《现代密码学》实验报告

实验名称：DLP计算	实验时间：2024年12月28日
学生姓名：黄集瑞	学号：22336090
学生班级：22保密管理	离散对数计算结果：182786828444927184

一、实验目的

> 本次实验通过让同学们实现 Pollard ρ 算法求解离散对数问题，能够帮助同学们深入理解离散对数问题的数学背景及其在有限域 $\mathbb{Z}_p$ 中的计算过程；使得同学们掌握 Pollard ρ 算法的实现细节及其优化方法，有助于帮助大家理解随机化算法在解决数论问题中的应用，提高对算法复杂性分析与大数运算的实际编程能力，为后续的实验打下基础。

二、实验内容

> - 用C++实现DLP的Pollard ρ算法求解过程
> - 输入如下：
> - p：为素数输入
> - n：生成元阶数
> - α：生成元
> - β：目标值
>
> 以上输入均由字符串输入
>
> - 输出如下：
> - x：离散对数的唯一解

三、实验原理

> - 本次实验的原理较为简单，需要我们利用ρ算法来进行离散对数问题的求解，而算法的具体内容可以在课本或者网上找到相应的参考资料，具体过程如下所示：

算法 6.2 Pollard ρ 离散对数算法 (G, n, α, β)

procedure $f(x, a, b)$

if $x \in S_1$

  then $f \leftarrow (\beta \cdot x, a, (b+1) \bmod n)$

  else if $x \in S_2$

  then $f \leftarrow (x^2, 2a \bmod n, 2b \bmod n)$

  else $f \leftarrow (\alpha \cdot x, (a+1) \bmod n, b)$

return (f)

main

定义划分 $G = S_1 \cup S_2 \cup S_3$

$(x, a, b) \leftarrow f(1, 0, 0)$

$(x', a', b') \leftarrow f(x, a, b)$

while $x \neq x'$

$$\textbf{do}\begin{cases}(x,a,b) \leftarrow f(x,a,b) \\ (x', a', b') \leftarrow f(x', a', b') \\ (x', a', b') \leftarrow f(x', a', b')\end{cases}$$

if $\gcd(b'-b, n) \neq 1$

  then return("failure")

  else return $((a-a')(b'-b)^{-1} \bmod n)$

此处都是Z_p下的计算

如何求$a \in Z_n$的阶n？注意到：

- $\mathrm{ord}(Z_n) = \varphi(n)$ 特别地，$\mathrm{ord}(Z_p) = p - 1$
- $\mathrm{ord}(a) \mid \mathrm{ord}(Z_n)$ 拉格朗日定理的推论

因此可以从$\varphi(Z_n)$开始，尝试除以$\varphi(n)$的每个素因子：

$order = \varphi(Z_n)$
for p in factorize($\varphi(Z_n)$):
  if $a^{order/p} = 1$:
    $order = order/p$
return $order$

假定我们如下定义 S_1, S_2, S_3：

$$S_1 = \{x \in \mathbb{Z}_p \quad : x \equiv 1 (\bmod 3)\}$$
$$S_2 = \{x \in \mathbb{Z}_p \quad : x \equiv 0 (\bmod 3)\}$$
$$S_3 = \{x \in \mathbb{Z}_p \quad : x \equiv 2 (\bmod 3)\}$$

注意，必须保证 $1 \notin S_2$ [因为 $1 \in S_2$ 时，对任意的 $i \geqslant 0$ 都有 $x_i = (1,0,0)$]。

$c(b'-b) \equiv a - a' \ (\bmod n)$有$\gcd(\cdots)$个解，对于$\gcd(\cdots) \neq 1$的情况，可以求出全部解再检验，求这些解的方法参见：https://oi-wiki.org/math/number-theory/linear-equation/

但是需要注意的是，课本上并没有考虑$gcd(b'-b)$与n之间没有逆的情况，这种情况可以通过求解线性同余方程来解决。由于本人一开始也并没有完全理解求解过程，所以这里附上详细的思路：

