Part 1 什么是数据结构

一些定义...

> “
>
> - 数据结构是数据对象，以及存在于该对象的实例和组成实例的数据元素之间的各种联系，这些联系可以通过定义相关函数来给出
> - 数据结构是ADT（抽象数据类型）的物理实现
> - 数据结构是计算机中存储、组织数据的方式，通常情况下，精心选择的数据结构可以带来最优效率算法
> - ...

例1：如何在书架上摆放图书?

- 随便放

- 按照拼音字母顺序放
- 先分类到指定区域，再在每个类中按照书名的拼音字母顺序排放
- ...

每一个种类使用多少书架？事先分配好吗？类别要怎么分，分多细？

解决问题方法的效率，跟数据组织方式有关

例2：写一个函数`PrintN()`根据参数N打印1~N全部整数

```
#include <iostream>
using namespace std;
void PrintN_C(int N); //循环
void PrintN_R(int N); // 递归
int main()
{
    int N;
    cin >> N;
    PrintN_C(N);
    return 0;
}

void PrintN_C(int N)
{
    for (int i = 1; i < N + 1; i++)
        cout << i << endl;
    return;
}
void PrintN_R(int N)
{
    if (N != 1)
        PrintN_R(N - 1);
    cout << N << endl;
}
```

经过测试，当输入N=10 0000时，循环函数正常输出，递归函数错误退出，主要原因在层层递归占用了大量的空间，空间不足，递归便无法正常运行

解决问题方法的效率，跟空间的利用效率有关

例3：写程序计算给定多项式$f(x)=\sum_{i=0}^{9} i\cdot x^i$在给定点x处的值

暴力算法：$f(x)=a_0+a_1x+\cdots+a_{n-1}x^{n-1}+a_nx^n$

秦九韶算法：$f(x)=a_0+x\left(a_1+x\left(\cdots\left(a_{n-1}+x\left(a_n\right)\right)\cdots\right)\right)$

分别使用两种算法计算该多项式1e9次，得到的结果如下图，算法不同得到相同的结果需要的时间相差了一个数量级！

测试代码如下，修改 `MAXK` 以节省时间：）

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

//暴力的算法
double f1(int n, double a[], double x)
{
    double p = a[0];
    for (int i = 1; i <= n; i++)
        p += a[i] * pow(x, i);
    return p;
}

//秦九韶算法
double f2(int n, double a[], double x)
{
    double p = a[n];
    for (int i = n; i > 0; i--)
        p = a[i - 1] + x * p;
    return p;
}

#define MAXN 10
```

```
#define MAXK 1e9

clock_t start, stop;
double duration1, duration2; //记录被测函数运行时间，以秒为单位
int main()
{
    double a[MAXN];
    for (int i = 0; i < MAXN; i++)
    {
        a[i] = (double)i;
    }
    //不在测试范围的准备工作
    start = clock();
    //这里写被测函数
    //如果被测函数太快，根本不到一个tick，建议重复运行!
    for (int i = 0; i < MAXK; i++)
        f1(MAXN - 1, a, 1.1);
    stop = clock();
    duration1 = (double)(stop - start) / CLK_TCK;
    //不在测试范围的后继工作
    cout << "duration1 is " << duration1 << "s" << endl;

