算法基础

必考

十大经典排序

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

0、冒泡排序

> 越小的元素会经由交换慢慢“浮”到数列的顶端，第k大的元素在第k次外循环即可到达其顺序位置。
>
> 缺点：每次内循环都要比较+交换数据，swap的代价比较大。
>
> 可记录内循环总的swap次数，若为0，则可终止程序。

1、选择排序

> 首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。
>
> 第k大的元素在第k次外循环即可到达其顺序位置，每次内循环只需swap不超过1次。

2、希尔排序

> 简单插入排序的改进版
>
> 第一个突破$O(n^2)$的排序算法

1、插入排序

```
Insertion_sort(A,n) // A[1..n], 从小到大排序
    for j = 2 to n
        key = A[j]
        i = j-1
        while i>0 and A[i]>key
            A[i+1] = A[i]
            i = i-1
        A[i+1]=key // 把key放在该放的位置上
```

2、归并排序

> 时间复杂度不受输入数据影响

```
Merge(A,p,q,r)// A[p..q],A[q+1..r] 是已经分别排好序的两段
    n1 = q-p+1
    n2 = r-q
    creat L[1..n1+1],R[1..n2+1]
    for i=1 to n1
        L[i]=A[p+i-1]
    for j=1 to n2
        R[j]=A[q+j]
    L[n1+1]=inf, R[n2+1]=inf // max值哨兵
    i=1,j=1 // index
    for k=p to r
        if(L[i]<=R[j])
            A[k]=L[i++]
        else
            A[k]=R[j++]

Merge_sort(A,p,r)
    if p<r
        q = (p+r)/2
        Merge_sort(A,p,q)
        Merge_sort(A,q+1,r)
        Merge(A,p,q,r)
```

3、快速排序

```
Partition(A,p,r)
    x = A[r] // 选中pivot，放在末尾
    i = p-1 // 当前比pivot小的标记点
    for j=p to r-1
        if A[j]<=x
            then i = i+1
                 swap(A[i],A[j])
    swap(A[i+1],A[r]) // 换pivot至中间位置
    return i+1 // 返回pivot位置

Quick_sort(A,p,r)
    if p<r
        then q = Partition(A,p,r)
             Quick_sort(A,p,q-1)
             Quick_sort(A,q+1,r)
```

计数排序

> count[num]进行记录，并反向输出

桶排序

> 计数排序的升级版，利用函数映射，count[num1,num2,...,numk]，其中f(num1)=f(num2)=f(numk)，对桶进行排序，然后拼接

基数排序

> 融合了桶排序的思想
>
> 基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。

平均情况往往和最坏情况一样差

排序算法	Best Time	Ave Time	Worst Time	Best Mem	Ave Mem	Worst Mem	稳定性
插入排序 摸牌过程	输入顺序数组 O(n)	O(n)	输入反序数组 O(n^2)				稳定
归并排序 分而治之	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	$O(n)$	$O(n)$	稳定
堆排序 优先队列	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	$O(1)$	$O(1)$	不稳定
快速排序 以轴分之	$O(n\log n)$	$O(n\log n)$	选中最值为轴 O(n^2)	$O(n\log n)$	$O(1)$	$O(n)$ 递归深度为n	不稳定

RAM模型

指令一步一步执行，无并发操作。

1、指令

- 算术指令：add, subtract, multiply, divide, remainder（取余）, floor（向下取整）, ceiling（向上取整）
- 逻辑指令: and, or, large, less, equal, not equal
- 数据移动指令：load, store, copy
- 控制指令：conditional and unconditional branch, subroutine（子程序） call and return

2、数据类型

- 整数型和浮点实数型

渐近分析

$\mathrm{O}, \Theta, \Omega$：Look at **growth** of T(n) as n →∞.

1、Θ记号 渐进紧确界

$\Theta(g(n))=\ \{f(n): \exists c_1, c_2, n_0>0, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

$f(n)=\Theta(g(n))$, 即$g(n)$是$f(n)$的渐进紧确界

理解：删除低阶项；忽略常数系数。

函数的上界和下界

2、O记号 渐进上界

$\mathrm{O}(g(n))=f(n):$ 存在正常量c和n_0, 使得对于所有$n \geq n_0$，有$0 \leq f(n) \leq cg(n)$

函数的渐近上界

3、Ω记号 渐进下界

$\mathrm{O}(g(n))=f(n):$ 存在正常量c和n_0, 使得对于所有$n \geq n_0$，有$0 \leq cg(n) \leq f(n)$