假设此时我们需要求解线性同余方程$ax \equiv b(mod\ n)$，由于a与n之间不互素，所以无法求出逆元然后将a除过去。所以此时我们计算$t = gcd(a, n)$的值，并且让等式两边包括n同时除t，得到$a' = a/t$，$b' = b/t$，$n' = n/t$，方程便可以化为$a'x \equiv b'(mod\ n')$，此时a'已经与n'互素，那么便可以求出逆元然后得到一个解为x，但是由于该线性同余方程的解的个数等于t，所以我们需要遍历所有解，然后一一进行验证（验证方法就是将x带入进行模幂运算，检查是否与β值相等），从而找到属于离散对数的唯一解。由于我们已经产生了一个x解，剩下的解只需要通过$x = x + i * n$这样来遍历即可。

- 本次实验的另外一个重点就是要实现好大数运算，如果大数运算不够快的话，在oj上是会出现超时现象的。（大部分大数运算都在RSA运算中介绍了，本篇实验报告只会挑选一些重要的内容来讲解）

四、实验步骤

- Pollard ρ算法的具体实现

首先，我们需要实现f函数，这个f函数就是我们找到碰撞的核心。我们根据书本上的内容，将其分为3部分，但是我们需要注意的**是必须保证1不属于S_2，因为带入S_2的方程可以发现对于任意$i \geq 0$，都有（1，0，0）属于S_2，而这个我们是无法找到任何碰撞的。**

```
void f(uint32_t x[SMP], uint32_t a[SMP], uint32_t b[SMP])
{
    uint32_t state[SMP] = {0};
    s[0] = 3;
    mod_improve(state, x, s);
    // 因为最后是mod 3的，所以只需要考虑state[0]的值
    if (state[0] == 0)
    {
```

```
            mod_mul_improve(x, x, x, dp);
            mod_add_improve(a, a, a, N);
            mod_add_improve(b, b, b, N);
        }
        else if (state[0] == 1)
        {
            mod_mul_improve(x, x, Y, dp);
            mod_add_improve(b, b, ONE, N);
        }
        else
        {
            mod_mul_improve(x, x, G, dp);
            mod_add_improve(a, a, ONE, N);
        }
    }
```

在这个主体函数中，实现了算法图中的各个步骤，这里便不进行赘述。

其次，我们来观察一下主体循环，这里也是按照算法的流程图来进行初始化的，最后通过gcd函数来判断$b'-b$与n之间是否互素，如果二者互素，那么就通过扩展欧几里得算法（inv_exculid）来求出$b'-b$在n之下的逆元与$a-a'$相乘便可以得到对应的解了。

```
    x[0] = 1;
    X[0] = 1;
    f(x, a, b);
    f(X, A, B);
    f(X, A, B);
    while (bi2str(x) != bi2str(X))
    {
        f(x, a, b);
        f(X, A, B);
        f(X, A, B);
    }
//  ..........................................................................................
    add(res_B, B, N);
    sub(res_B, res_B, b); // b'-b
    // mod_improve(res_B, res_B, N);
    add(res_A, a, N);
    sub(res_A, res_A, A); // a-a'

    gcd(res_B, tmp_N, gcd_num); //gcd(b'-b,n)
    if (equal(gcd_num, ONE)) // 代表其有逆元素
    {
        uint32_t inv[SMP] = {0};
        changeP(N);
        inv_exculid(res_B, tmp_N, tmp_X, tmp_Y);
        mod_mul_mont(inv, tmp_X, res_A);
        cout << bi2str(inv) << endl;

        return 0;
    }
```

最后，我们来看看如果此时二者之间不互素，我们求解线性同余方程的思路：就如同我上面所解释的一样我们可以通过先都除掉对应的gcd值，然后再计算对应的解，然后后面就是循环验证即可。

```
    else // 此时存在多解
    {
        // 此时tmp_B里面存在着我们循环解的多个结果
        // 这里迭代器的值默认是不会超过2^32的
        int iter = stoi(bi2str(gcd_num));
        // 方程两边全部同时除以公约数tmp_B
        div(res_A, res_A, gcd_num); // a = a / gcd
        div(res_B, res_B, gcd_num); // b = b / gcd
        div(tmp_N, N, gcd_num);     // n = n / gcd
        // 此时，由于已经互素了，所以可以通过上面的方法重新得到一个解
        uint32_t inv[SMP] = {0};
        inv_exculid(res_B, tmp_N, inv, tmp_Y);
        mod_mul_improve(inv, inv, res_A, tmp_N); // 此时的inv是众多解中的一个
        uint32_t one[SMP] = {1};
        uint32_t tmp[SMP] = {0};
        uint32_t tmp_ans[SMP] = {0};
        for (int i = 0; i < iter; i++)
        {
            mul(tmp, tmp, tmp_N);
            add(inv, inv, tmp);
            mod_improve(inv, inv, N);
            // 检测当前值是否是离散对数解
            mod_pow_improve(tmp_ans, G, inv, dp);
            if (equal(tmp_ans, Y))
            {
                cout << bi2str(inv) << endl;
            }
            add(tmp, tmp, one);
        }
    }
```

