上机题第一题实验报告

王伟任　　计 33　　2013011333

一、题目要求及分析

第一章上机题 3：编程观察无穷级数 $\sum_1^\infty \frac{1}{n}$ 的求和计算。分别采用 IEEE 单精度浮点数和 IEEE 双精度浮点数计算，观察当 n 为何值时求和结果不再变化。

由题可知，当求和结果不再变化时，说明此时新增项 $\frac{1}{n}$ 的精度已经超出浮点数求和的计算精度。浮点数求和时，需要将两个相加的浮点数按位对齐，即小数点对齐。在这种情况下，当一个加数很大另一个加数很小时，由于浮点数所能表示的精度有限，因此求和运算会优先保留较大数的有效位数，如果较小数的有效位数对齐到浮点数精度范围之外就会被视为 0。

IEEE 单精度浮点数能够表示 24 位二进制有效位，即 24 位尾数；IEEE 双精度浮点数能够表示 53 位尾数。

二、实验结果及分析

（1）　采用 IEEE 单精度浮点数计算：

由于单精度浮点数所能表示的精度并不算大，因此直接采用依次相加判断结果是否与上一次的结果相同的办法。最后输出的结果如下：

Answer is 2097152

The summary is 15.403683

Time cost：15

当 n=2097152 时，结果中再加 $\frac{1}{n}$ 就没有效果了。此时总和约为 15.4，输出结果只保留六位小数。得出这些结果花费 15 个计时单位，即 15ms。

2097152 是 2 的 21 次方，因此 1/2097152 用二进制表示为 0.000000000000000000001，小数点后共 20 个 0 ，1 在小数点后第 21 位。而当前的级数和整数部分为 15，用二进制表示即小数点前有四位 1111。单精度浮点数能够表示 24 位有效位数，因此求和的时候小数点后至多能够记录 20 位；然而加数 1/2097152 的首位非零位在小数点后第 21 位，与已有求和结果对齐时，第 21 位由于超过记录范围被截断，因此相当于加上 0，结果没有变化。

理论上，从 2 的 20 次方开始，加数用二进制表示结果都是小数点后跟着 20 个 0，小数点后第 21 位开始出现非零位，这些数应该都被舍去。但是，C++编译中会将这些非零位数大于等于两位的数取近似为 0.00000000000000000001，小数点后有 19 个 0，第 20 位变成了 1。所以说，2097151 以及之前的一百多万个数在加到已有求和结果上时加的数实际都是 9.5367431640625e-07，也就是 2^(-20)。

（2）　采用 IEEE 双精度浮点数计算：

理论上，双精度浮点数的计算过程应该和单精度浮点数相同，但是由于结果太大，难以用现有计算机在短时间内实现，因此只能采取近似方法。

数学上，调和级数求和有近似公式 $\sum_{k=1}^{n}\frac{1}{k} = \ln(n+1) + \varepsilon$ ，其中 ε 为欧拉常数。因

此，在算法中采用近似 $\sum_{k=i}^{j}\frac{1}{k}=\ln(j+1)-\ln(i)$ 使计算时间大幅度减少。但是这样的近似前提是 i 和 j 都比较大，因此前 20000000 个数采用依次累加的原则，从 20000000 开始利用近似。每次近似的区间为[k,5k)，k 为计数值。在当前和中加上从 k 到 5k 的近似计算结果，然后比较此值与此值加上 1/(5k)是否相等。如果相等，则说明最终所求的结果在 k 到 5k 之间。得到这样的结论之后对该区间分为四份，对每一个区间采取类似的判断原则，得到新的结果区间，继续细分……直到确定结果。最后输出的结果如下：

Answer is 281474976710656

The summary is 33.84828030677634558287536492571234703063964843750000

Cor summary is 33.84828030677621768518292810767889022827148437500000

第一行的是最终的计数值 k，后面两行为累加的结果，用两种计算方法计算防止我的算法出问题。结果显示算法并没有什么问题。

双精度能表示 53 位尾数。和值的整数部分为 33（100001），需要消耗 6 位，因此小数点后只能记录 47 位。而 281474976710656 则恰好是 2 的 48 次方，因此无法记录。

如果采用与单精度相同的算法，根据比例算得需要的计时单位数量为 281474976710656 / 2097152 * 15 = 2013265920，约为 23 天。虽然不同的电脑计算速度略有差别，但是终归不是可接受的时间。

（3） 单精度与双精度的计算结果比较

只需要简单的用两种数据类型依次相加而已，结果如下：

Using single: 15.40368270874023437500000000000000000000000000000000

Using double: 15.13330669507819337127330072689801454544067382812500

可以看出，用单精度计算出的结果比双精度计算出的结果要大很多，而且正确的位数实际只有整数部分和第一位小数部分（以二进制表示的话，前五位都是 1111.0）。原因如同之前所说，用单精度浮点数求和过程中采取了一些近似，放大了一百多万个数，因此求和结果并不准确。

三、实验代码

采用 C++语言实现。

单精度算法实现：

```
#include <cstdio>
#include <time.h>

int main(){
	float s2, s1 = 0.0;
	float adder;
	long i = 0;
	time_t start, finish;
	start = clock();
	do{
		i++;
		adder = 1.0 / i;
		s2 = s1;
		s1 = s2 + adder;
```

```
    } while(s1 != s2);
    finish = clock();
    printf("Answer is %ld\n", i);
    printf("The summary is %f\n", s1);
    printf("Time cost: %ld\n", finish - start);
    return 0;
}
```

双精度算法实现：

```
#include <cstdio>
#include <cmath>

int main(){
    double s1 = 0.0;
    long long i = 1;
    double adder;
    for (;i < 20000000; i++){
        adder = 1.0 / i;
        s1 += adder;
    }
    double klans = s1;
    double ansSum = s1;
    adder = log(5*i)-log(i);
    double s2 = s1 + adder;
    adder = 1.0 / (5*i);
    s1 = s2 + adder;
    while(s1 != s2){
        ansSum = s1;
        i = 5 * i + 1;
        adder = log(5*i)-log(i);
        s2 = s1 + adder;
        adder = 1.0 / (5*i);
        s1 = s2 + adder;
    }
    long long beginn = i;
    long long eend = 5 * i;
    while (beginn < eend){
        long long delta = (eend - beginn) / 4;
        long long b[5];
        b[0] = beginn - 1;
        b[1] = beginn + delta;
        b[2] = b[1] + delta;
        b[3] = b[2] + delta;
        b[4] = eend;
```

```
        for (int kk = 0; kk < 4; kk ++){
            adder = log(b[kk + 1]) - log(b[kk] + 1);
            double as1 = ansSum + adder;
            double as2 = as1 + 1.0 / b[kk + 1];
            if (as1 == as2){
                beginn = b[kk] + 1;
                eend = b[kk + 1];
                break;
            }
            ansSum = as2;
        }
    }
    printf("Answer is %lld\n", beginn);
    printf("The summary is %.50f\n", ansSum);
    printf("Cor summary is %.50f\n", klans + log(beginn+1) - log(20000000));
    return 0;
}
```

计算结果比较：

```
#include <cstdio>

int main(){
    float s1 = 0, adder1;
    double s2 = 0, adder2;

    for (int i = 1; i <= 2097152; i++){
        adder1 = 1.0 / i;
        adder2 = 1.0 / i;
        s1 += adder1;
        s2 += adder2;
    }

    printf("Using single: %.50f\n", s1);
    printf("Using double: %.50f\n", s2);
    return 0;
}
```