Tips in programming

1. 快速幂算法

该法利用了指数的二进制分解。以计算319为例。

这样，原本需要进行19次操作的幂计算，只需次操作即可完成（3次乘法加上幂翻倍，2次纯幂翻倍）。复杂度从朴素法的O(n)降到了O(logn)。

**Code:**

int quickpow(int a,int b,int c)

{

int result=1,t=a;

while (b)

{

if (b%2) result=result\*t%c;

b>>=1; t=t\*t%c;

}

return result;

}

1. STL std::deque

双端队列容器，可以使用一个可逆/随机访问迭代器。

|  |  |
| --- | --- |
| 容器成员 | 操作 |
| std::deque<T>::size() | 返回该双端队列的长度 |
| std::deque<T>::push\_back(T val) | 将值为val的项插入队列尾部 |
| std::deque<T>::push\_front(T val) | 将值为val的项插入队列头部 |
| std::deque<T>::pop\_back() | 弹出队列尾部项 |
| std::deque<T>::pop\_front() | 弹出队列头部项 |
| std::deque<T>::begin() | 返回一个指向队列头的迭代器（也可写作front） |
| std::deque<T>::end() | 返回一个指向队列尾的迭代器（也可写作back） |
| std::deque<T>::iterator | 类型名。指向deque<T>中一个元素的随机访问迭代器 |
| std::deque<T>::iterator operator ++  std::deque<T>::iterator operator -- | 使迭代器指向下一个（上一个）元素 |

1. Eratosthenes素数筛法

一种快速寻找素数（制作素数表）的方法。当需要频繁处理素数时此法优于朴素判定法。

具体步骤如下：

1. 先尝试第一个素数2，将2的每一个倍数都标记为合数
2. 再尝试下一个质数3，将其倍数标记为合数
3. 再尝试5,7,11,13……重复以上步骤

该法的时间复杂度为O(nlog log n)，快于只使用朴素法制表的O(n1.5)

**Code:**

void make(bool np[],int p[],int &count,int n)

{

int i;

np[1]=1;

for (i=2;i\*i<=n;i++)

{

if (!np[i])

{

p[count++]=i;

for (int j=i\*i;j<=n;j+=i)

np[j]=1;

}

}

for (;i<=n;i++)

if (!np[i])

{

p[count++]=i;

}

}

1. 记忆化搜索范例

在DFS/BFS的基础上，用表f存储w的计算状态（如果需要的话，再加一个表v记录是否算过）。

**Code:** *(from luogu P1464 Function)*

int w(long long a,long long b,long long c)

{

//可以直接给出的例外值不需入表

if (a<=0 || b<=0 || c<=0) return 1;

if (a>20 || b>20 || c>20)

return w(20,20,20);

//核心1：已经算过的值就从“记忆”里找，不要再算

if (v[a][b][c]) return f[a][b][c];

//核心2：算一次就要“记忆”下来

if (a<b && b<c)

{

v[a][b][c]=1;

return f[a][b][c]=w(a,b,c-1)+w(a,b-1,c-1)-w(a,b-1,c);

}

v[a][b][c]=1;

return f[a][b][c]=w(a-1,b,c)+w(a-1,b-1,c)+w(a-1,b,c-1)-w(a-1,b-1,c-1);

}

1. 快速排序

一种排序方法。核心是分治法和选“枢轴点”：对于每一段序列，在枢轴点两侧寻找不满足“左小右大”的值并交换，再对枢轴点两侧的序列重复以上操作，直到所有层级下枢轴点两侧均为“左小右大”为止。

最好时间复杂度O(n)

最坏时间复杂度O(n­2)

平均时间复杂度O(nlogn)

**Code:**

//泛型化快速排序，支持对任意类型T排序，cmp为比较函数，x<y时返回true

template<typename T>

void qsort(T\* a,int l,int r,bool (\*cmp)(T x,T y))//应用二分思想

{

T mid=a[(l+r)/2];//中间数

int i=l,j=r;

do{

while(cmp(a[i],mid)) i++;//查找左半部分比中间数大的数

while(cmp(mid,a[j])) j--;//查找右半部分比中间数小的数

if(i<=j)//如果有一组不满足排序条件（左小右大）的数

{

//交换

T tmp=a[i];

a[i]=a[j];

a[j]=tmp;

i++;

j--;

}

}while(i<=j);//这里注意要有=

if(l<j) qsort(l,j);//递归搜索左半部分

if(i<r) qsort(i,r);//递归搜索右半部分

}

1. STL std::map

映射表容器，可以使用一个正向/反向/随机访问迭代器，所有元素均自动按字典升序排列。

|  |  |
| --- | --- |
| 容器成员/用法 | 操作 |
| std::map<T1,T2> a; | 定义一个从T1值指向T2值的映射表 |
| std::map<T1,T2>::begin() | 返回指向开头的迭代器 |
| std::map<T1,T2>::end() | 返回指向尾部的迭代器 |
| std::map<T1,T2>::insert(pair<T1,T2>(x,y)) | 插入一个T1值x到T2值y的映射 |
| std::map<T1,T2>::insert(map<T1,T2>::value\_type(x,y)) | 效果同上 |
| std::map<T1,T2> a;  //...  a[x]=y; | 访问映射表a里键值x的对应值并修改为y。  若没有该映射，就插入一个x->y的映射 |
| std::map<T1,T2>::erase(iterator i) | 删除迭代器i指向的映射 |
| std::map<T1,T2>::erase(T1 x) | 删除键值x的映射 |
| std::map<T1,T2>::count(iterator a,iterator b,T1 x) | 判断映射表中从a到b的位置是否有键值x的映射，有返回1，无返回0 |
| std::map<T1,T2>::find(T1 x) | 查找映射表中键值为x的映射，有返回一个指向位置的迭代器，无返回end() |