至此，我们便完成了Pollard ρ算法的主体流程

- 大数运算

其实，整体的代码逻辑实现并不困难，困难的主要是实现的大数运算的部分，大数运算就如同本次实验的基石，只有这个实现的好，最后才能真正的完成整个实验。

- 基础函数改造

  在RSA实验中，我已经详细的介绍了大数运算的基本函数，不过由于本次题目中的模数并不是固定的，所以需要对一些基础函数进行改造，具体改造如下：

```
// 改造前
void mod_add(uint32_t res[128], uint32_t a[128], uint32_t b[128])
{
    add(res, a, b);
    mod(res, res);
}
// 改造后
void mod_add_improve(uint32_t res[SMP], uint32_t a[SMP], uint32_t
b[SMP], uint32_t m[SMP])
{
    uint32_t temp[SMP] = {0};
    add(temp, a, b);
    mod_improve(res, temp, m);
}
```

为了能够适应不同的模数要求，本次实验对一些基础函数都进行了改造。

- 其他函数的补充

  为了实现本次实验的其他功能，也相应的补充了一些其他的函数

  gcd函数用来实现求最大公因数，具体实现逻辑运用了欧几里得算法，这里便不过多赘述。

```
void gcd(uint32_t a[SMP], uint32_t b[SMP], uint32_t res[SMP])
{
    uint32_t r[SMP] = {0}, temp[SMP] = {0};
    memcpy(temp, b, SMP * sizeof(uint32_t)); // t = n
    memcpy(res, a, SMP * sizeof(uint32_t));  // x = t
    while (!equal(temp, ZERO))
    {
        mod_add_improve(r, res, ZERO, temp);
        for (int i = 0; i < SMP; ++i)
            swap(res[i], temp[i]);
        for (int i = 0; i < SMP; ++i)
            swap(temp[i], r[i]);
    }
}
```

  求逆函数inv_exculid_improve函数，同样的实现扩展欧几里得的求逆运算

```
void inv_exculid_improve(uint32_t a[SMP], uint32_t b[SMP], uint32_t
x[SMP], uint32_t y[SMP], uint32_t m[SMP])
{
    bool flag = false;
    for (int i = 0; i < SMP; i++)
        if (b[i])
        {
            flag = true;
            break;
        }

    if (!flag)
    {
        x[0] = 1;
        y[0] = 0;
        return;
```

```
    }

    uint32_t temp[SMP] = {0};
    div(temp, a, b);
    mul(temp, b, temp);
    sub(temp, a, temp);

    inv_exculid_improve(b, temp, y, x, m);

    uint32_t temp2[SMP] = {0};
    div(temp2, a, b);
    mul(temp2, temp2, x);

    if (bigger(y, temp2))
    {
        sub(temp2, y, temp2);
        mod_improve(temp2, temp2, m);
    }
    else
    {
        sub(temp2, temp2, y);
        mod_improve(temp2, temp2, m);
        sub(temp2, m, temp2);
    }

    for (int i = 0; i < SMP; i++)
        y[i] = temp2[i];
}
```

这里需要注意的是，在一开始我是直接使用了**基于费马小定律的求逆函数**，但是我发现只有个别样例能够适用，后面思考了一下发现是由于**费马小定律只能用于素数**的情况下，而**扩展欧几里得算法**只需要参与运算的**两个值是互素**的就可以。（不过这边也给出对应的实现）

fermat函数

```
void fermat(uint32_t res[SMP], uint32_t a[SMP], uint32_t m[SMP])
{
    uint32_t temp[SMP];
    sub(temp, m, TWO);
    mod_pow_improve(res, a, temp, m);
}
```

