# 基本算法

## 暴搜

### 全排列

全排列是可以不用写dfs的

do {  
 for (int num : sequence) {  
 std::cout << num << " ";  
 }  
 std::cout << std::endl;  
} while (std::next\_permutation(sequence.begin(), sequence.end()));

## 贪心

贪心算法只关注当前问题的最优解，并不关注全局。在一些经典算法中有贪心算法的体现，比如Dijkstra算法以及Kruskal算法。

根据这样的性质，要求贪心法解决的问题有“无后效性”——当前的决策不会影响到后续的决策。

一般来说，对于贪心问题，需要抓住两个方面：

* 找到合适的“贪心标准”，如区间安排问题中找最小的结束时间；
* 进行恰当的排序，如Kruskal算法中对边权尽行排序。

## Huffman编码

一套正确的不定长编码一定是互不为前缀的，我们把这样的编码成为前缀码。那么，如果构建一棵树，每个节点连向左右儿子的边分别赋值为0及1，那么从根到每个叶子的路径即可表示一个编码，包括n个叶子的一棵树就能构成一套前缀码。

霍夫曼Huffman编码是基于贪心的压缩空间方法构建前缀码的，具体思想为将更短的编码赋给出现频率高的字串。构建Huffman树满足以下两个性质：

* 贪心选择性质：第一步贪心法选择保留最优解。
* 最优子结构性质：原问题的最优解包含子问题的最最优解。

基于这两个性质，算法实现如下

1. 将所有字串按照出现频率排成表P，存入队列中，记为队列P；
2. 将两个频率最低的节点取出并合并，**即指向同一个根节点**；将其根节点视为新元素存入新队列Q中，新节点的频率为两个节点的频率和；
3. 每次比较Q和P的队首，将最小的两个元素取出合并，放入P的队尾。

复杂度主要在排序上，是O(nlogn)的

上述方法的Huffman树是二叉的，但是NOI2015荷马史诗中要求构造k叉Huffman树。对于k叉的情况，只需要维护一个堆，每次取最小的k个节点合并即可。

## 二分法

### 二分答案

以P3853为例

while (left <= right) {  
 int mid = (left + right)/2;  
 if (ok(mid)) ans = mid, right = mid - 1;  
 else left = mid + 1;  
}

二分的是答案，而ok中判断的是该答案是否可行

### 二分查找（STL）

lower\_bound(a, a+n, x); //开始指针，结束指针，查找变量  
upper\_bound(a, a+n, x);

两个函数都要求a是升序排列的，lower\_bound()返回≥x的第一个位置，而upper\_bound()返回第一个>x的位置。

值得注意的是，如果是STL中的数据结构（尤其是vector），那么开始和结束的位置可以是两个迭代器，比如：

vector<int> a;  
//...  
lower\_bound(a.begin(), a.end(), x);

map内置了二分查找方法，可以通过.lower\_bound()来查找相应的的键

auto it = myMap.lower\_bound(3);  
  
if (it != myMap.end()) {  
 // 找到了  
} else {  
 // 没找到  
}

## 差分

可以看做前缀和的逆操作，数学上，差分可以看做离散的导数。

简单定义：记录上一个数和这个数的差值（因为数组里相邻数的间隔是一定的）

那显然，差分数组的前缀和就是原来的数组。由此优秀的性质，**可以地进行区间修改，但是查询是，因此适用于离线的数据修改**

裸的差分考察的很少，更多的是作为辅助，如NOIP2012 day2 T2借教室（P1083），就是二分答案然后差分维护。

## 排序

### 归并排序

基本应用：求逆序对

在归并的时候，如果右边区间的某个数放到了左边，那说明左边区间剩余的所有数都会产生逆序对。

下面给出py代码

def merge\_sort(arr):  
 if len(arr) <= 1:  
 return arr, 0  
   
 mid = len(arr) // 2  
 left, left\_count = merge\_sort(arr[:mid])  
 right, right\_count = merge\_sort(arr[mid:])  
 merged, merge\_count = merge(left, right)  
   
 return merged, left\_count + right\_count + merge\_count  
  
def merge(left, right):  
 result = []  
 count = 0  
 i = j = 0  
   
 while i < len(left) and j < len(right):  
 if left[i] <= right[j]:  
 result.append(left[i])  
 i += 1  
 else:  
 result.append(right[j])  
 j += 1  
 count += len(left) - i  
   
 result.extend(left[i:])  
 result.extend(right[j:])  
   
 return result, count  
   
n = int(input())  
arr = input().split()  
arr = [int(num) for num in arr]  
sorted\_arr, ans = merge\_sort(arr)  
print(ans)