    //不在测试范围的准备工作
    start = clock();
    //这里写被测函数
    //如果被测函数太快，根本不到一个tick，建议重复运行!
    for (int i = 0; i < MAXK; i++)
        f2(MAXN - 1, a, 1.1);
    stop = clock();
    duration2 = (double)(stop - start) / CLK_TCK;
    //不在测试范围的后继工作
    cout << "duration2 is " << duration2 << "s" << endl;
    return 0;
}
```

解决问题方法的效率，跟算法的巧妙程度有关

小结

数据结构是**数据对象**在计算机中的组织方式

- 逻辑结构
- 物理存储结构

数据对象必定与一系列加在其上的**操作**相关联

完成这些操作所使用的方法就是**算法**

抽象数据类型 ADT

- 数据类型
  - 数据对象集
  - 数据集合相关联的操作集
- 抽象：描述数据类型的方法不依赖于具体实现
  - 与存放数据的机器无关
  - 与数据存储的物理结构无关
  - 与实现操作的算法和编程语言均无关

只描述**是什么**，不涉及**如何做**

例4：矩阵的抽象数据类型定义

- 类型名称：矩阵（Matrix）
- 数据对象集：一个M×N的矩阵$A_{M\times N}=(a_{ij})$（i=1, …, M; j=1, …, N）（不关心这个矩阵怎样实现，二维数组？一维数组？十字链表？）由M×N个三 元组$<a,i,j>$构成，其中a是矩阵元素的值（不关心元素的值是什么类型，int? float? double?），i是元素所在的行号，j是元素 所在的列号。
- 操作集：对于任意矩阵A、B、C ∈ Matrix，以及整数i、j、M、N
  - Matrix Create(int M, int N)：返回一个M×N的空矩阵；
  - int GetMaxRow(Matrix A)：返回矩阵A的总行数；
  - int GetMaxCol(Matrix A)：返回矩阵A的总列数；

- ElementType GetEntry(Matrix A, int i, int j)：返回矩 阵A的第i行、第j列的元素；
- Matrix Add(Matrix A, Matrix B)：如果A和B的行、列数一 致，则返回矩阵C=A+B（不关心怎样实现加法，按行相加？按列相加？什么语言实现？），否则返回错误标志；
- Matrix Multiply(Matrix A, Matrix B)：如果A的列数等于B 的行数，则返回矩阵C=AB，否则返回错误标志；

Part 2 什么是算法Algorithm

定义

1. 一个有限指令集
2. 接受一些输入（有时也不需要输入）
3. 产生输出
4. 一定在有限步骤之后终止
5. 每一条指令必须
   - 有充分明确的目标，不能有歧义
   - 计算机处理的范围之内
   - 描述应不依赖于任何一种计算机语言以及具体实现手段

例1：选择排序算法的伪码描述

```
void SelectionSort(int List[], int N)
{
    //将N个整数List[0] ~ List[N - 1]进行非递减排序
    for(i = 0; i < N; ++i)
    {
        //从List[i] ~ List[N - 1]中找出最小元素并将其的位置赋给
MinPosition
        MinPosition = ScanForMin(List, i, N - 1);
        //未排序部分（即List[i] ~ List[N]）的最小元素交换到有序部分的最后
位置
        Swap(List[i], List[MinPosition]);
    }
}
```

看上这似乎和C语言有点类似，但是它依然是抽象的算法，比如

- List怎么实现？数组？链表？
- Swap怎么实现？函数？宏？

什么是好算法?

空间复杂度 $S(n)$ —— 根据算法写成的程序在执行时 **占用存储单元的长度**。这个长度往往与输入数据的规模有关。空间复杂度过高的算法可能导致使用的 内存超限，造成程序非正常中断

譬如前面的例2，我们输入10000时，递归程序直接无法正常打印结果，原因是每次递归都需要一块新的存储空间去存放产生的递归函数，输入10000就需要10000个函数存储空间，显然，它的空间复杂度和 `N` 有关，满足不了空间需求时程序就会出错；而对应循环函数，它只使用了自己一个函数，空间复杂度为常数（与 `N` 无关），正常运行

时间复杂度 $T(n)$ —— 根据算法写成的程序在执行时 **耗费时间的长度**。这个长度往往也与输入数据的规模有关。时间复杂度过高的低效算法可能导致我们在有生之年都等不到运行结果

譬如前面的例3，暴力算法和秦九韶算法得到的运行时间直接相差了一个数量级，我们注意到，两函数 `for` 循环的次数相同，但是暴力算法中的 `pow(x, i)` 相当于执行 `i` 次相乘，就是这里导致了暴力算法的时间复杂度远大于秦九韶算法

```
//暴力的算法
double f1(int n, double a[], double x)
{
    double p = a[0];
    for (int i = 1; i <= n; i++)
        p += a[i] * pow(x, i);
    return p;
}

