对于i，需要维护一个小于等于i的质数的数组。所以对于每一个i，如果i是质数就先插入这个数组。

写
记下来

```
#include <bits/stdc++.h>
using namespace std;
int n;
int notprime[5000001] = {0};//notprime[i]为0表示i是质数
vector<int> primelist;//存放当前找到的所有质数
int main(){
   cin>>n;
   for (int i = 2 ; i < n ; i++)   {
      if (!notprime[i])//如果前面没有筛到，说明是质数
         primelist.push_back(i);

      //下面筛选：小于等于i的第一个质因数的所有质数*i为合数
      for (int j = 0 ; j < primelist.size() && i*primelist[j] < n ; j++)    {
         notprime[i*primelist[j]] = 1;//筛掉一个质数*i
         if (i % primelist[j] == 0)
            break; //当一个质数是i的最小的质数因子的时侯，不再筛，留给后面的筛
      }
   }

    cout<<primelist.size()<<endl;//输出质数的个数，就是primelist的大小
    return 0;
}
```

下节课预习

- 不需要做，争取理解题目，思考一下可能的算法
- [CSP-J2020] 方格取数

作业1

给你一个字符串ABCACCDAEAE，根据其中大写字母出现的次数（频率），尝试手工构造（画）一个huffman树，并给出每一个大写字母的huffman编码。注意：答案不唯一。

当然，如果可以，你可以参考课堂例子，试着自己实现一个huffman的程序，用程序构造树，输出01字符串表示编码。

作业2
线性筛法
xianxin.cpp

请根据课堂讲解的几种质数的求法，理解每一种思路，自己默写出线性筛法的程序。并自己和课堂例子进行对比。

找出小于正整数n的所有质数的个数。
$0=<n<=5*10^6$ 。

示例输入：5000000

示例输出：348513

复习作业3：回文数转换

tranpal.cpp

要求用文件输入输出

一个十进制数，每一次把这个数倒过来和这个数相加，算作一次操作，看多少次可以变成回文数字。比如给定67，第一次 67+76=143， 第二次143+341=384，操作2次变成回文数。

先给出一个整数（小于10的9次方），求这个整数转换为K进制（2<=K<=10）之后按照上述规则转换几次变成回文数。如果经过35次（包含35）还没有变成回文数字，则输出impossible

输入格式（文件名tranpal.in）：

一行两个空格分离的整数N和K，分别为给定整数以及要求的进制。

输出格式（文件名tranpal.out）：

一个整数，表示变成回文数所需要的操作次数，如果35次还没有转换为回文数字，输出”impossible”.

示例输入：

4 2

示例输出：

1

示例分析

4的二进制形式为100， 100+001=101 经过1次转换。

复习作业4: 完数 wanshu.cpp

- 一个数如果恰好等于它的所有的真因子（即除了自身以外的约数）的和，这个数就称为“完数”。例如6=1＋2＋3.找出小于n（1=<n<=1000）的所有完数。
- 输入格式：一行，一个整数n （1=<n<=1000）。
- 输出格式：按照从小到大的顺序输出小于n的所有完数，空格分开。
- 示例输入:
- 100
- 示例输出:
- 6 28

由易到难，思维体系训练
实战结合，创新协作培养
兴趣导向，未来职业引领

https://www.35tang.com

https://www.三五堂.com

扫码关注公众号

添加辅导老师