大作业

1131190111-唐川淇

介绍:

- Eigen 是一个优秀的 C++数学库
- Eigen 的百度百科 https://baike.baidu.com/item/Eigen/18001249
- Eigen3 的官方文档链接 https://eigen.tuxfamily.org/dox/

任务:

- 模仿 Eigen，制作高性能的数学库
- 编写测试用例，展示计算的精度和效率
- 完成本文档下方的项目说明
- 6 月 19 日 24:00 前将整个项目代码和本文档提交至钉钉群

C++程序设计大作业——Zuth

目录

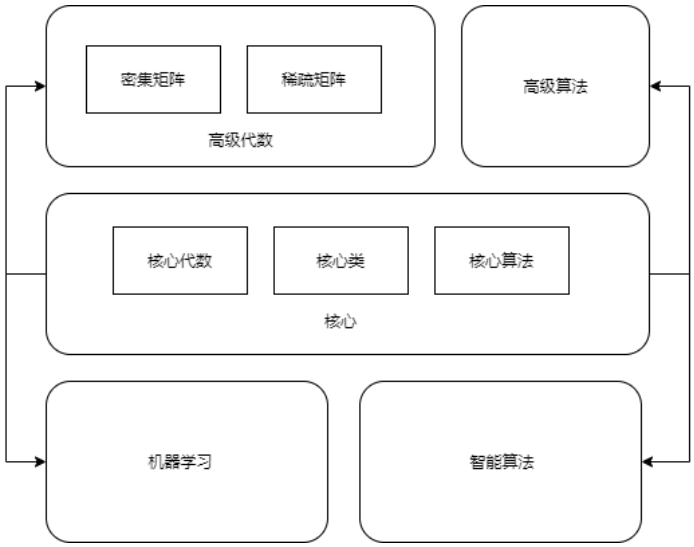
项目简介

本项目名为Zuth，为单词“Zeus”（宙斯）和“Math”（数学）的组合，寓意本项目包罗万象、功能强大。

Zuth内置了矩阵类、向量类、大数类等，可以方便的进行线性方程组求解、求积分等多种数学计算。除矩阵计算部分，Zuth对其他的一些数学问题也提供了算法，其中包括素数筛、快速幂、乘法逆元等，这些算法对于解决特定的数学问题有着非常高的效率。Zuth也有大量的计算机常用算法，包括排序、前缀和、差分等。对于优化问题，Zuth也内置了一些智能算法，为这些算法为NP完全问题提供了一种解决方案。除此之外Zuth还有机器学习算法模块，可以为机器学习者提供封装完善的机器学习模板。

项目架构

本项目提供了一个核心和四个模块，每一部分都有特定功能。核心模块为本项目的基础，核心模块为其他模块提供了支撑，核心模块里又包含核心代数、核心类和核心算法。高级代数模块主要解决密集矩阵和系数矩阵的线性方程求解以及相关问题，另外还包含一些求积分等常用数学工具。高级算法包含一些高级数据结构例如线段树等。机器学习模块提供了机器学习模型。智能算法模块提供了解决优化问题的方法。本项目的结构图如下：



项目功能

头文件	功能	说明
CoreClass	矩阵的四则运算	
CoreClass	矩阵分块	
CoreClass	矩阵转置	
CoreClass	矩阵求逆	用伴随矩阵求逆
CoreClass	高精度计算	类似java的BigInteger
CoreAlgebra	矩阵行列式	
CoreAlgebra	筛选素数	埃拉托色尼筛、欧拉筛
CoreAlgebra	快速幂	二进制取幂
CoreAlgorithm	排序	快速排序、归并排序、堆排序、希尔排序
CoreAlgorithm	前缀和、差分	一维和二维的前缀和、差分
CoreAlgorithm	集合操作	并查集
AdvancedAlgebra	定积分	辛普森法、自适应辛普森法
AdvancedAlgebra	大组合数取模	卢卡斯定理

头文件	功能	说明
AdvancedAlgebra	常用组合数	卡特兰数、贝尔数
AdvancedAlgebra	平方根	牛顿迭代
AdvancedAlgorithm	乘法逆元	扩展欧几里得算法
AdvancedAlgorithm	线段树	
AdvancedAlgorithm	树状态数组	
Dense	高斯消元	
Dense	求解线性方程组	LU分解，QR分解
Dense	曲线拟合	最小二乘
Sparse	稀疏矩阵四则运算	
Sparse	稀疏矩阵转置	
Sparse	求解线性方程组	稀疏矩阵LU分解
MachineLearning	BP神经网络	
IntelligentAlgorithm	优化问题	爬山算法、模拟退火算法

CoreClass

Matrix

1. 矩阵类的属性

	形式
矩阵数据	`dataType* pointer`
行数	`int rowSize`
列数	`int colSize`

1. 矩阵类的构造

	形式
无参数构造	`Matrix(int rows=0,int cols=0)`
使用数组构造	`Matrix(int rows , int cols ,dataType* dataPointer)`
拷贝构造	`Matrix(const Matrix& tempMatrix)`
二维构造	`Matrix(int rows, int cols, dataType** dataPointer)`
一维构造	`Matrix(int rows, int cols, dataType* dataPointer)`

3. 矩阵类的方法

类内的方法

	形式

	形式
行数	`int getRowSize()`
列数	`int getColSize()`
取出元素	`dataType & getNum(int i, int j)`
矩阵分块	`Matrix block(int i, int j, int p, int q)`
矩阵取行	`Matrix row(int i)`
矩阵取列	`Matrix col(int i)`
逆矩阵	`Matrix adgMatrix()`
代数余子式	`Matrix modMatrix(int k, int l)`

类外定义的函数

	形式
零矩阵	`template<typename Type>Matrix<Type> Zeros(int i)`
单位矩阵	`template<typename Type>Matrix<Type> Eye(int i)`
矩阵转置	`template<class Type>Matrix<Type> Transpose(Matrix<Type>& m)`

矩阵转置

矩阵转置，需要将可以将矩阵的下三角矩阵转换到上半部分。计算时需要扫过一般都矩阵，时间复杂度为$O(N^2)$。

单位矩阵

对角为1，其余为0的方阵。

零矩阵

全部元素为0的矩阵。

伴随矩阵

在线性代数中，一个方形矩阵的伴随矩阵（adjugate matrix）是一个类似于逆矩阵的概念。如果矩阵可逆，那么它的逆矩阵和它的伴随矩阵之间只差一个系数。然而，伴随矩阵对不可逆的矩阵也有定义，并且不需要用到除法。

逆矩阵

如果矩阵可逆，则

$$A^{-1} = \frac{A^*}{|A|}$$

其中A^*是伴随矩阵，$|A|$是矩阵的行列式。

```
Matrix adgMatrix() {
    Matrix<dataType> newM(rowSize,colSize);
    for (int i = 0; i < rowSize; i++)
```

```
      for (int j = 0; j < colSize; j++)
        if((i+j)%2==0)
          newM.getNum(i, j) = Del(modMatrix(i,j));
        else
          newM.getNum(i, j) = -Del(modMatrix(i, j));
    int d = Del(newM);
    for (int i = 0; i < rowSize; i++)
      for (int j = 0; j < colSize; j++)
        newM.getNum(i, j) /= d;
    return Transpose(newM);
  }
```

4. 矩阵类重载运算符

	形式
加法(+)	`Matrix operator+(const Matrix& otherMat)`
减法(-)	`Matrix operator-(const Matrix& otherMat)`
乘法(*)	`Matrix operator*(const Matrix& otherMat)`
	`Matrix operator*(int num)`
输出流(<<)	`friend std::ostream& operator<<(std::ostream& os, Matrix& m)`