//秦九韶算法
double f2(int n, double a[], double x)
{
    double p = a[n];
    for (int i = n; i > 0; i--)
        p = a[i - 1] + x * p;
    return p;
}
```

分析算法的复杂度时我们往往关注**最坏情况复杂度**

复杂度的渐进表示法

- $T(n) = O(f(n))$ 表示存在常数 $C > 0, n_0 > 0$ 使得当 $n \geq n_0$ 时有 $T(n) \leq C \cdot f(n)$

  简单来说，$O(f(n)$的意思就是对于充分大的n来说，$f(n)$是$T(n)$的上界

- $T(n) = \Omega(g(n))$ 表示存在常数 $C > 0, n_0 > 0$ 使得当 $n \geq n_0$ 时有 $T(n) \geq C \cdot g(n)$

- $T(n) = \Theta(h(n))$ 表示同时有 $T(n) = O(h(n))$ 和 $T(n) = \Omega(h(n))$

要注意：复杂度的上界和下界均不是唯一的，例如一个算法的复杂度是n，它可以写作$O(n)$，$O(n^2)$等等，但我们日常分析时通常希望上界和下界可以更贴近算法的实际情况

- **感性认识一下不同复杂度**

函数	1	2	4	8	16	32
1	1	1	1	1	1	1
$\log n$	0	1	2	3	4	5
n	1	2	4	8	16	32
$n \log n$	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296
$n!$	1	2	24	40326	2092278988000	26313×10^{33}

复杂度分析窍门

- **两段算法拼接**

$$T_1(n) + T_2(n) = \max\left(O\left(f_1(n)\right), O\left(f_2(n)\right)\right)$$

- **两端算法嵌套**

$$T_1(n) \times T_2(n) = O\left(f_1(n) \times f_2(n)\right)$$

- **$T(n)$是关于n的k阶多项式**

$T(n) = \Theta\left(n^k\right)$，即取最高阶项即可

- `for`

时间复杂度等于循环次数乘以循环体代码长度

- `if else`

时间复杂度取决于 `if` 的**条件判断复杂度**和两（也可以更多）**分枝部分的复杂度**，总体复杂度取这些部分的最大值

Part 3 应用实例-最大子列和问题

题面

给定N个整数的序列$\{A_1, A_2, \ldots, A_n\}$，求函数$f(i,j) = \max\left\{0, \sum_{k=i}^{j} A_k\right\}$的最大值，如果是负数，则返回0

也就是从这个整数序列中找到连续的一串数的和最大，求这个最大值

算法1：暴力

思路

暴力枚举出这个序列的所有子串的和，比较出最大值就是结果

实现

```
int MaxSubseqSum1( int A[], int N )
{   int ThisSum, MaxSum = 0;
    for(int i = 0; i < N; i++ ) { /* i是子列左端位置 */
          for(int j = i; j < N; j++ ) { /* j是子列右端位置 */
                ThisSum = 0;  /* ThisSum是从A[i]到A[j]的子列和 */
                for(int k = i; k <= j; k++ )
                          ThisSum += A[k];
                          if( ThisSum > MaxSum ) /* 如果刚得到的这
个子列和更大 */
                                    MaxSum = ThisSum;     /* 则更
新结果 */
          } /* j循环结束 */
     } /* i循环结束 */
     return MaxSum;
}
```

- **时间复杂度**

三层 `for` 循环，$O(N^3)$

- **问题所在**

前两个循环确定了 `i` 和 `j` ，通过第三个循环，算出了 `A[i]` 到 `A[j]` 的和

当 `j = j + 1` 后，再一次进入了第三个循环计算 `A[i]` 到 `A[j + 1]` 的和，实际上，我们将上一次循环计算的和再加上 `A[j + 1]` 不就可以了吗，不必要从头到尾再加一遍

算法2：$O(N^3) \to O(N^2)$

- **思路**

根据算法1的问题所在，我们用每次 `j` 更新的时候更新一下 `ThisSum` 的做法，取代每次 `j` 更新就调用一趟循环求和，这样也可以正确求解问题

- **实现**

```
int MaxSubseqSum2( int A[], int N )
{   int ThisSum, MaxSum = 0;
    for(int i = 0; i < N; i++ ) { /* i是子列左端位置 */
          ThisSum = 0;  /* ThisSum是从A[i]到A[j]的子列和 */
          for(int j = i; j < N; j++ ) { /* j是子列右端位置 */
                ThisSum += A[j];        /*对于相同的i，不同的j，只要在j-1次循环的基础上累加1项即可*/
                if( ThisSum > MaxSum ) /* 如果刚得到的这个子列和更大 */
                      MaxSum = ThisSum;    /* 则更新结果 */
          } /* j循环结束 */
     } /* i循环结束 */
     return MaxSum;
}
```

时间复杂度

改进后的算法少了一层循环，时间复杂度降低到$O(N^2)$

> 当一个算法的复杂度是$O(n^2)$时，专业的程序员下意识地考虑能不能降到$O(nlogn)$?

算法3：分而治之

思路

我们总是把当前的序列分为左右两个部分，分别去求这两个序列的最大子串和A和B，还要再从当前序列的中间开始向左右两边扫描，找到跨越中界的最大子串和C，这样，A，B，C中的最大值就是当前序列的最大子串和

举个例子来理解这种分治的思想

实现

