C++常用函数
C++哈希表
C++列表
C++栈
C++队列

常见算法

贪心、动态规划、二分查找、KMP、广度优先搜索、宽度优先搜索、狄克斯特拉算法

查看提示信息

```
>>help(func)
```

易错点及提示

1.`split()`使用时注意可能存在连续多个空格的情况

2.`RuntimeError`考虑使用`sys.setrecrusionlimit(layor)`避免爆栈

3.不要默认最大值初始是0，有可能所有数据都小于零，最后输出0导致WA

4.常见的两种动态规划：线性动态规划和矩阵动态规划

5.线性动态规划：每点存两个值，每次调用上一个点；每次调用上两个点

6.注意浮点数精度问题

7.候选人追踪$k=314159$等特殊情况（边界情况）

8.矩阵行数列数区分好

9.可以考虑打表

10.多组数据不能使用exit()

11.不能对空列表进行某些操作，如`min`，`max`等

12.考虑统一输出节省时间

位运算

符号	描述	运算规则
&	与	两个位都为1时，结果才为1
|	或	两个位都为0时，结果才为0
^	异或	两个位相同为0，相异为1
~	取反	0变1，1变0

符号	描述	运算规则
<<	左移	各二进位全部左移若干位，高位丢弃，低位补0
>>	右移	各二进位全部右移若干位，对无符号数，高位补0，有符号数，各编译器处理方法不一样，有的补符号位（算术右移），有的补0（逻辑右移）

与运算的用途：

清零

如果想将一个单元清零，即使其全部二进制位为0，只要与一个各位都为零的数值相与，结果为零。

取一个数的指定位

比如取数 `X=1010 1110` 的低4位，只需要另找一个数Y，令Y的低4位为1，其余位为0，即 `Y=0000 1111`，然后将X与Y进行按位与运算 `X&Y=0000 1110` 即可得到X的指定位。

判断奇偶

只要根据最未位是0还是1来决定，为0就是偶数，为1就是奇数。因此可以用 `if ((a & 1) == 0)` 来判断a是不是偶数。

小技巧

1.进制转换：2进制 `bin()` 输出0b...，八进制 `oct()` 输出0o...，十六进制 `hex()` 输出0x...，`int(str,base)` 可以把字符串按照指定进制转为十进制默认base=10

2.format：

```
print('{:.2f}'.format(num))
```

3.字符串匹配等

```
iterable.count(value)
str.find(sub)       #未找到抛出-1
list.index(x)       #未找到抛出ValueError
```

4.使用 `try` + `except` 判断错误类型，辅助处理RE问题

5.math库

```
math.pow(x, y) == x**y
math.factorial(n) == n!
```

6. `ord()` 把字符变为ASCII， `chr()` 把ASCII变为字符

7.年份

```
import calendar
calendar.isleap(year)    #返回T/F判断闰年
```

8.旋转矩阵

```
for a1, a2, ..., am in zip(b1, b2, ..., bn)
```

9.排列组合

```
from itertools import permutations,combinations
permutations(list)          #生成list的全排列（每个以元组形式存在）
combinations(list,k)        #生成list的k元组合（无序）（每个以元组形式存在）
```

树状数组

```
def lowbit(x):
    return x & -x


def query(x, y):            #查询[x, y]，索引从1开始
    x -= 1
    ans = 0
    while y > x:
        ans += tr[y]
        y -= lowbit(y)
    while x > y:
        ans -= tr[x]
        x -= lowbit(x)
    return ans


def add(i, k):              #原数组第i个数加上k，更新树状数组
    while i <= n:
        tr[i] += k
        x += lowbit(i)


tr = [0] * (n + 1)
for i in range(1, n+1):     #O(nlogn)建树
    add(i, ls[i - 1])
```

排队

```
n, d = map(int, input().split())
h = [int(input()) for _ in range(n)]
out = {i: False for i in range(n)}
ans = []
while len(ans) < n:
    i = 0
    new_out = []
    minh = 0
    maxh = float('inf')
    while i < n:
        if out[i]:
            i += 1
            continue
```

```
        if len(new_out) == 0:
            new_out.append(h[i])
            minh = maxh = h[i]
            out[i] = True
            i += 1
            continue
        maxh = max(h[i], maxh)
        minh = min(h[i], minh)
        if maxh-h[i] <= d and h[i]-minh <= d:
            new_out.append(h[i])
            out[i] = True
        i += 1
    ans += sorted(new_out)
print('\n'.join([str(i) for i in ans]))
```