Vector

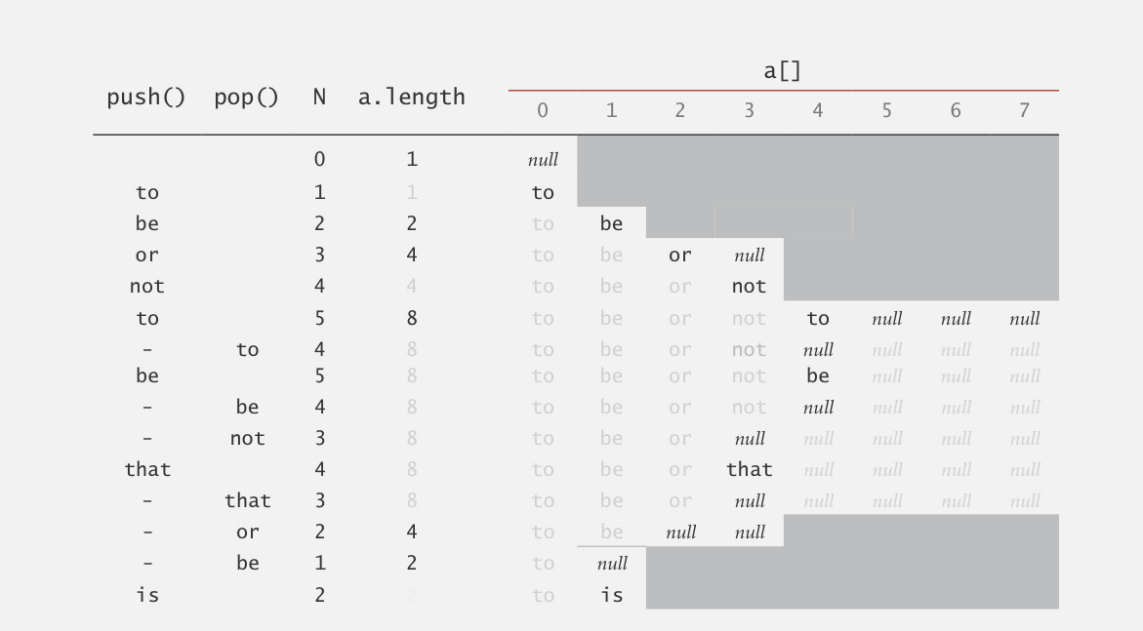
`Zuth::Vector`是变长数组，其可以动态的调节长度。首先考虑最简单的方法，假设存在一个数组，其长度为N，每次调用push函数都会在数组后添加一个元素，这时需要创建一个N+1的空数组，然后将原来数组的元素拷贝进新数组中，并且将新元素拷贝到数组中，显然这样的操作复杂度为$O(N)$的，同样的如果使用pop函数删除元素同样具有$O(N)$的时间复杂度。那么假设一共需要插入N个元素，那么总体的时间复杂度为：

$$1+2+3+\cdots+N\sim\frac{N^2}{2}$$

显然，这样的复杂度难以接受。为了优化变长数组，可以使用如下策略：

1. `push()`：当数组存满的时候将数组容量翻倍。
2. `pop()`:当数组元素数量小于等于1/4容量时，将容量减半。

改进的动态数组效果如下：



push()	pop()	N	a.length	a[] 0	1	2	3	4	5	6	7
		0	1	null							
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or	null				
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	null	null	null
-	to	4	8	to	be	or	not	null	null	null	null
be		5	8	to	be	or	not	be	null	null	null
-	be	4	8	to	be	or	not	null	null	null	null
-	not	3	8	to	be	or	null	null	null	null	null
that		4	8	to	be	or	that	null	null	null	null
-	that	3	8	to	be	or	null	null	null	null	null
-	or	2	4	to	be	null	null				
-	be	1	2	to	null						
is		2	2	to	is						

同样考虑插入N个元素的情况，使用优化后的变长数组的时间复杂度为：

$$1+2+3+\cdots+N\sim 3N$$

显然这样的复杂度要比平方级别的复杂度好得多。

1. Vector类的属性

	形式
容量	`int cap`
大小	`int size`
数据	`Type* pointer`

2. Vector类的方法

	形式
在尾部插入元素	`void pushback(const Type &data)`
移除尾部元素	`void popback()`
调整容量	`void resize(int newCap)`
重载[]	`Type & operator[](int i)`

BigNum

众所周知，C++存储的整数不能超过一定的范围，下面给出64位编译器各类型整数范围

	字节	范围
`short int`	2	-32768 ~+32767

	字节	范围
`int`	4	-2147483648~+2147483647
`long int`	4	-2147483648 ~+2147483647
`long long int`	8	-9223372036854775808~9223372036854775807

高精度计算（Arbitrary-Precision Arithmetic），也被称作大整数（bignum）计算，运用了一些算法结构来支持更大整数间的运算（数字大小超过语言内建整型）。包括Java、Python等现代编程语言都内置了对 bignums 的支持，其他语言具有可用于任意精度整数和浮点数学的库。这些实现通常使用可变长度的数字数组，而不是将值存储为与处理器寄存器大小相关的固定位数。下面在Zuth中实现BigNum类。

1. BigNum的属性

BigNum通过变长数组构成，所以属性只有一个`Zuth::Vector`，高精度数字利用字符串表示，每一个变长数组元素表示数字的一个十进制位。因此可以说，高精度数值计算实际上是一种特别的向量处理。读入字符串时，数字最高位在字符串首（下标小的位置）。但是习惯上，下标最小的位置存放的是数字的最低位，即存储反转的字符串。这么做的原因在于，数字的长度可能发生变化，但我们希望同样权值位始终保持对齐（例如，希望所有的个位都在下标`[0]`，所有的十位都在下标`[1]`……）；同时，加、减、乘的运算一般都从个位开始进行，这都给了「反转存储」以充分的理由。

2. BigNum的构造

在BigNum中我创建了四种构造函数，如下所示：

	形式
无参数构造	`BigNum()`
使用整数构造	`BigNum(const int &data)`
使用`string`构造	`BigNum(std::string str)`
使用`Vector`构造	`BigNum(Zuth::Vector<int> tempV)`

无参数的构造可以创建一个空的大数，其中没有任何数字，只有一个空的`Zuth::Vector`；使用整数的构造可以将`int`型变量转变为大数类；使用`string`类构造，可以将一个长的字符串变为大数，例如可以将字符串”9999999999“转变为大数9999999999；使用`Zuth::Vector`的构造主要在类中的重载运算符用到。

3. BigNum重载运算符

	形式
加法(+)	`BigNum operator+(BigNum& otherBN)`
减法(-)	`BigNum operator-(BigNum& otherBN)`
乘法(*)	`BigNum operator*(int otherNum)`
	`BigNum operator*(BigNum& otherBN)`
输出流(<<)	`friend std::ostream& operator<<(std::ostream& os, BigNum& b)`

高精度加法

$$\begin{array}{r} 9\ 6\ 2 \\ +\quad 9\ 3 \\ \hline 1\ 0\ 5\ 5 \end{array}$$

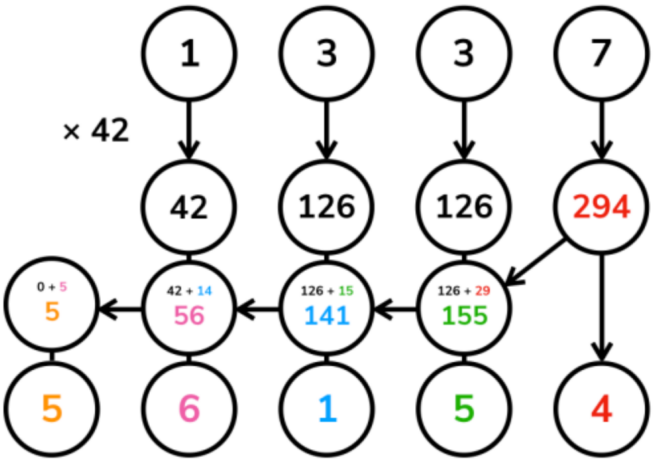
高精度加法其实就是竖式加法，从最低位开始，将两个加数对应位置上的数码相加，并判断是否达到或超过 10。如果达到10，那么处理进位：将更高一位的结果上增加 1，当前位的结果减少 10。

高精度减法

高精度减法，也就是竖式减法，具体思路和高精度加法类似，这里不做赘述。

高精度*单精度

乘数中的一个是普通的 `int` 类型。一个直观的思路是直接将a每一位上的数字乘以b。从数值上来说，这个方法是正确的，但它并不符合十进制表示法，因此需要将它重新整理成正常的样子。重整的方式，也是从个位开始逐位向上处理进位。但是这里的进位可能非常大，甚至远大于 9，因为每一位被乘上之后都可能达到 9b的数量级。所以这里的进位不能再简单地进行-10 运算，而是要通过除以10 的商以及余数计算。详见代码注释，也可以参考下图展示的一个计算高精度数 1337乘以单精度数42 的过程。

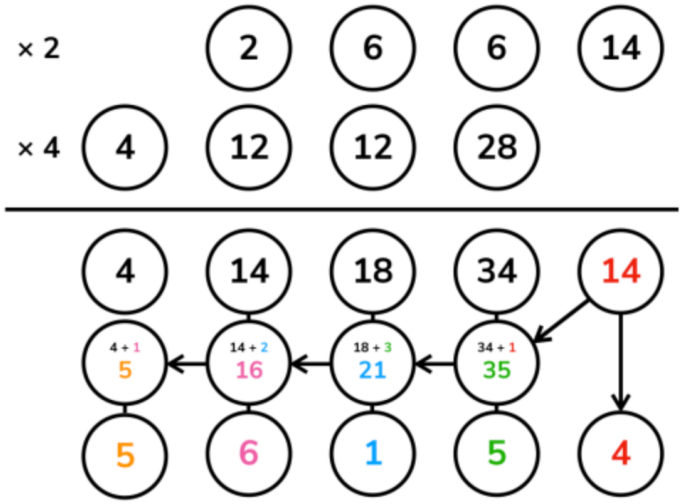


高精度*高精度

如果两个乘数都是高精度，那么可以继续使用竖式乘法。回想竖式乘法的每一步，实际上是计算了若干$a \times b_i \times 10^i$的和。例如计算 1337×42，计算的就是 $1337 \times 2 \times 10^0 +$

