# 2. 基础算法

## 2.1. 二分

## 2.2. 排序

### 2.2.1. 冒泡排序

Description:  
冒泡排序，年轻人的第一款排序算法  
“弟中之弟。” 复杂度:

void bubbleSort(int a[], int n) {  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < n - i - 1; j++) {  
 if (a[j] > a[j + 1]) {  
 swap(a[j], a[j + 1]);  
 }  
 }  
 }  
}

### 2.2.2. 选择排序

Description:  
复杂度: 不稳定，比冒泡强

void selectionSort(int a[], int n) {  
 for (int i = 0; i < n; i++) {  
 int min = i;  
 for (int j = i + 1; j < n; j++) {  
 if (a[j] < a[min]) {  
 min = j;  
 }  
 }  
 swap(a[i], a[min]);  
 }  
}

### 2.2.3. 插入排序

Description:  
复杂度: 比较稳定

void insertionSort(int a[], int n) {  
 for (int i = 1; i < n; i++) {  
 int j = i;  
 while (j > 0 && a[j] < a[j - 1]) {  
 swap(a[j], a[j - 1]);  
 j--;  
 }  
 }  
}

### 2.2.4. 希尔排序

Description:  
复杂度:不知道，十大经典排序之一  
缩小增量排序法，是插入排序的一种改进版本。

void shellSort(int a[], int n) {  
 int gap = n / 2;  
 while (gap > 0) {  
 for (int i = gap; i < n; i++) {  
 int j = i;  
 while (j >= gap && a[j] < a[j - gap]) {  
 swap(a[j], a[j - gap]);  
 j -= gap;  
 }  
 }  
 gap /= 2;  
 }  
}

### 2.2.5. 归并排序

Description:  
复杂度:  
快速计算逆序对，有用

int a[MaxN];  
void mSort(int \*Data, int l, int r){  
 if(l>=r) return;  
 int mid=l+r>>1;  
 mSort(Data,l,mid), mSort(Data,mid+1,r);  
 int i=l, j=mid+1;  
 int t=0;  
 int tmp[r-l+1];  
 while((i<=mid)&&(j<=r))  
 if(Data[i]<=Data[j]) tmp[t++]=Data[i++];  
 else tmp[t++]=Data[j++];   
 while(i<=mid) tmp[t++]=Data[i++];  
 while(j<=r) tmp[t++]=Data[j++];  
 for(t=0,i=l;i<=r;t++,i++) Data[i]=tmp[t];  
}  
  
int main(){  
 int n;  
 scanf("%d",&n);  
 for(int i=1;i<=n;i++) scanf("%d",&a[i]);  
 mSort(a,1,n);  
 for(int i=1;i<=n;i++) printf("%d ",a[i]);  
 putchar('\n');  
}

### 2.2.6. 堆排序

Description:  
复杂度:  
参照插入排序

### 2.2.7. 基数排序

Description:  
复杂度:，其中为所采取的基数，而为堆数  
比比较排序快，不使用比较的排序算法，占用空间较大  
模数为256时可以恰好使用CPU缓存可以应对卡常 [P4604 [WC2017]挑战](https://www.luogu.com.cn/problem/P4604)

const int N = 100010;  
const int W = 100010;  
const int K = 100;  
  
int n, w[K], k, cnt[W];  
  
struct Element {  
 int key[K];  
  
 bool operator<(const Element& y) const {  
 // 两个元素的比较流程  
 for (int i = 1; i <= k; ++i) {  
 if (key[i] == y.key[i]) continue;  
 return key[i] < y.key[i];  
 }  
 return false;  
 }  
} a[N], b[N];  
  
void counting\_sort(int p) {  
 memset(cnt, 0, sizeof(cnt));  
 for (int i = 1; i <= n; ++i) ++cnt[a[i].key[p]];  
 for (int i = 1; i <= w[p]; ++i) cnt[i] += cnt[i - 1];  
 // 为保证排序的稳定性，此处循环i应从n到1  
 // 即当两元素关键字的值相同时，原先排在后面的元素在排序后仍应排在后面  
 for (int i = n; i >= 1; --i) b[cnt[a[i].key[p]]--] = a[i];  
 memcpy(a, b, sizeof(a));  
}  
  
void radix\_sort() {  
 for (int i = k; i >= 1; --i) {  
 // 借助计数排序完成对关键字的排序  
 counting\_sort(i);  
 }  
}