排列

```
from heapq import heapify, heappop, heappush

n, k = int(input()), int(input())
ls = list(map(int, input().split()))
for i in range(k):
    found = False
    j = n - 2
    heap = [ls[-1]]
    heapify(heap)
    for j in range(n - 2, -1, -1):
        heappush(heap, ls[j])
        if ls[j] < ls[j + 1]:
            found = True
            p = ls[j]
            break
    if found:
        ocp = False
        idx = j
        j += 1
        while j < n:
            t = heappop(heap)
            if not ocp and t > p:
                ls[idx] = t
                ocp = True
                continue
            ls[j] = t
            j += 1
        if not ocp:
            ls[idx] = heappop(heap)
    else:
        ls = [i for i in range(1, n + 1)]
print(' '.join([str(i) for i in ls]))
```

欧拉筛

```
lim = LIMIT
nums = {i: 1 for i in range(2, lim + 1)}
primes = []

for i in range(2, max_sqrt + 1):
    if nums[i]:
        primes.append(i)
    for j in primes:
        if i*j > lim:        #大于边界值时停止
            break
        nums[i*j] = 0
        if i % j == 0:       #保证每个数被最小的因数筛掉
            break
```

背包问题（DP）

```
w, n = map(int, input().split())         #背包容量和物品数量
data = [list(map(int, input().split())) for _ in range(n)]
dp = [[0]*(w + 1) for _ in range(m)]
for i in range(m):
    for j in range(1, t + 1):
        if j >= data[i][0]:
            if not i:
                dp[0][j] = data[i][1]
            else:
                dp[i][j] = max(data[i][1] + dp[i - 1][j - data[i][0]], dp[i - 1]
[j])
        else:
            dp[i][j] = dp[i - 1][j]
print(dp[-1][-1])
```

定容背包问题（DP）

```
for i in range(1, n):
    for j in range(t + 1):
        dp[i][j] = dp[i - 1][j]
        if j < v[i]:
            continue
        if dp[i - 1][j - w[i]] != -1:            #若容量j-w[i]存在定容分配，才可以更新
dp[i][j]
            dp[i][j] = max(dp[i][j], v[i] + dp[i - 1][j - v[i]])
```

分组背包问题（DP）

```
for k in range(1, ts + 1):                  #循环每一组
    for i in range(m, -1, -1):              #循环背包容量
        for j in range(1, cnt[k] + 1):      #循环该组的每一个物品
            if i >= w[t[k][j]]:             #背包容量充足
                dp[i] = max(dp[i], dp[i - w[t[k][j]]] + c[t[k][j]])
```

背包问题最优解方案数

$f_{i,j}$为在只能放前i个物品的情况下，刚好装满容量为j的背包所能达到的最大总价值，$g_{i,j}$表示对应的方案数。

转移方程：

如果$f_{i,j}=f_{i-1,j}$且$f_{i,j}\neq f_{i-1,j-w}+v$说明我们此时不选择把物品放入背包更优，方案数由$g_{i-1,j}$转移过来；如果$f_{i,j}\neq f_{i-1,j}$且$f_{i,j}=f_{i-1,j-w}+v$说明我们此时选择把物品放入背包更优，方案数由$g_{i-1,j-w}$转移过来；如果$f_{i,j}=f_{i-1,j}$且$f_{i,j}=f_{i-1,j-w}+v$说明放入或不放入都能取得最优解，方案数由$g_{i-1,j}$和$g_{i-1,j-w}$转移过来。

二进制分组

```
count = 0
n, p = [], []
t = list(map(int, input().split()))
for i in range(7):
    c = 1
    while t[i] >= c:
        t[i] -= c
        n.append(c)
        p.append(c*ls[i])
        count += 1
        c *= 2
    if t[i]:
        n.append(t[i])
        p.append(t[i] * ls[i])
        count += 1
```

充实的寒假生活——区间背包问题（DP）

```
#注：区间不可重合，若可重合应使用递推DP（记忆化DFS）
s, e, v = [], [], []
ls = []
for _ in range(int(input())):
    si, ei, vi = map(str, input().split())
    ls.append((si, ei, vi))
ls.sort(key=lambda t: t[1])          #关键步骤：对区间右端点排序
for si, ei, vi in ls:
    s.append(si)
    e.append(ei)
    v.append(vi)
dp = [[0]*46 for _ in range(count)]
for i in range(46):
    if i >= e[0]:
        dp[0][i] = v[0]
for i in range(1, count):
    for j in range(1, 46):
        dp[i][j] = max(dp[i][j - 1], dp[i - 1][j])
        if j >= e[i]:
            dp[i][j] = max(dp[i][j], dp[i - 1][s[i] - 1] + v[i])
print(dp[-1][-1])
```