$1337 \times 4 \times 10^1$。

于是可以将 b分解为它的所有数码，其中每个数码都是单精度数，将它们分别与 a相乘，再向左移动到各自的位置上相加即得答案。当然，最后也需要用与上例相同的方式处理进位。



CoreAlgorithm

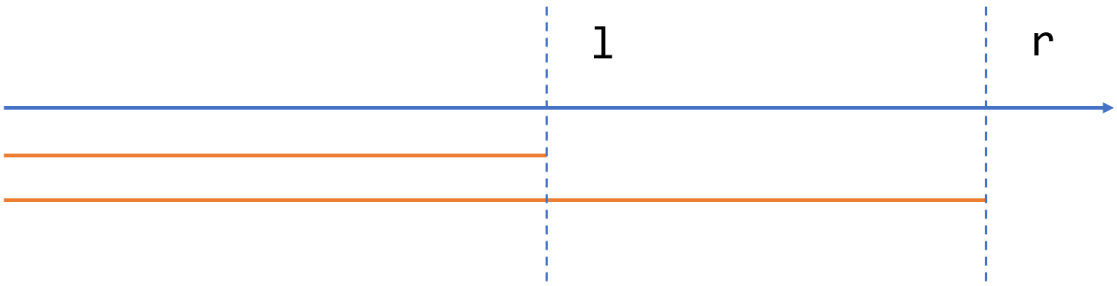
前缀和和差分

	形式
一维前缀和	`Zuth::Vector<T> Prefix(Zuth::Vector<T>& V)`
一维差分	`Zuth::Vector<T> Difference(Zuth::Vector<T> &V)`
二位前缀和	`Zuth::Matrix<T> Prefix(Zuth::Matrix<T> M)`
二维差分	`Zuth::Matrix<T> Difference(Zuth::Matrix<T> M)`

如果我们要求数组中某一段区间(L, R)的和，用普通的循环每次查询都需要从L到R循环一遍，显然每次询问的时间复杂度是$O（r-l）$，当询问次数变多是，这样的复杂度难以忍受。而前缀和就是解决这一问题的最好方法。

1. 一维前缀和

前缀和，顾名思义就是前缀的和，前缀和数组中的每一位都是原数组中从头到目前的和，也就是`S[i] = A[1] + A[1] + ... + A[i]`，其中s是前缀和数组，A是原数组。显然需要数组长度$O（N）$的时间复杂度预处理，而询问（l，r）区间和时只需要$O（1）$的复杂度。其原理如下：



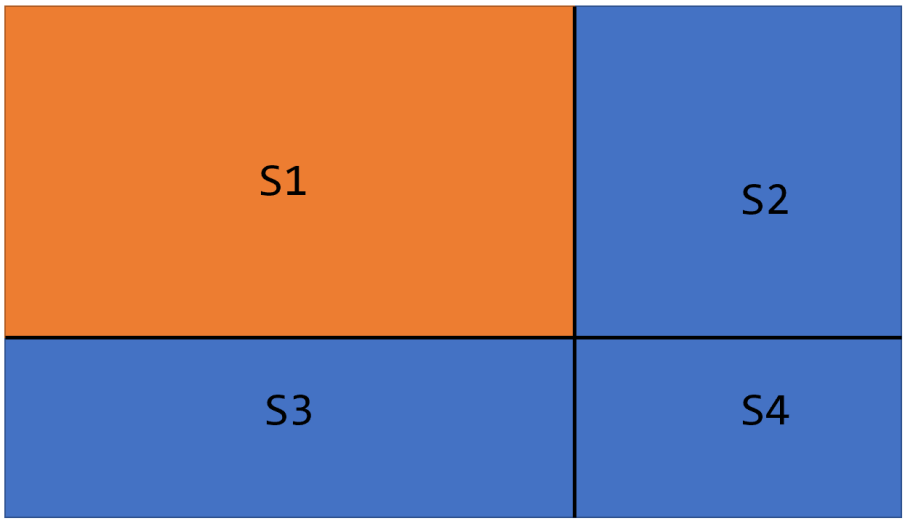
计算(l,r)区间和只需要将两橘色的线段相减，就得到了中间的部分。

2. 二维前缀和

和一维前缀和类似，使用一个元素记录到此元素之前的所有元素和，用伪代码表示为：

```
for (int i = 1; i <= n; i++)
         for (int j = 1; j <= m; j++)
                     //目前   + 上面一格      + 左边一格      - 重复部分
             S[i][j] = A[i][j] + S[i - 1][j] + S[i][j - 1] - S[i - 1][j - 1];
```

同样的，显然需要矩阵大小$O\ （N）$的时间复杂度预处理，而询问（l，r）到（x，y）区间和时只需要$O\ （1）$的复杂度。



取得S_4的面接可以用$(S_1+S_2+S_3+S_4)-(S_1+S_2)-(S_1+S_3)+S_1$表示。

3. 一维差分

考虑这样一种情况，给定一个序列，每次将数组（l，r）的范围内的所有数据进行操作，那么使用最朴素的想法，每次操作的时间复杂度为$O\ （r-l）$，当操作次数非常多的时候，显然

这样的操作并不好，而差分可以解决范围操作的问题。和前缀和类似，用伪代码表示：

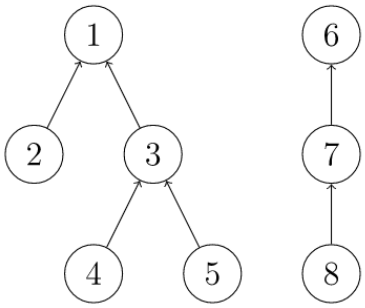
```
for (int i = 1; i <= n; i++)
    B[i] = A[i] - A[i - 1];
```

4. 二维差分

伪代码表示：

```
B[i][j] = B[i][j] - B[i - 1][j] - B[i][j - 1] + B[i - 1][j - 1];
 //        本身     - 上部分       - 左部分       + 重复部分
```

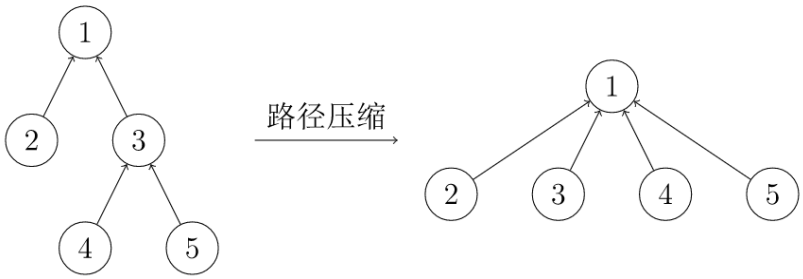
并查集



并查集是一种树形的数据结构，顾名思义，它用于处理一些不交集的合并及查询问题。 它支持两种操作：

- 查找（Find）：确定某个元素处于哪个子集；
- 合并（Union）：将两个子集合并成一个集合。

1. 查找



利用递归算法可以不断向上找到父节点，同时用到路径压缩，代码如下：

```
int findFather(int i) {
    if (f[i] == i)return i;
    else {
      f[i] = findFather(f[i]);
      return f[i];
    }
  }
```

2. 合并

直接将一棵树并到另一颗树上面，代码如下：

```
void dUnion(int i, int j) {
    int fi = findFather(i), fj = findFather(j);
    if (i != j)f[fi] = fj;
  }
```

该代码可以优化至使用秩合并，同时使用路径压缩和启发式合并之后，并查集的每个操作平均时间仅为$O(\alpha(N))$，其中$\alpha()$为阿克曼函数的反函数，其增长极其缓慢，也就是说其单次操作的平均运行时间可以认为是一个很小的常数。

排序

排序算法（Sorting algorithm）是一种将一组特定的数据按某种顺序进行排列的算法。排序算法多种多样，性质也大多不同。

	形式
快速排序	`void quickSort(T arr[], const int len)`
归并排序	`void mergeSort(T a[], int ll, int rr)`
堆排序	`void heapSort(int arr[], int len)`
希尔排序	`void shellSort(T array[], int length)`

1. 快速排序

快速排序的工作原理是通过 分治 的方式来将一个数组排序。

快速排序分为三个过程：

1. 将数列划分为两部分（要求保证相对大小关系）；
2. 递归到两个子序列中分别进行快速排序；
3. 不用合并，因为此时数列已经完全有序。

快速排序是一种不稳定的排序，最优时间复杂度和平均时间复杂度都是$O(NlogN)$，最坏情况下的时间复杂度是$O(N^2)$。

2. 归并排序

归并排序（merge sort）是一种采用了 分治 思想的排序算法。

归并排序分为三个步骤：

1. 将数列划分为两部分；
2. 递归地分别对两个子序列进行归并排序；
3. 合并两个子序列。

```
if (rr - ll <= 1) return;
  int mid = ll + ((rr - ll) >> 1);
  merge(ll, mid);
  merge(mid, rr);
  int p = ll, q = mid, s = ll;
  while (s < rr) {
    if (p >= mid || (q < rr && a[p] > a[q])) {
      t[s++] = a[q++];
      // ans += mid - p;
    } else
      t[s++] = a[p++];
  }
  for (int i = ll; i < rr; ++i) a[i] = t[i];
}
```