```
int Max3( int A, int B, int C )
{ /* 返回3个整数中的最大值 */
    return A > B ? A > C ? A : C : B > C ? B : C;
}

int DivideAndConquer( int List[], int left, int right )
{ /* 分治法求List[left]到List[right]的最大子列和 */
    int MaxLeftSum, MaxRightSum; /* 存放左右子问题的解 */
    int MaxLeftBorderSum, MaxRightBorderSum; /*存放跨分界线的结果*/

    int LeftBorderSum, RightBorderSum;
    int center, i;

    if( left == right )  { /* 递归的终止条件，子列只有1个数字 */
        if( List[left] > 0 )  return List[left];
```

```
        else return 0;
    }

    /* 下面是"分"的过程 */
    center = ( left + right ) / 2; /* 找到中分点 */
    /* 递归求得两边子列的最大和 */
    MaxLeftSum = DivideAndConquer( List, left, center );
    MaxRightSum = DivideAndConquer( List, center+1, right );

    /* 下面求跨分界线的最大子列和 */
    MaxLeftBorderSum = 0; LeftBorderSum = 0;
    for( i=center; i>=left; i-- ) { /* 从中线向左扫描 */
        LeftBorderSum += List[i];
        if( LeftBorderSum > MaxLeftBorderSum )
            MaxLeftBorderSum = LeftBorderSum;
    } /* 左边扫描结束 */

    MaxRightBorderSum = 0; RightBorderSum = 0;
    for( i=center+1; i<=right; i++ ) { /* 从中线向右扫描 */
        RightBorderSum += List[i];
        if( RightBorderSum > MaxRightBorderSum )
            MaxRightBorderSum = RightBorderSum;
    } /* 右边扫描结束 */

    /* 下面返回"治"的结果 */
    return Max3( MaxLeftSum, MaxRightSum, MaxLeftBorderSum +
MaxRightBorderSum );
}
```

• 时间复杂度

当问题中有N个数字时，复杂度为$T(N)$，则求左右两半各自的最大子串和A，B的复杂度各为$T(\frac{N}{2})$，求跨越中界的最大子串和C的复杂度为$O(n)$（扫描整个序列），由此，可以得到一个递推公式并推出最终的时间复杂度

算法4：在线处理

思路

在线的意思就是实时处理每个输入的数据，在任何地方终止输入，算法得到的解都是在当前状况下正确的

从头开始，累加 `ThisSum`，每次循环判断 `ThisSum < 0` 时就将其置 `0`（相当于前面这部分的和小于0，没有累加价值，直接舍弃），每次循环判断 `ThisSum > MaxSum` 时，将 `ThisSum` 的值赋给 `MaxSum`

实现

```
int MaxSubseqSum4( int A[], int N )
{   int ThisSum, MaxSum;
    ThisSum = MaxSum = 0;
    for(int i = 0; i < N; i++ ) {
          ThisSum += A[i]; /* 向右累加 */
          if( ThisSum > MaxSum )
                MaxSum = ThisSum; /* 发现更大和则更新当前结果 */
          else if( ThisSum < 0 ) /* 如果当前子列和为负 */
                ThisSum = 0; /* 则不可能使后面的部分和增大，抛弃之 */
    }
    return MaxSum;
}
```

时间复杂度

我们发现这个算法实现中只有一个 `for` 循环，时间复杂度仅为$O(N)$，这是我们可以达到的最快算法了，因为至少应该将每个元素都扫描过一遍

四种算法的运行时间比较

练习

1. 最大子列和问题

这里使用在线处理解决这道题

代码