最短时间BFS

```
from heapq import heappush, heappop, heapify
n, m, start, matrix = 0, 0, 0, 0


def bfs():
    visited = [[0 for _ in range(m)] for _ in range(n)]
    q = []
    heapify(q)
    heappush(q, (0, start[0], start[1]))
    visited[start[0]][start[1]] = 1
    while len(q):
        time, x, y = heappop(q)
        time += 1
        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            nx, ny = x + dx, y + dy
            if not (0 <= nx < n and 0 <= ny < m) or visited[nx][ny] or matrix[nx]
[ny] == '#':
                continue
            if matrix[nx][ny] == 'a':
                return time
            else:
                heappush(q, (time + (matrix[nx][ny] == 'x'), nx, ny))
                visited[nx][ny] = True
    return 'Impossible'


for _ in range(int(input())):
    n, m = map(int, input().split())
    matrix = [input() for _ in range(n)]
    for i in range(n):
        for j in range(m):
            if matrix[i][j] == 'r':
                start = (i, j)
                break
    print(bfs())
```

最长非增子序列（DP）

```
k = int(input())    #序列长度
h = list(map(int, input().split()))     #正向序列
h.reverse()
dp = [[0]*k for _ in range(k)]
for i in range(k):
    for j in range(i):
        if not i:
            dp[0][0] = 1
        else:
            dp[i][j] = [0, dp[j][j]][h[j] <= h[i]] + 1
    dp[i][i] = max(dp[i])
    dp[i][i] = max(1, dp[i][i])
print(max([dp[i][i] for i in range(k)]))
```

集合覆盖问题（贪心）

世界杯只因

```
n = int(input())
ls = list(map(int, input().split()))
ends = [0]*n         #通过区间左端点获取右端点
for i in range(n):
    if i - ls[i] > 0:
        ends[i - ls[i]] = i + ls[i]
    else:
        ends[0] = max(ends[0], i + ls[i])        #0处需要特殊判断
count = 1
l = r = ends[0]
for i in range(1, n):
    r = max(r, ends[i])
    if i >= l + 1:
        l = r
        count += 1
print(count)
```

Radar Installation

```
from math import sqrt

num = 1
while True:
    n, d = map(int, input().split())
    if n + d == 0:
        break
    x_y = {}
    for _ in range(n):
        x, y = map(int, input().split())
        if x in x_y:
            x_y[x] = max(x_y[x], y)
        else:
            x_y[x] = y
    n = len(x_y)
    count = 1
    x = list(sorted(x_y.keys()))
    try:
        ls = [x[i] + sqrt(d**2 - x_y[x[i]]**2) for i in range(n)]
        for i in range(2, n + 1):
            ls[-i] = min(ls[-i], ls[-i + 1])
        l = ls[0]
        for i in range(1, n):
            if (x[i] - l)**2 + x_y[x[i]]**2 - d**2 > 0.001:
                l = ls[i]
                count += 1
    except ValueError:
        count = -1
    if d < 0:
        count = -1
    print(f'Case {num}: {count}')
```

```
    num += 1
    input()
```

田忌赛马（贪心）

```
while True:
    n = int(input())
    if not n:
        break
    t = list(map(int, input().split()))
    k = list(map(int, input().split()))
    t.sort()
    k.sort()
    k_j = t_j = n - 1
    ans = k_k = t_k = 0
    for i in range(n):
        if k[k_j] < t[t_j]:
            k_j -= 1
            t_j -= 1
            ans += 1
        elif k[k_j] > t[t_j]:
            ans -= 1
            t_k += 1
            k_j -= 1
        else:
            if k[k_k] < t[t_k]:
                k_k += 1
                t_k += 1
                ans += 1
            else:
                ans -= (k[k_j] > t[t_k])
                k_j -= 1
                t_k += 1
    print(ans*200)
```