归并排序是一种稳定的排序算法。归并排序的最优时间复杂度$O(NlogN)$、平均时间复杂度和最坏时间复杂度均为$O(NlogN)$ 。归并排序的空间复杂度为 $O(N)$。

3. 堆排序

堆排序（Heapsort）是指利用 二叉堆 这种数据结构所设计的一种排序算法。堆排序的适用数据结构为数组。

首先建立大顶堆，然后将堆顶的元素取出，作为最大值，与数组尾部的元素交换，并维持残余堆的性质；之后将堆顶的元素取出，作为次大值，与数组倒数第二位元素交换，并维持残余堆的性质；以此类推，在第$n-1$次操作后，整个数组就完成了排序。

同选择排序一样，由于其中交换位置的操作，所以是不稳定的排序算法。时间复杂度:堆排序的最优时间复杂度、平均时间复杂度、最坏时间复杂度均为$O(NlogN)$ 。空间复杂度:由于可以在输入数组上建立堆，所以这是一个原地算法。

4. 希尔排序

希尔排序（Shell sort），也称为缩小增量排序法，是 插入排序的一种改进版本。希尔排序以它的发明者希尔（Donald Shell）命名。

排序对不相邻的记录进行比较和移动：

1. 将待排序序列分为若干子序列（每个子序列的元素在原始数组中间距相同）；
2. 对这些子序列进行插入排序；
3. 减小每个子序列中元素之间的间距，重复上述过程直至间距减少为 1。

希尔排序是一种不稳定的排序算法。

希尔排序的最优时间复杂度为$O(N)$。希尔排序的平均时间复杂度和最坏时间复杂度与间距序列的选取（就是间距如何减小到 1）有关，比如「间距每次除以 3」的希尔排序的时间复杂度是 $O(N^{3/2})$。已知最好的最坏时间复杂度为 $O(Nlog^2N)$。

CoreAlgebra

矩阵相关

矩阵的行列式

通常情况下低阶矩阵的行列式要比高阶矩阵算起来容易，因此可以很容易的想到递归的方法，低阶矩阵直接计算行列式，高阶矩阵利用公式化为低阶矩阵计算。在n阶矩阵行列$D=|a_{ij}|$中去除所在的第i行和第j列得到的n-1阶行列式为元素a_{ij}的代数余子式，记作M_{ij}。n阶矩阵的行列式可以用如下公式表示：

$$D=\sum_{k=1}^{n}a_{ik}A_{ik},i=1,2\cdots n$$

伪代码表示如下：

```
template<typename Type>
double Del(Matrix<Type>& m) {
  // 一阶直接计算
  if (m.getRowSize() == 1)return m.getNum(0, 0);
  // 二阶直接计算
  else if (m.getRowSize() == 2)
    return m.getNum(0, 0) * m.getNum(1, 1) - m.getNum(1, 0) * m.getNum(0, 1);
  // 三阶及以上递归
  else {
      // 省略其中代码
      // 代数余子式
      Matrix<Type> tM(col-1,col-1,tempM);
      if(i%2==1)
        ans -= m.getNum(0, i) * Del(tM);
      else
        ans += m.getNum(0, i) * Del(tM);
    }
    return ans;
  }
```

素数筛

	形式
埃拉托色尼筛	`Zuth::Vector<bool> eratosthenesSieve(int len)`
欧拉筛	`Zuth::Vector<int> eulerSieve(int len)`

1. 埃拉托色尼筛

素数筛法，是一种快速“筛”出2~n之间所有素数的方法。朴素的筛法叫埃氏筛（the Sieve of*Eratosthenes*，埃拉托色尼筛）。这个算法的复杂度是 $O(NloglogN)$，还是非常优秀了。其核心代码如下：

```
bool isnp[MAXN]; // is not prime: 不是素数
void init(int n)
{
    for (int i = 2; i * i <= n; i++)
        if (!isnp[i])
            for (int j = i * i; j <= n; j += i)
                isnp[j] = 1;
}
```

但是我们可能会发现，在筛的过程中我们会重复筛到同一个数，例如12同时被2和3筛到，30同时被2、3和5筛到。所以我们引入欧拉筛，也叫线性筛，可以在 $O(N)$时间内完成对2~n的筛选。它的核心思想是：让每一个合数被其最小质因数筛到。

2. 欧拉筛

欧拉筛的算法复杂度证明比较繁琐，这里不再赘述，这里只给出其伪代码：

```
bool isnp[MAXN];
vector<int> primes;// 质数表
voidinit(int n)
{
for (int i= 2; i<= n; i++)
    {
if (!isnp[i])
            primes.push_back(i);
for (int p : primes)
        {
if (p* i> n)
break;
            isnp[p* i]= 1;
if (i% p== 0)
break;
        }
    }
}
```

快速幂

快速幂，二进制取幂（Binary Exponentiation，也称平方法），是一个在$O(lonN)$的时间内计算a^n的小技巧，而暴力的计算需要$O(N)$的时间。而这个技巧也常常用在非计算的场景，因为它可以应用在任何具有结合律的运算中。其中显然的是它可以应用于模意义下取幂、矩阵幂等运算。

计算 a的 n次方表示将n个a 乘在一起。然而当 n太大的时侯，这种方法就不太适用了。二进制取幂的想法是，我们将取幂的任务按照指数的 二进制表示 来分割成更小的任务。

首先我们将n表示为 2 进制，举一个例子：

$$3^{13} = 3^{(1101)_2} = 3^8 * 3^4 * 3^2$$

这个算法的复杂度是$O(lonN)$的，我们计算了$O(lonN)$个 2^k次幂的数，然后花费$O(lonN)$的时间选择二进制为 1 对应的幂来相乘。

其核心代码如下：

```
T quickPower(T a, T b) {
  T res = 1;
  while (b > 0) {
    if (b & 1) res = res * a;
    a = a * a;
    b >>= 1;
  }
  return res;
}
```

AdvancedAlgebra

数值积分

函数$f(x)$在区间[l,r]上的定积分$\int_l^r f(x)dx$指的是$f(x)$在区间[l,r]中与x轴围成的区域面积。

1. 辛普森法

这个方法的思想是将被积区间分为若干小段，每段套用二次函数的积分公式进行计算。

$$\int_l^r f(x)dx = (f(x_0) + 4f(x_1) + 2f(x_2) + \cdots + 4f(x_{2N+1} + f(x_{2N})))\frac{h}{3}$$

2. 自适应辛普森法

普通的方法为保证精度在时间方面无疑会受到n的限制，我们应该找一种更加合适的方法。现在唯一的问题就是如何进行分段。如果段数少了计算误差就大，段数多了时间效率又会低。我们需要找到一个准确度和效率的平衡点。我们这样考虑：假如有一段图像已经很接近二次函数的话，直接带入公式求积分，得到的值精度就很高了，不需要再继续分割这一段了。于是我们有了这样一种分割方法：每次判断当前段和二次函数的相似程度，如果足够相似的话就直接代入公式计算，否则将当前段分割成左右两段递归求解。现在就剩下一个问题了：如果判断每一段和二次函数是否相似？我们把当前段直接代入公式求积分，再将当前段从中点分割成两段，把这两段再直接代入公式求积分。如果当前段的积分和分割成两段后的积分之和相差很小的话，就可以认为当前段和二次函数很相似了，不用再递归分割了。

大组合数取模

Lucas 定理用于求解大组合数取模的问题，其中模数必须为素数。正常的组合数运算可以通过递推公式求解，但当问题规模很大，而模数是一个不大的质数的时候，就不能简单地通过递推求解来得到答案，需要用到 Lucas 定理。

卢卡斯定理如下：

$$\binom{n}{m} mod\ p = \binom{\lfloor n/p \rfloor}{\lfloor m/p \rfloor} \cdot \binom{n\ mod\ p}{m\ mod\ p} mod\ p$$

时间复杂度为 $O(f(p) + g(n)log\ n)$，其中$f(x)$ 为预处理组合数的复杂度，$g(n)$为单次求组合数的复杂度。

```
long long Lucas(long long n, long long m, long long p) {
  if (m == 0) return 1;
  return (C(n % p, m % p, p) * Lucas(n / p, m / p, p)) % p;
}
```