### 2.2.8. 桶排序

Description:  
复杂度:   
效率取决于值域，很好用 等价于基数排序模数为1的情况

void bucketSort(int a[], int n) {  
 int bucket[MaxN];  
 for (int i = 0; i < n; i++) {  
 bucket[a[i]]++;  
 }  
 for (int i = 0, j = 0; j < n; i++) {  
 for (int k = 0; k < bucket[i]; k++) {  
 a[j++] = i;  
 }  
 }  
}

### 2.2.9. 睡眠排序

Description:  
复杂度:  
使用多线程，以数值为唤醒时间，usless,

void sleepSort(int a[], int n) {  
 vector<thread> threads;  
 for (int i = 0; i < n; i++) {  
 threads.push\_back(thread([&, i]() {  
 this\_thread::sleep\_for(chrono::milliseconds(a[i]));  
 cout << a[i] << endl;  
 }));  
 }  
 for (auto &t : threads) {  
 t.join();  
 }  
}

### 2.2.10. 快速排序

Description:  
复杂度:  
算法思想重要,

void quickSort(int a[], int l, int r) {  
 if (l >= r) return;  
 int i = l, j = r;  
 int x = a[l];  
 while (i < j) {  
 while (i < j && a[j] >= x) j--;  
 if (i < j) a[i++] = a[j];  
 while (i < j && a[i] <= x) i++;  
 if (i < j) a[j--] = a[i];  
 }  
 a[i] = x;  
 quickSort(a, l, i - 1);  
 quickSort(a, i + 1, r);  
}

## 2.3. 快速幂

Description:  
复杂度  
进行幂运算，比较快,指数过大要用欧拉降幂 乘法溢出可以使用快速乘

template <typename T>  
T quickPow(T a, T b, T m) {  
 T res = 1;  
 while (b) {  
 if (b & 1) {  
 res = (res \* a) % m;  
 }  
 a = (a \* a) % m;  
 b >>= 1;  
 }  
 return res;  
}

## 2.4. 矩阵

Description:  
封装后的矩阵，拷贝慢  
方便

template <typename T>  
class Matrix {  
public:  
 Matrix(int n, int m) : n(n), m(m), data(new T[n \* m]) {}  
 Matrix(const Matrix &mat) : n(mat.n), m(mat.m), data(new T[n \* m]) {  
 memcpy(data, mat.data, sizeof(T) \* n \* m);  
 }  
 Matrix &operator=(const Matrix &mat) {  
 if (this == &mat) return \*this;  
 n = mat.n;  
 m = mat.m;  
 delete[] data;  
 data = new T[n \* m];  
 memcpy(data, mat.data, sizeof(T) \* n \* m);  
 return \*this;  
 }  
 Matrix operator\*(const Matrix &mat) {  
 Matrix res(n, mat.m);  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < mat.m; j++) {  
 for (int k = 0; k < m; k++) {  
 res.data[i \* mat.m + j] += data[i \* m + k] \* mat.data[k \* mat.m + j];  
 }  
 }  
 }  
 return res;  
 }  
 void multiply(const Matrix &mat, Matrix &res) {  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < mat.m; j++) {  
 for (int k = 0; k < m; k++) {  
 res.data[i \* mat.m + j] += data[i \* m + k] \* mat.data[k \* mat.m + j];  
 }  
 }  
 }  
 }  
 Matrix operator%(const Matrix &mat) {  
 Matrix res(n, m);  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < m; j++) {  
 res.data[i \* m + j] = data[i \* m + j] % mat.data[i \* m + j];  
 }  
 }  
 return res;  
 }  
 Matrix operator+(const Matrix &mat) {  
 Matrix res(n, m);  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < m; j++) {  
 res.data[i \* m + j] = data[i \* m + j] + mat.data[i \* m + j];  
 }  
 }  
 return res;  
 }  
 Matrix operator-(const Matrix &mat) {  
 Matrix res(n, m);  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < m; j++) {  
 res.data[i \* m + j] = data[i \* m + j] - mat.data[i \* m + j];  
 }  
 }  
 return res;  
 }  
  
 ~Matrix() {  
 delete[] data;  
  
 }  
 T &operator()(int i, int j) {  
 return data[i \* m + j];  
 }  
 const T &operator()(int i, int j) const {  
 return data[i \* m + j];  
 }  
 int getN() const {  
 return n;  
 }  
 int getM() const {  
 return m;  
 }  
private:  
 int n, m;  
 T \*data;  
};