Jumping Cows（贪心）

```
n = int(input())
a = [int(input()) for _ in range(n)]
ans = 0
b = True
for i in range(n - 1):
    if b:
        if a[i] > a[i + 1]:
            ans += a[i]
            b = False
    else:
        if a[i] < a[i + 1]:
            ans -= a[i]
            b = True
if b:
    ans += a[-1]
print(ans)
```

最大连续子序列和（DP）

```
dp = [0]*n
dp[0] = a[0]
for i in range(1, n):
    dp[i] = max(dp[i-1]+ls[i], ls[i])
print(max(dp))
```

最大上升子序列和（DP）

```
n = int(input())
ls = [-1] + list(map(int, input().split()))      #已知输入数据非负，因此ls[0]可设为-1
dp = [0, ls[1]] + [0]*(n - 1)
for i in range(2, n + 1):
    for j in range(i):
        if ls[i] > ls[j]:
            dp[i] = max(dp[i], dp[j] + ls[i])
print(max(dp))
```

最长公共子序列

```
while True:
    try:
        x, y = input().split()
    except EOFError:
        break
    lx, ly = len(x), len(y)
    dp = [[0]*(ly + 1) for _ in range(lx + 1)]
    for i in range(1, lx + 1):
        for j in range(1, ly + 1):
            if x[i - 1] == y[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            dp[i][j] = max([dp[i][j], dp[i - 1][j], dp[i][j - 1]])
    print(dp[-1][-1])
```

最长回文子序列问题：A的最长回文子序列长度等于A与reversed(A)的最长公共子序列长度

KMP算法

```
for _ in range(int(input())):
    txt, pat = map(str, input().split())
    n = len(txt)
    m = len(pat)
    ne = [0]*m
    for i in range(1, m):
        if pat[ne[i - 1] + 1] == pat[i] and ne[i - 1] + 1 < i:
            ne[i] = ne[i - 1] + 1
    ne = [0] + ne
    i = j = 0
    ans = []
    while i < n:
        if txt[i] == pat[j]:
```

```
            i += 1
            j += 1
            if j == m:
                ans.append(str(i - j))
                j = min(m - 1, ne[j] + 1)
        else:
            i += (not j)
            j = ne[j]
    if ans:
        print(' '.join(ans))          #输出结果为子串首部在txt中的索引
    else:
        print('no')
```

河中跳房子（二分查找）

```
L,n,m = map(int,input().split())
rock = [0]
for i in range(n):
    rock.append(int(input()))
rock.append(L)


def check(x):
    num = 0
    now = 0
    for i in range(1, n+2):
        if rock[i] - now < x:
            num += 1
        else:
            now = rock[i]
    if num > m:
        return True
    else:
        return False


lo, hi = 0, L
while lo < hi:
    mid = (lo + hi) // 2
    if check(mid):
        hi = mid
    else:
        lo = mid + 1
print(lo-1)
```

动态规划难题

核电站

```
n, m = map(int, input().split())
a = [0] * (n + 1)
a[0] = 1
for i in range(1, n + 1):
    if i < m:
```

```
        a[i] = 2 * a[i - 1]
    elif i == m:
        a[i] = 2 * a[i - 1] - 1
    else:
        a[i] = 2 * a[i - 1] - a[i - 1 - m]
print(a[n])
```

复杂的整数划分问题

```
def divide_k(n, k):
    # dp[i][j]为将i划分为j个正整数的划分方法数5
    dp = [[0]*(k+1) for _ in range(n+1)]
    for i in range(n+1):
        dp[i][1] = 1
    for i in range(1, n+1):
        for j in range(1, k+1):
            if i >= j:
                # dp[i-1][j-1]为包含1的划分的数5
                # 若不包含1，我们对每个数-1仍为正整数，划分数5为dp[i-j][j]
                dp[i][j] = dp[i-j][j]+dp[i-1][j-1]
    return dp[n][k]


def divide_dif(n):
    # dp[i][j]表示将数字 i 划分，其中最大的数字不大于 j 的方法数5
    dp = [[0] * (n + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            # 比i大的数没用
            if i < j:
                dp[i][j] = dp[i][i]
            # 多了一种：不划分
            elif i == j:
                dp[i][j] = dp[i][j - 1] + 1
            # 用/不用j
            else:
                dp[i][j] = dp[i][j - 1] + dp[i - j][j - 1]
    return dp[n][n]


def divide_odd(n):
    # dp[i][j]整数i的划分里最大的数是j
    dp = [[0] * (n + 1) for _ in range(n + 1)]
    dp[0][0] = 1
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            if j % 2 == 0:
                dp[i][j] = dp[i][j-1]
            else:
                if i < j:
                    dp[i][j] = dp[i][i]
                elif i == j:
                    dp[i][j] = dp[i][j - 1] + 1
                # 用/不用j
                else:
```

```
                    dp[i][j] = dp[i][j - 1] + dp[i - j][j]
    return dp[n][n]


while True:
    try:
        n, k = map(int, input().split())
        print(divide_k(n, k))
        print(divide_dif(n))
        print(divide_odd(n))
    except EOFError:
        break
```