常用组合数

	形式
卡特兰数	`int Catalan(int n)`
贝尔数	`int Bell(int n)`

1. 卡特兰数

递推公式如下：

$$H_n = \frac{\binom{2}{n}}{n+1}$$

2. 贝尔数

贝尔数以埃里克·坦普尔·贝尔命名，是组合数学中的一组整数数列。

贝尔数适合递推公式：

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

每个贝尔数都是相应的第二类斯特林数的和。 因为第二类斯特林数是把基数为n的集合划分为正好
k个非空集的方法数目。

$$B_{n+1} = \sum_{k=0}^{n} S(n,k)$$

求解平方根

牛顿迭代法的递推公式如下：

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

求解平方根问题，可以令$f(x) = x^2 - n$，方程的解就是平方根的近似解。

```
double sqrtNewton(double n) {
  const double eps = 1E-15;
  double x = 1;
  while (true) {
    double nx = (x + n / x) / 2;
    if (abs(x - nx) < eps) break;
    x = nx;
  }
  return x;
}
```

AdvancedAlgorithm

线段树

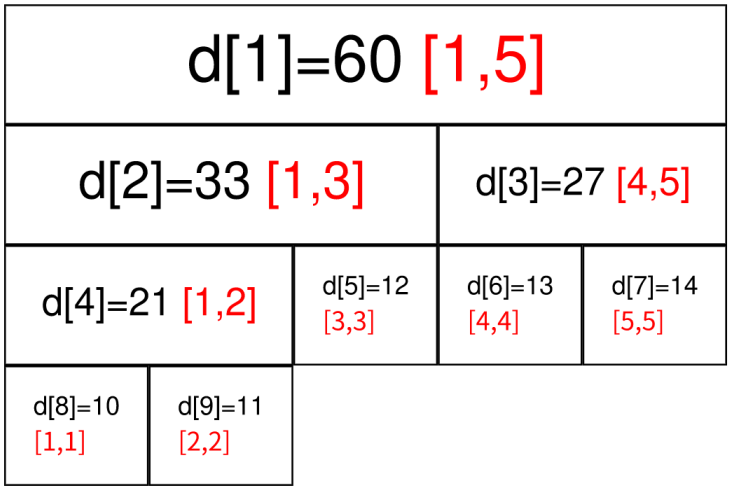
线段树是一个平衡的二叉树，它将每个长度不为1的区间划分成左右两个区间递归求解。令整个区间的长度为N，则其有N个叶节点，每个叶节点代表一个单位区间，每个内部结点代表的区间为其两个儿子代表区间的联集。这种数据结构可以方便的进行大部分的区间操作。

	形式
构造函数	`SegmentTree(int m)`
创建线段树	`void buildSegmentTree(int left, int right, int i)`
插入元素	`void insert(int data)`
清除元素	`void erase(int data)`
计算区间	`int count(int left, int right)`
内部计算区间	`int countInternal(int left, int right, int i)`

线段树将每个长度不为1的区间划分成左右两个区间递归求解，把整个线段划分为一个树形结构，通过合并左右两区间信息来求得该区间的信息。这种数据结构可以方便的进行大部分的区间操作。

有个大小为5的数组 ，要将其转化为线段树$a = \{10, 11, 12, 13, 14, 15\}$，有以下做法：设线段树的根节点编号为 1，用数组d来保存我们的线段树， 用d_i来保存线段树上编号为 i的节点的值（这里每个节点所维护的值就是这个节点所表示的区间总和）。

线段树的形态：



乘法逆元

这里介绍模意义下乘法运算的逆元（Modular Multiplicative Inverse），并介绍如何使用扩展欧几里德算法（Extended Euclidean algorithm）求解乘法逆元。

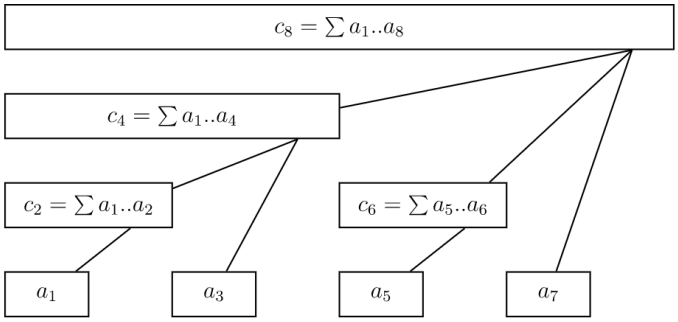
如果一个线性同余方程 $ax \equiv 1 mod(b)$，称x是a mod b的逆元，计做a^{-1}。

扩展欧几里得代码如下：

```
template<typename T>
  void exgcd(T a, T b, T& x, T& y) {
    if (b == 0) {
      x = 1, y = 0;
      return;
    }
    exgcd(b, a % b, y, x);
    y -= a / b * x;
  }
```

树状数组

树状数组和线段树具有相似的功能，但他俩毕竟还有一些区别：树状数组能有的操作，线段树一定有；线段树有的操作，树状数组不一定有。但是树状数组的代码要比线段树短，思维更清晰，速度也更快，在解决一些单点修改的问题时，树状数组是不二之选。



lowbit函数

```
int lowbit(int x) {
  // x 的二进制表示中，最低位的 1 的位置。
  // lowbit(0b10110000) == 0b00010000
  //           ~~~^~~~~
  // lowbit(0b11100100) == 0b00000100
  //           ~~~~~^~~
  return x & -x;
}
```

Dense

	形式
高斯消元	`template<typename T>Zuth::Matrix<T> gaussianElimination(Zuth::Matrix<T> m)`
LU分解	`template<typename T>Zuth::Vector<T> LUsolve(Zuth::Matrix<T> m)`
QR分解	`template<typename T>Zuth::Vector<Zuth::Matrix<T> > QR(Zuth::Matrix<T> M)`
QR分解求方程	`template<typename T>Zuth::Vector<T> QRsolve(Zuth::Matrix<T> M)`

高斯消元法

高斯消元法是线性代数中的一个算法，可以把矩阵变为行阶梯矩阵，高斯消元法可以用来求解线性方程组，求出矩阵的秩，以及求出可逆方程的逆矩阵。

高斯消元法的算法复杂度是O(*n*3)；这就是说，如果系数矩阵的是*n* × *n*，那么高斯消元法所需要的计算量大约与*n*3成比例。高斯消元法可以用在电脑中来解决数千条等式及未知数。不过，如果有过百万条等式时，这个算法会十分费时。一些极大的方程组通常会用迭代法来解决。亦有一些方法特地用来解决一些有特别排列的系数的方程组。高斯消元法可用在任何域中。高斯消元法对于一些矩阵来说是稳定的。对于普遍的矩阵来说，高斯消元法在应用上通常也是稳定的，不过亦有例外。

LU分解

LU分解在本质上是高斯消元法的一种表达形式。实质上是将A通过初等行变换变成一个上三角矩阵，其变换矩阵就是一个单位下三角矩阵。这正是所谓的杜尔里特算法（Doolittle algorithm）：从下至上地对矩阵A做初等行变换，将对角线左下方的元素变成零，然后再证明

这些行变换的效果等同于左乘一系列单位下三角矩阵，这一系列单位下三角矩阵的乘积的逆就是L矩阵，它也是一个单位下三角矩阵。这类算法的复杂度一般在$O(n^3)$左右

```
// LU分解求解
template<typename T>
Zuth::Vector<T> LU(Zuth::Matrix<T> m){
    int row = m.getRowSize();
    int col = m.getColSize();
    Zuth::Matrix<T> temp=gaussianElimination(m);
    Zuth::Vector<T> ans(col-1);
    for (int i = 0; i < col-1; i++)ans[i] = temp.getNum(i, col - 1) / temp.getNum(i,i);
    return ans;
}
```

QR分解

QR（正交三角）分解法是求一般矩阵全部特征值的最有效并广泛应用的方法，一般矩阵先经过正交相似变化成为Hessenberg矩阵，然后再应用QR方法求特征值和特征向量。它是将矩阵分解成一个正规正交矩阵Q与上三角形矩阵R，所以称为QR分解法，与此正规正交矩阵的通用符号Q有关。

QR分解的实际计算有很多方法，例如Givens旋转、Householder变换，以及Gram-Schmidt正交化等等。每一种方法都有其优点和不足。

利用QR分解可以求线性方程的解：

$$
\begin{aligned}
A \cdot x &= b \\
Q \cdot R \cdot x &= b \\
R \cdot x &= Q^T \cdot b
\end{aligned}
$$

曲线拟合

	形式
构造	`CurveFit(std::vector<double> Vx, std::vector<double> Vy, int n, int ex, double coefficient[])`
求和	`double sum(std::vector<double> Vnum, int n)`
乘积和	`double MutilSum(std::vector<double> Vx, std::vector<double> Vy, int n)`
e^x和	`double RelatePow(std::vector<double> Vx, int n, int ex)`
x的e^x次方与y的乘积的累加	`double RelateMutiXY(std::vector<double> Vx, std::vector<double> Vy, int n, int ex)`
增广矩阵	`void EMatrix(std::vector<double> Vx, std::vector<double> Vy, int n, int ex, double coefficient[])`
消元	`void CalEquation(int exp, double coefficient[])`