```
#include <iostream>
using namespace std;
int a[100005];
int main()
{
    int k;
    cin >> k;
    for (int i = 0; i < k; i++)
        cin >> a[i];

    int thisSum = 0, maxSum = 0;
    for (int i = 0; i < k; i++)
    {
        thisSum += a[i];
        if (thisSum < 0)
            thisSum = 0;
        else if (thisSum > maxSum)
            maxSum = thisSum;
    }

    cout << maxSum << endl;
    return 0;
}
```

评测结果

2. Maximum Subsequence Sum

和前面那道题的区别在这道题还要求输出我们找到的最大和的子串的头尾元素，注意全为负数时要将输出整个数组头尾元素，只存在负数和0时输出头尾元素均为0即可

• 代码

```
#include <iostream>
using namespace std;
int a[100005];
int main()
{
    int k;
    cin >> k;
    for (int i = 0; i < k; i++)
        cin >> a[i];

    //在线处理算法
    int thisSum = 0, maxSum = 0;
    // 这几个变量用来存放thisSum和maxSum对应的首尾元素的索引
    int thisHead = 0, thisTail = 0, maxHead = 0, maxTail = 0;
    for (int i = 0; i < k; i++)
    {
        thisSum += a[i];
        thisTail = i; //加上a[i]后thisSum对应的尾元素自然是a[i]
        if (thisSum < 0)
        {
            //thisSum置零将前面的元素全部抛弃，将其首尾元素置为后面第一个元
素
            thisSum = 0;
            thisHead = i + 1;
            thisTail = i + 1;
        }
        else if (thisSum > maxSum)
        {
            //将thisSum对应的子串接纳为最大子串，最大子串对应的首尾元素也要
同时更改
            maxSum = thisSum;
            maxHead = thisHead;
```

```
            maxTail = thisTail;
        }
    }

    if (maxSum == 0 && maxHead == 0 && maxTail == 0 && a[0] <= 0)
//针对全为负数或0的情况
    {
        int flag = -1; //记录第一个0的位置，如果没有0就保持-1
        for (int i = 0; i < k; i++)
            if (a[i] == 0)
            {
                flag = i;
                break;
            }
        if (flag == -1) //全为负数，将maxTail置为整个数组尾部索引
            maxTail = k - 1;
        else //存在0，将第一个0的位置设为maxHead与maxTail
        {
            maxHead = flag;
            maxTail = flag;
        }
    }

    cout << maxSum << ' ' << a[maxHead] << ' ' << a[maxTail] <<
endl;
    return 0;
}
```

- **评测结果**

3. 实现二分查找函数

函数接口定义

```
Position BinarySearch(List L, ElementType X);
```

其中 `List` 结构定义如下

```
typedef int Position;
typedef struct LNode *List;
struct LNode{
    ElementType Data[MAXSIZE];
    Position Last;  //保存线性表中最后一个元素的位置
}
```

`L` 是用户传入的一个线性表，其中 `ElementType` 元素可以通过 `>、==、<` 进行比较，并且题目保证传入的数据是递增有序的。函数 `BinarySearch` 要查找 `X` 在 `Data` 中的位置，即数组下标（注意：元素从下标1开始存储）。找到则返回下标，否则返回一个特殊的失败标记 `NotFound`。

裁判程序样例

```
#include <stdio.h>
#include <stdlib.h>

#define MAXSIZE 10
#define NotFound 0
typedef int ElementType;

typedef int Position;
typedef struct LNode *List;
struct LNode {
    ElementType Data[MAXSIZE];
    Position Last; /* 保存线性表中最后一个元素的位置 */
};

List ReadInput(); /* 裁判实现，细节不表。元素从下标1开始存储 */
Position BinarySearch( List L, ElementType X );

int main()
{
    List L;
    ElementType X;
    Position P;

    L = ReadInput();
```

```
    scanf("%d", &X);
    P = BinarySearch( L, X );
    printf("%d\n", P);

    return 0;
}

/* 你的代码将被嵌在这里 */
```

- **代码**

```
Position BinarySearch(List L, ElementType X)
{
    Position first = 0, last = L->Last, mid;

    while (first <= last)    //少了'=', 出错一次
    {
        mid = first + (last - first) / 2;
        if (L->Data[mid] < X)
            first = mid + 1;     //直接使用mid, 而没有+1,-1, 出错一次
        else if (X < L->Data[mid])
            last = mid - 1;
        else
            return mid;
    }
    return NotFound;
}
```

- **评测结果**