宠物小精灵之收服

```
N, M, K = map(int, input().split())
L = [[-1]*(M+1) for i in range(K+1)]
L[0][M] = N
for i in range(K):
    cost, dmg = map(int, input().split())
    for p in range(M):
        for q in range(i+1, 0, -1):
            if p+dmg <= M and L[q-1][p+dmg] != -1:
                L[q][p] = max(L[q][p], L[q-1][p+dmg]-cost)


def find():
    for i in range(K, -1, -1):
        for j in range(M, -1, -1):
            if L[i][j] != -1:
                return [str(i), str(j)]


print(' '.join(find()))
```

Sticks（DFS)

```
N = L = 0
used = []
length = []


def Dfs(R, M):
    if R==0 and M==0:
        return True
    if M==0:
        M = L
    for i in range(N):
        if used[i]==False and length[i] <= M:
            if i > 0:
                if used[i-1]==False and length[i]==length[i-1]:
                    continue # 不要在同一个位置多次尝试相同长度的木棒，剪枝1
            used[i] = True
            if (Dfs(R - 1, M - length[i])):
```

```
                return True
            else:
                used[i] = False
                # 不能仅仅通过替换最后一根木棒来达到目的，剪枝3
                # 替换第一个根棍子是没有用的，因为就算现在不用，也总会用到这根木棍，剪枝2
                if length[i]==M or M==L:
                    return False
    return False


while True:
    N = int(input())
    if N==0:
    break
    length = [int(x) for x in input().split()]
    length.sort(reverse = True) # 排序是为了从长到短拿木棒进行尝试
    totalLen = sum(length)
    for L in range(length[0], totalLen//2 + 1):
        if totalLen % L:
            continue # 不是木棒长度和的因子的长度，直接否定
        used = [False]*65
        if Dfs(N, L):
            print(L)
            break
    else:
        print(totalLen)
```

注：还有一种没有使用的剪枝是单调拼接剪枝

正则表达式

限定符：

`a?` a出现0/1次

`a*` a可以出现0/多次

`a+` a出现一次以上

`a{6}` a出现6次

`a{2, }` a出现2次以上

`a{2, 6}` a出现2-6次

匹配多个字符：

`(ab)+` ab出现一次以上

或运算：

`a (cat|dog)` 匹配 a cat or a dog

`a cat|dog` 匹配 a cat or dog

字符类：

`[abc]+` a/b/c出现一次以上 abc aabbcc

`[a-zA-Z0-9]` ABCabc123

`\[^0-9]` 匹配0-9之外的数据(包括换行符)

元字符：

`\d` 数字字符 \d+ 匹配一个以上的数字

`\D` 非数字字符

`\w` 单词字符（单词，数字，下划线，即英文字符）

`\W` 非单词字符

`\s` 空白符（包含空格和换行符）

`\S` 非空白字符

`\b` 单词的边界

`\B` 非单词的边界

`.`任意字符（不包含换行符）

`\.` 表示. 通过\进行了转义

`^`匹配行首

`$` 匹配行尾

贪婪匹配/懒惰匹配：

`<.+>` 贪婪匹配

`<.+?>` ？设置为懒惰匹配

模块引入：

```
from re import match
print('YES' if match(pattern, string) else 'NO')
```

取整函数

```
from math import ceil   #向上取整
from math import floor  #向下取整
round(num, n)   #四舍五入，小数点后最终有n位
```

enumerate快速获取索引和值

```
for index, value in enumerate(list, start)
```

补齐位数

```
str.ljust(width, string)            #右侧补充string至str长度为width
str.rjust(width, string)            #左侧补充string至str长度为width
```