最小二乘法（又称最小平方法）是一种数学优化技术。它通过最小化误差的平方和寻找数据的最佳函数匹配。利用最小二乘法可以简便地求得未知的数据，并使得这些求得的数据与实际数据之间误差的平方和为最小，其在曲线拟合中应用广泛。

给定函数$y=f(x)$在点$x_1,x_2\cdots x_n$的函数值$y_1,y_2\cdots y_n$，求一多项式:

$$\sum_{i=1}^{n}(p(x_i)-y_i)^2=min$$

令：

$$S(a_0,a_1\cdots a_n)=\sum_{i=1}^{n}(a_0+a_1x_1+\cdots+a_mx_i^m-y_i)^2=min$$

则：

$$\frac{\partial S(a_0,a_1\cdots a_n)}{\partial a_j}=\sum_{i=1}^{n}2(a_0+a_1x_1+\cdots+a_mx_i^m-y_i)x_i^j=0$$

将方程整理得到：

$$\begin{cases}na_0+(\sum_{i=1}^{n}x_i)a_1+\cdots+(\sum_{i=1}^{n}x_i^m)a_m=\sum_{i=1}^{n}y_i\\(\sum_{i=1}^{n}x_i)na_0+(\sum_{i=1}^{n}x_i^2)a_1+\cdots+(\sum_{i=1}^{n}x_i^{m+1})a_m=\sum_{i=1}^{n}x_iy_i\\\cdots\\(\sum_{i=1}^{n}x_i^m)na_0+(\sum_{i=1}^{n}x_i^{m+1})a_1+\cdots+(\sum_{i=1}^{n}x_i^{2m})a_m=\sum_{i=1}^{n}x_i^my_i\end{cases}$$

求解方程可以得到拟合的曲线$p(x)=a_0+a_1x_1+\cdots+a_nx^m$。

Sparse

SparseMatrix

稀疏矩阵（sparse matrix），在数值分析中，是其元素大部分为零的矩阵。反之，如果大部分元素都非零，则这个矩阵是稠密(dense)的。在科学与工程领域中求解线性模型时经常出现大型的稀疏矩阵。

在使用计算机存储和操作稀疏矩阵时，经常需要修改标准算法以利用矩阵的稀疏结构。由于其自身的稀疏特性，通过压缩可以大大节省稀疏矩阵的内存代价。更为重要的是，由于过大的尺寸，标准的算法经常无法操作这些稀疏矩阵。

我使用了邻接表的方式存储稀疏矩阵，例如矩阵：

$$\begin{bmatrix}11&22&0&0&0&0&0\\0&33&44&0&0&0&0\\0&0&55&66&77&0&0\\0&0&0&0&0&88&0\\0&0&0&0&0&0&99\end{bmatrix}$$

可以转换为我定义的稀疏矩阵：

$$\begin{bmatrix} 0 & 0 & 11 \\ 0 & 1 & 22 \\ 1 & 1 & 33 \\ 1 & 2 & 44 \\ 2 & 2 & 55 \\ 2 & 3 & 66 \\ 2 & 4 & 77 \\ 3 & 5 & 88 \\ 4 & 6 & 99 \end{bmatrix}$$

将原来的5*7矩阵压缩为9*3的矩阵，当稀疏矩阵比较大时，压缩效率更加明显。

1. 稀疏矩阵的属性

	形式
邻接表	`Zuth::Matrix<T> m;`
原始矩阵大小	`int oRow, oCol`
邻接表行数	`int size`

2. 稀疏矩阵的构造

	形式
普通构造	`SparseMatrix(Zuth::Matrix<T> input)`
拷贝构造	`SparseMatrix(const SparseMatrix& input)`
无参数构造	`Zuth::Matrix<T> orignMatrix()`

2. 稀疏矩阵的方法

	形式
加法(+)	`SparseMatrix operator+(const SparseMatrix& otherMat)`
减法(-)	`SparseMatrix operator-(const SparseMatrix& otherMat)`
矩阵转置	`SparseMatrix Transpose()`

MachineLearning

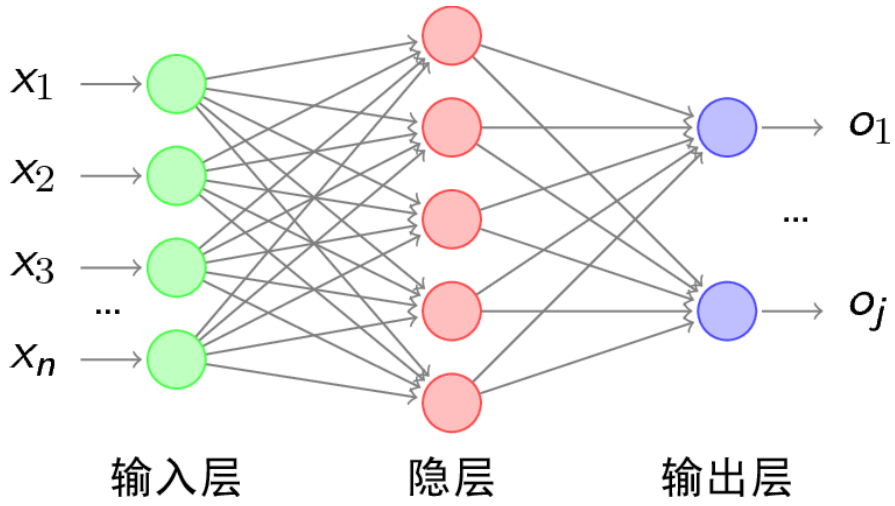
机器学习是人工智能的一个分支。人工智能的研究历史有着一条从以“推理”为重点，到以“知识”为重点，再到以“学习”为重点的自然、清晰的脉络。显然，机器学习是实现人工智能的一个途径，即以机器学习为手段解决人工智能中的问题。机器学习在近30多年已发展为一门多领域交叉学科，涉及概率论、统计学、逼近论、凸分析、计算复杂性理论等多门学科。机器学习理论主要是设计和分析一些让计算机可以自动“学习”的算法。机器学习算法是一类从数据中自动分析获得规律，并利用规律对未知数据进行预测的算法。因为学习算法中涉及了大量的统计

学理论，机器学习与推断统计学联系尤为密切，也被称为统计学习理论。算法设计方面，机器学习理论关注可以实现的，行之有效的学习算法。很多推论问题属于无程序可循难度，所以部分的机器学习研究是开发容易处理的近似算法。

BP神经网络

	形式
前向传播	`void forwardPropagationEpoc();`
反向传播	`void backPropagationEpoc();`
训练	`void training(static vector<sample> sampleGroup, double threshold)`
预测	`void predict(vector<sample>& testGroup)`
输入样本	`void setInput(static vector<double> sampleIn)`
输出样本	`void setOutput(static vector<double> sampleOut)`

BP（Back-propagation，反向传播）神经网络是最传统的神经网络。也就是使用了Back-propagation算法的神经网络。



BP神经网络伪代码如下：

```
初始化网络权值（通常是小的随机值）
do
    forEach 训练样本 ex
        prediction = neural-net-output(network, ex)  // 正向传递
        actual = teacher-output(ex)
        计算输出单元的误差 (prediction - actual)
        计算del(wh)对于所有隐藏层到输出层的权值                          // 反向传递
        计算del(wi)对于所有输入层到隐藏层的权值                          // 继续反向传递
        更新网络权值 // 输入层不会被误差估计改变
until 所有样本正确分类或满足其他停止标准
return 该网络
```

IntelligentAlgorithm

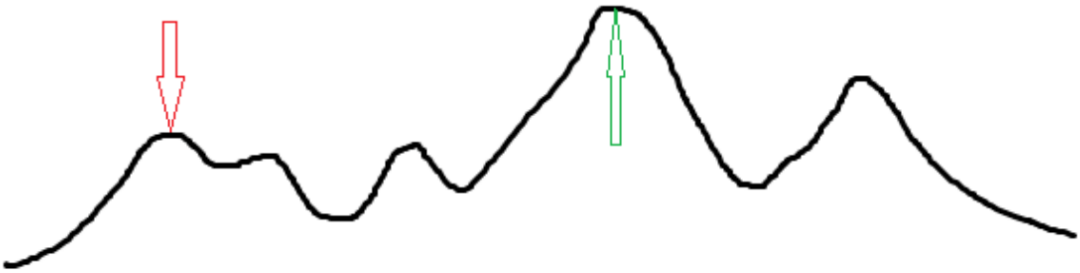
	形式
爬山算法	`class MounClimbling`
模拟退火	`class SA`