函数的渐近下界

4、o记号 非渐近紧确上界

$\mathrm{O}(g(n))=f(n):$ 存在正常量c和n_0, 使得对于所有$n \geq n_0$，有$0 \leq f(n) < cg(n)$

5、ω记号 非渐近紧确下界

$\mathrm{O}(g(n))=f(n):$ 存在正常量c和n_0, 使得对于所有$n \geq n_0$，有$0 \leq cg(n) < f(n)$

传递性：

$$f(n)=\Theta(g(n)) \text{ 且 } g(n)=\Theta(h(n)) \quad \text{蕴涵 } f(n)=\Theta(h(n))$$
$$f(n)=O(g(n)) \text{ 且 } g(n)=O(h(n)) \quad \text{蕴涵 } f(n)=O(h(n))$$
$$f(n)=\Omega(g(n)) \text{ 且 } g(n)=\Omega(h(n)) \quad \text{蕴涵 } f(n)=\Omega(h(n))$$
$$f(n)=o(g(n)) \text{ 且 } g(n)=o(h(n)) \quad \text{蕴涵 } f(n)=o(h(n))$$
$$f(n)=\omega(g(n)) \text{ 且 } g(n)=\omega(h(n)) \quad \text{蕴涵 } f(n)=\omega(h(n))$$

自反性：

$$f(n)=\Theta(f(n))$$
$$f(n)=O(f(n))$$
$$f(n)=\Omega(f(n))$$

对称性：

$$f(n)=\Theta(g(n)) \text{ 当且仅当 } g(n)=\Theta(f(n))$$

转置对称性：

$$f(n)=O(g(n)) \text{ 当且仅当 } g(n)=\Omega(f(n))$$
$$f(n)=o(g(n)) \text{ 当且仅当 } g(n)=\omega(f(n))$$

$$f(n)=O(g(n)) \text{ 类似于 } a\leqslant b$$
$$f(n)=\Omega(g(n)) \text{ 类似于 } a\geqslant b$$
$$f(n)=\Theta(g(n)) \text{ 类似于 } a=b$$
$$f(n)=o(g(n)) \text{ 类似于 } a<b$$
$$f(n)=\omega(g(n)) \text{ 类似于 } a>b$$

分治模型

- 分解：原问题==>规模较小的子问题
- 解决：递归求解子问题，当子问题规模足够小便直接求解
- 合并

例：归并算法时间复杂度求解

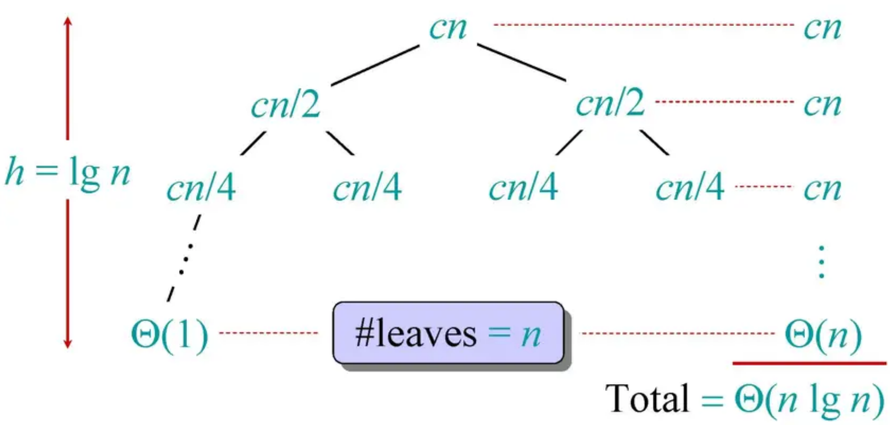
$T(n)$		**MERGE-SORT** $A[1 \,..\, n]$
$\Theta(1)$		1. If $n = 1$, done.
$2T(n/2)$	***Abuse***	2. Recursively sort $A[1 \,..\, \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \,..\, n]$.
$\Theta(n)$		3. ***"Merge"*** the 2 sorted lists

Sloppiness: Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS and Lecture 2 provide several ways to find a good upper bound on $T(n)$.

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



贪心算法

找零钱问题（change making）

固定面额值{1，5，10，25，50，100}，数量不限，以最少的货币数量进行找零

> 特殊背包问题
>
> 背包容量：找零的数值
>
> 背包要装的东西：零钱面额值
>
> 数量：无限

贪心算法：先找最大的。贪心算法

礼品分组

https://blog.csdn.net/Xuuuuuuuuuuu/article/details/106885521

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int w;
    cin>>w;
    int n;
    cin>>n;
    int a[100000];
    for(int i=1;i<=n;i++) cin>>a[i];
    sort(a+1,a+1+n);
    int begin=1,end=n,count=0;
    while(begin<end){
        if((a[end]+a[begin])<=w){
            begin++;
            end--;
            count++;
        }else{
            end--;
            count++;
        }
    }
    if(begin==end)  count++;  // 容易忽略，最后退出循环源自while中的else
    cout<<count;
}
```