堆

```
import heapq
```

队列

```
from queue import Queue

queue_obj = Queue()                 #创建Queue对象
queue_obj.put(obj)                  #从队尾加入元素
item = queue_obj.get()              #取出第一个元素
queue_size = queue_obj.qsize()      #元素个数
queue_empty = queue_obj.empty()     #若队列为空，返回True
queue_full = queue_obj.full()       #若队列已满，返回True
max_size = queue_obj.maxsize        #队列最大长度
```

栈

```
from queue import LifoQueue

stack_obj = LifoQueue()             #创建LifoQueue对象
stack_obj.put(obj)                  #压入元素
item = stack_obj.get()              #弹出元素
stack_size = stack_obj.qsize()      #元素个数
stack_empty = stack_obj.empty()     #若栈为空，返回True
stack_full = stack_obj.full()       #若栈已满，返回True
```

优先队列

```
from queue import PriorityQueue

q = PriorityQueue()                 #创建PriorityQueue对象
q.put((priority number, data))      #存入数据，其中priority number越小代表优先级越大
q.empty()                           #判断优先队列是否为空
q.get()                             #弹出优先级最高的优先级和元素（以元组的形式）
q.qsize()                           #返回优先队列的大小
```

默认值字典

```
from collections import defaultdict

dic = defaultdict(key_type)             #初始化时须指定键的类型，其余操作基本与字典相同
```

装饰器

```
from functools import lru_cache
```

二分查找

```
import bisect

index_left = bisect.bisect_left(list, vary)
#在list中查找vary的插入位置，使得插入后序列仍然保持有序,返回插入位置的索引;如果元素已存在，则返回最左边的位置
index3_right = bisect.bisect_right(list, num)
#在list中查找vary的插入位置，使得插入后序列仍然保持有序,返回插入位置的索引;如果元素已存在，则返回最右的位置
bisect.insort_left(list, vary)
#将vary的插入list，使得插入后序列仍然保持有序;如果元素已存在，则插入到最左边的位置
bisect.insort_right(list, num)
#将vary的插入list，使得插入后序列仍然保持有序;如果元素已存在，则插入到最右边的位置
```

C++常用函数

```
#include<iostream>
using namespace std;

#include<algorithm>
int max_value = max(a, b);       //求两个数的最大值
int min_value = min(a, b);       //求两个数的最小值

#include<numeric>
int sum = accumulate(begin(array), end(array), 0);       //求数组中元素的和
```

C++哈希表

```
#include<map>
map<key_type, value_type> map_object;        //定义map对象

map_object[key] = value;                     //插入键值对
value = map_object[key];                     //通过键访问值
size_t size = map_object.size();             //获取map大小

for(auto it = map_object.begin(); it != map_object.end(); it++)      //遍历map中的键值对
{
    cout << "Key:" << it->first << endl << "Value:" << it->second << endl;
    //通过it->first访问键，it->second访问值
}

auto it = map_object.find(key);              //查找键，若不存在返回map_object.end()
if (it != map_object.end())
{
    // 元素找到
}
```

```
else
{
    // 元素未找到
}

map_object.erase(key);
map_object.clear();

//如果对不存在的键进行操作，那么这个键的值会首先被赋值为默认值0
```

C++列表

```
#include<vector>
#include<iostream>
using namespace std;

vector<value_type> vector_objcet;                //定义vector对象，不初始化
vector<value_type> vector_objcet = {1, 2};       //定义vector对象，初始化

vector_objcet.push_back(value);                  //末尾插入数据
value = vector_objcet[index];                    //通过索引获取数据

size_type size = vector_objcet.size();           //获取vector大小

#include<numeric>
value_type sum = accumulate(vector_objcet.begin(), vector_objcet.end(), 0);
//使用accumulate(begin, end, initial_value)对vector进行求和，结果不包括end

it = vector_objcet.begin() + index;              //将value插入到vector中索引为index的位置
vector_objcet.insert(it, value);
```

C++栈

```
#include<iostream>
#include<stack>

stack<value_type> stack_object;          //定义stack对象
stack_object.push(value);                //压入元素
value = stack_object.pop();              //弹出元素
bool is_empty = stack_object.empty();        //若stack为空，返回true
size_t size = stack_object.size();       //获取stack大小
```

C++队列

```
#include<iostream>
#include<stack>

queue<value_type> queue_object;          //定义queue对象
queue_object.push(value);                //压入元素
value = queue_object.pop();              //弹出元素
bool is_empty = queue_object.empty();    //若queue为空，返回true
size_t size = queue_object.size();       //获取queue大小
```