爬山算法

爬山算法是一种局部择优的方法，采用启发式方法，是对深度优先搜索的一种改进，它利用反馈信息帮助生成解的决策。

直白地讲，就是当目前无法直接到达最优解，但是可以判断两个解哪个更优的时候，根据一些反馈信息生成一个新的可能解。

因此，爬山算法每次在当前找到的最优方案x附近寻找一个新方案。如果这个新的解 x'更优，那么转移到x'，否则不变。



爬山算法一般会引入温度参数（类似模拟退火）。类比地说，爬山算法就像是一只兔子喝醉了在山上跳，它每次都会朝着它所认为的更高的地方（这往往只是个不准确的趋势）跳，显然它有可能一次跳到山顶，也可能跳过头翻到对面去。不过没关系，兔子翻过去之后还会跳回来。显然这个过程很没有用，兔子永远都找不到出路，所以在这个过程中兔子冷静下来并在每次跳的时候更加谨慎，少跳一点，以到达合适的最优点。

模拟退火

模拟退火是一种随机化算法。当一个问题的方案数量极大（甚至是无穷的）而且不是一个单峰函数时，我们常使用模拟退火求解。

定义当前温度为T，新状态与已知状态之间的能量差值为$\triangle E$,发证状态转移的概率为：

$$P(\triangle E)\begin{cases} 1 & \text{新状态更优} \\ e^{-\frac{\triangle E}{T}} & \text{新状态更劣} \end{cases}$$

项目技术

1. 标识的使用

两个C++文件，如果这两个C文件都include了同一个头文件。而编译时，这两个C文件要一同编译成一个可运行文件，所以就会存在大量的声明冲突。把头文件的内容都放在#ifndef和#endif中。不管你的头文件会不会被多个文件引用。在本项目所有的头文件中都使用了这样的表示。

2. 类的使用

类在物件导向程式设计中是一种面向对象计算机编程语言的构造，是创建对象的蓝图，描述了所创建的对象共同的特性和方法。本项目创建了大量的类。

3. 运算符重载

C++ 允许在同一作用域中的某个函数和运算符指定多个定义，分别称为函数重载和运算符重载。本项目的矩阵类、稀疏矩阵类都有用到运算符的重载。

4. 友元

类的友元函数是定义在类外部，但有权访问类的所有私有（private）成员和保护（protected）成员。尽管友元函数的原型有在类的定义中出现过，但是友元函数并不是成员函数。在矩阵类以及稀疏矩阵类的运算符重载时用到了友元。

5. 引用

引用变量是一个别名，也就是说，它是某个已存在变量的另一个名字。一旦把引用初始化为某个变量，就可以使用该引用名称或变量名称来指向变量。本项目的矩阵类的构造等都用到了引用。

6. 继承

面向对象程序设计中最重要的一个概念是继承。继承允许我们依据另一个类来定义一个类，这使得创建和维护一个应用程序变得更容易。这样做，也达到了重用代码功能和提高执行效率的效果。我在定义类时，如智能算法类用到了继承。

7. 虚函数

虚函数是应在派生类中重新定义的成员函数。 当使用指针或对基类的引用来引用派生的类对象时，可以为该对象调用虚函数并执行该函数的派生类版本。我在定义类时，如智能算法类用到了虚函数。

项目性能分析

素数筛

	复杂度
埃拉托色尼筛	$O(NlogN)$
欧拉筛	$O(N)$

排序算法

	稳定性	平均复杂度	最好情况时间复杂度	最坏情况时间复杂度
快速排序	不稳定	$O(NlogN)$	$O(NlogN)$	$O(N^2)$
归并排序	稳定	$O(NlogN)$	$O(NlogN)$	$O(NlogN)$
堆排序	不稳定	$O(NlogN)$	$O(NlogN)$	$O(NlogN)$
希尔排序	不稳定	$O(N^{1.3})$	$O(N)$	$O(N^2)$

稠密矩阵求解线性方程

		优点	缺点
LU分解	高斯消元	承袭性	数值不稳定
QR分解	**Gram–Schmidt 过程**	易于实现	数值不稳定
QR分解	**Householder 反射**	数值稳定	不可并行化
QR分解	**Givens 旋转**	最复杂的实现	可并行性

测试用例

CoreClass测试

```
void test_class() {
  // 大数
  Zuth::BigNum b1(123);
  Zuth::BigNum b2= b1 * 99;
  cout << b2<<endl;
  // 矩阵的逆
  double data[4] = { 1,2,3,4 };
  Zuth::Matrix<double> m1(2, 2,data );
  Zuth::Matrix<double> m2 = m1.adgMatrix();
  cout << m1 << m2;
}
```

结果：

```
Matrix 2 X 2
1       2
3       4
Matrix 2 X 2
2       4
6       8
Matrix 2 X 2
1       2
3       4
Matrix 2 X 2
14      20
30      44
12177
Matrix 2 X 2
```

```
1       2
3       4
Matrix 2 X 2
-2      1
1.5     -0.5
```

CoreAlgorithm测试

```
void test_algorithm() {
  // 一维前缀和
  Zuth::Vector<int> V;
  for (int i = 1; i < 10; i++)V.pushback(i);
  cout << "orign vector" << endl;
  for (int i = 0; i < V.getsize(); i++)cout << V[i] << " ";
  cout << endl<<"prefix vector" << endl;
  Zuth::Vector<int> V1 = Zuth::Prefix(V);
  for (int i = 0; i < V1.getsize(); i++)cout << V1[i] << " ";
  cout << endl;
  // 一阶差分
  Zuth::Vector<int> V2 = Zuth::Difference(V);
  cout << "Difference vector" << endl;
  for (int i = 0; i < V2.getsize(); i++)cout << V2[i] << " ";
  cout << endl;
  // 二维前缀和
  int data[4] = { 1,2,3,4 };
  Zuth::Matrix<int> M(2, 2,data);
  Zuth::Matrix<int> M2 ;
  M2 = Zuth::Prefix(M);
  cout << "orign matrix\n"<<M << "prefix matrix\n" << M2<<endl;
  // 二维差分
  Zuth::Matrix<int> M3;
  M2 = Zuth::Difference(M);
  cout << "difference matrix\n" << M2 << endl;
  // 并查集
  Zuth::DisjointSet Ds(10);
  cout << "size is " << Ds.setNumber() << endl;
  Ds.dUnion(0, 1);
  Ds.dUnion(0, 5);
  cout << "size is "<<Ds.setNumber() << endl;
  // 快速排序
  int dataq[6] = { 5,7,3,9,3,7 };
  cout << "quick sort" << endl;
  for (int i = 0; i < 6; i++)cout << dataq[i] << " ";
  cout << endl;
  Zuth::quickSort(dataq,  6);
  for (int i = 0; i < 6; i++)cout << dataq[i] << " ";
  cout << endl;
  // 归并排序
  int datam[6] = { 5,7,3,9,3,7 };
  cout << "merge sort" << endl;
  for (int i = 0; i < 6; i++)cout << datam[i] << " ";
  cout << endl;
  Zuth::mergeSort(datam, 0,6);
  for (int i = 0; i < 6; i++)cout << datam[i] << " ";
  cout << endl;
  // 堆排序
  int datah[6] = { 5,7,3,9,3,7 };
  cout << "heap sort" << endl;
  for (int i = 0; i < 6; i++)cout << datah[i] << " ";
```

```
  cout << endl;
  Zuth::heapSort(datah, 6);
  for (int i = 0; i < 6; i++)cout << datah[i] << " ";
  cout << endl;
  // 希尔排序
  int datas[6] = { 5,7,3,9,3,7 };
  cout << "shell sort" << endl;
  for (int i = 0; i < 6; i++)cout << datas[i] << " ";
  cout << endl;
  Zuth::shellSort(datas, 6);
  for (int i = 0; i < 6; i++)cout << datas[i] << " ";
  cout << endl;
}
```

结果：

```
orign vector
1 2 3 4 5 6 7 8 9
prefix vector
1 3 6 10 15 21 28 36 45
Difference vector
1 1 1 1 1 1 1 1
orign matrix
Matrix 2 X 2
1       2
3       4
prefix matrix
Matrix 2 X 2
1       3
4       10

difference matrix
Matrix 2 X 2
1       1
2       2

size is 10
size is 8
quick sort
5 7 3 9 3 7
3 3 5 7 7 9
merge sort
5 7 3 9 3 7
3 3 5 7 7 9
heap sort
5 7 3 9 3 7
3 3 5 7 7 9
shell sort
5 7 3 9 3 7
3 3 5 7 7 9
```

CoreAlgebra测试

```
void test_algebra() {
  // 素数筛
  Zuth::Vector<bool> Vb = Zuth::eratosthenesSieve(50);
  cout << "50以内的素数" << endl;
```

```
  for (int i = 2; i < Vb.getsize(); i++)if (!Vb[i])cout << i << " ";
  cout << endl;
  // 素数筛
  cout << "50以内的素数" << endl;
  Zuth::Vector<int> Vc = Zuth::eulerSieve(50);
  for (int i = 0; i < Vc.getsize(); i++)cout << Vc[i]<<" ";
  // 快速幂
  cout << "\n" << "pow(2,2)=" << Zuth::quickPower(2, 2) ;
  cout << "\n" << "pow(3,3)=" << Zuth::quickPower(3, 3) ;
  cout << "\n" << "pow(5,5)=" << Zuth::quickPower(5, 5) << endl;
}
```

结果：

```
50以内的素数
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
50以内的素数
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
pow(2,2)=4
pow(3,3)=27
pow(5,5)=3125
```

AdvancedAlgebra测试

```
void test_advanced_algebre() {
  // 辛普森
  cout << Zuth::simpsonIntegration(func1, 0, 1)<<endl;
  // 自适应辛普森
  cout << Zuth::zSimpsonIntegration(func1, 0, 1, 0.0001)<<endl;
  // 高斯消元
  double data[12] = { 2,1,-1,8,-3,-1,2,-11,-2,1,2,-3 };
  Zuth::Matrix<double> m(3, 4, data);
  Zuth::Matrix<double> m2 = Zuth::gaussianElimination(m);
  cout << m << m2 << endl;
  // LU分解求解线性方程组
  Zuth::Vector<double> ans1 = Zuth::LUsolve(m);
  cout << m << endl;
  for (int i = 0; i < ans1.getsize(); i++)cout << ans1[i] << ' ';
  cout <<endl;
  // QR分解
  double datan[9] = { 2,-2,3,1,1,1,1,3,-1 };
  Zuth::Matrix<double> M2(3,3,datan);
  cout << M2 << endl;
  Zuth::Vector<Zuth::Matrix<double> >ans2=Zuth::QR<double>(M2);
  cout << ans2[0] << ans2[1] << endl;
  // QR分解求解方程
  Zuth::Vector<double>  ans3=Zuth::QRsolve(m);
  for (int i = 0; i < ans3.getsize(); i++)cout << ans3[i] << ' ';
  cout << endl;
  // 曲线拟合
  double arry1[5] = { 0,0.25,0,5,0.75 };
  double arry2[5] = { 1,1.283,1.649,2.212,2.178 };
  double coefficient[5];
  memset(coefficient, 0, sizeof(double) * 5);
  std::vector<double> vx, vy;
  for (int i = 0; i < 5; i++)
  {
```

```
    vx.push_back(arry1[i]);
    vy.push_back(arry2[i]);
  }
  Zuth::CurveFit<double> cf(vx, vy, 5, 3, coefficient);
  printf("拟合方程为：y = %lf + %lfx + %lfx^2 \n", coefficient[1], coefficient[2], coefficient[3]);
  // 稀疏矩阵构造
  Zuth::Matrix<double> ms1 = Zuth::Eye<double>(4);
  Zuth::SparseMatrix<double> ms2(ms1);
  Zuth::Matrix<double> ms3 = ms2.orignMatrix();
  cout << "原矩阵\n"<<ms1<<"压缩矩阵\n"<<ms2<<"还原矩阵\n"<<ms3<<endl;
  // 稀疏矩阵加法
  Zuth::SparseMatrix<double> ms4 = ms1 + ms1;
  ms3 = ms4.orignMatrix();
  cout << ms3<<endl;
  ms4 = ms1 - ms1;
  ms3 = ms4.orignMatrix();
  cout << ms3 << endl;
  // 稀疏转置
  double data7[4] = { 1,2,3,4 };
  Zuth::Matrix<double> m7(2, 2, data7);
  Zuth::SparseMatrix<double> sm7(m7);
  sm7 = sm7.Transpose();
  ms3 = sm7.orignMatrix();
  cout << m7<<endl<<ms3;
  // 卡特兰数
  cout << Zuth::Catalan(10);
  cout << endl;
  // 贝尔数
  cout << Zuth::Bell(4);
  // 开方
  cout << endl << "sqrt(67)=" << Zuth::sqrtNewton(67) << endl;
}
```

结果：

```
0.5
0.5
Matrix 3 X 4
2       1       -1      8
-3      -1      2       -11
-2      1       2       -3
Matrix 3 X 4
2       0       0       4
0       0.5     0       1.5
0       0       -1      1

Matrix 3 X 4
2       1       -1      8
-3      -1      2       -11
-2      1       2       -3

2 3 -1
Matrix 3 X 3
2       -2      3
1       1       1
1       3       -1

Matrix 3 X 3
-0.816497       0.534522        0.218218
-0.408248       -0.267261       -0.872872
```

```
-0.408248       -0.801784       0.436436
Matrix 3 X 3
-2.44949        4.44089e-16     -2.44949
0       -3.74166        2.13809
0       0       -0.654654

2 3 -1
拟合方程为：y = 1.240746 + 1.281304x + -0.217348x^2
原矩阵
Matrix 4 X 4
1       0       0       0
0       1       0       0
0       0       1       0
0       0       0       1
压缩矩阵
0 0 1
1 1 1
2 2 1
3 3 1
还原矩阵
Matrix 4 X 4
1       0       0       0
0       1       0       0
0       0       1       0
0       0       0       1

Matrix 4 X 4
2       0       0       0
0       2       0       0
0       0       2       0
0       0       0       2

Matrix 4 X 4
0       0       0       0
0       0       0       0
0       0       0       0
0       0       0       0

Matrix 2 X 2
1       2
3       4

Matrix 2 X 2
1       3
2       4
16796
5
sqrt(67)=8.18535
```

AdvancedAlgorithm测试

```
void test_advanced_algorithm() {
  // 线段树
  Zuth::SegmentTree seg(51);
  std::vector<int> nums{ 1,2,3,4,5,6,2,2,7,8,9,0 };
  for (auto i : nums) {
    seg.insert(i);
  }
  cout << seg.count(2, 3) << endl;
```

```
  seg.erase(3);
  cout << seg.count(2, 3) << endl;
  // 乘法逆元
  int n1=5, n2=7, n3, n4;
  Zuth::exgcd(n1, n2, n3, n4);
  cout << n1 << " " << n2 << " " << n3 << " " << n4 << endl;
  // 树状数组
  Zuth::Vector<int> Zu;
  for (int i = 0; i < 9; i++)Zu.pushback(i+1);
  Zuth::BitTree Zb(Zu);
  cout << Zb.qSum(1, 2);
}
```

结果：

```
4
3
5 7 3 -2
3
```

MachineLearning测试

```
void test_machine_learning() {
  Zuth::BpNet testNet;
  // 学习样本
  vector<double> samplein[4];
  vector<double> sampleout[4];
  samplein[0].push_back(0); samplein[0].push_back(0); sampleout[0].push_back(0);
  samplein[1].push_back(0); samplein[1].push_back(1); sampleout[1].push_back(1);
  samplein[2].push_back(1); samplein[2].push_back(0); sampleout[2].push_back(1);
  samplein[3].push_back(1); samplein[3].push_back(1); sampleout[3].push_back(0);
  Zuth::sample sampleInOut[4];
  for (int i = 0; i < 4; i++)
  {
    sampleInOut[i].in = samplein[i];
    sampleInOut[i].out = sampleout[i];
  }
  vector<Zuth::sample> sampleGroup(sampleInOut, sampleInOut + 4);
  testNet.training(sampleGroup, 0.0001);

  // 测试数据
  vector<double> testin[4];
  vector<double> testout[4];
  testin[0].push_back(0.1);   testin[0].push_back(0.2);
  testin[1].push_back(0.15);  testin[1].push_back(0.9);
  testin[2].push_back(1.1);   testin[2].push_back(0.01);
  testin[3].push_back(0.88);  testin[3].push_back(1.03);
  Zuth::sample testInOut[4];
  for (int i = 0; i < 4; i++) testInOut[i].in = testin[i];
  vector<Zuth::sample> testGroup(testInOut, testInOut + 4);

  // 预测测试数据，并输出结果
  testNet.predict(testGroup);
  for (int i = 0; i < testGroup.size(); i++)
  {
    for (int j = 0; j < testGroup[i].in.size(); j++) cout << testGroup[i].in[j] << "\t";
    cout << "-- prediction :";
```

```
    for (int j = 0; j < testGroup[i].out.size(); j++) cout << testGroup[i].out[j] << "\t";
    cout << endl;
  }
}
```

部分结果：

```
...
training error: 0.000100008
training error: 0.000100007
training error: 0.000100006
training error: 0.000100004
training error: 0.000100003
training error: 0.000100002
training error: 0.0001
0.1     0.2     -- prediction :0.00680295
0.15    0.9     -- prediction :0.963454
1.1     0.01    -- prediction :0.99541
0.88    1.03    -- prediction :0.0089501
```

IntelligentAlgorithm测试

```
void test_Inte_algorithm() {
  // 爬山
  Zuth::MounClimbling MC(func2, -10, 10);
  cout << MC.getAns() << endl;
  // 模拟退火
  Zuth::SA S(func2, -10, 10);
  cout << S.getAns() << endl;
}
```

结果：

```
1.5529e-26
-0.118673
```