比较函数：equal以及bigger，这两个函数的实现愿你了较为简单，这里便不过多赘述了。

```
// 比较两数大小
bool equal(uint32_t a[SMP], uint32_t b[SMP])
{
    for (int i = 0; i < SMP; i++)
        if (a[i] != b[i])
            return false;
    return true;
}

bool bigger(uint32_t a[SMP], uint32_t b[SMP])
{
```

```
    for (int i = SMP - 1; i >= 0; i--)
    {
        if (a[i] > b[i])
            return true;
        else if (a[i] < b[i])
            return false;
    }
    return false;
}
```

以上便是完成本次实验的所需的一些相对应的大数运算

- 输入输出处理：

  由于本次实验的输入与输出都是字符串形式的，所以我们需要对其进行一些特殊处理

  通过这段代码可以将其转换成对应的二进制表示

```
void str2bi(uint32_t res[SMP], string &s)
{
    // 初始化结果数组
    for (int i = 0; i < SMP; i++)
        res[i] = 0;

    int count = 0; // 用于记录当前位的位置
    while (s != "0")
    {
        // 如果当前最低位为 1，则设置对应的二进制位
        if ((s[0] - '0') & 1)
            res[count / 32] += (1U << (count % 32));

        // 将字符串表示的十进制数除以 2
        int carry = 0;
        for (int i = s.length() - 1; i >= 0; i--)
        {
            int x = s[i] - '0';
            s[i] = (x + carry * 10) / 2 + '0';
            carry = x % 2;
        }
        // 移除字符串右侧的多余零
        size_t end = s.find_last_not_of('0');
        if (end != string::npos)
            s.erase(end + 1);
        else
            s = "0";
        count++; // 更新二进制位计数
    }
}
```

  对应的转换函数就是思路相反过来，这里也不再赘述。

五、实验结果

> 本人学号22336090的离散对数计算结果如下所示：
>
> `1827868284449271 84`

六、实验提升

> 本次实验在他人的提示下，找到了一种提升方法，本次实验的测试样例中最高bit不超过120bit，所以我们可以采取`__uint128_t`的数据结构（不需要引入对应库）来存储相对大的数字，而这类数字就可以用于求取碰撞的过程，函数修改如下：
>
> 可以看到，我们将其中只需要进行简单加一以及乘二的操作都进行了替换，通过使用`__uint128_t`的数据结构，我们可以不直接调用我们之前实现的函数，从而达到加速的效果。

```
void f(uint32_t x[SMP], __uint128_t &a, __uint128_t &b, __uint128_t
n_improve)
{
    uint32_t state[SMP] = {0};
    mod_improve(state, x, s);
    // 因为最后是mod 3的，所以只需要考虑state[0]的值
    if (state[0] == 0)
    {
        mod_mul_improve(x, x, x, dp);
        a = a * 2 % n_improve;
        b = b * 2 % n_improve;
    }
    else if (state[0] == 1)
    {
        mod_mul_improve(x, x, Y, dp);
        b = (b + 1) % n_improve;
    }
    else
    {
        mod_mul_improve(x, x, G, dp);
        a = (a + 1) % n_improve;
    }
}
```

> 对应的转换函数：

```
void uint32ArrayToUint128(const uint32_t n[8], __uint128_t &t)
{
    t = 0; // 初始化 t 为 0
    for (int i = 0; i < 8; ++i)
    {
        // 将数组的每一部分整合到 t 中
        t |= static_cast<__uint128_t>(n[i]) << (i * 32);
    }
}

void uint128ToUint32Array(__uint128_t t, uint32_t n[8])
{
    for (int i = 0; i < 8; ++i)
    {
        n[i] = static_cast<uint32_t>(t & 0xFFFFFFFF); // 提取低 32 位
        t >>= 32;                                      // 右移 32 位
    }
}
```

> 由于我们大部分运算都是需要进行大数运算的，所以还需要写两个转换函数，可以在计算完成之后进行使用。

七、实验总结

> 通过本次实验，我在理论学习和实际动手能力上都得到了显著提升。相比于上次的 RSA 实验，本次实验虽然在实现难度上有所降低，但仍然让我面对了一些新的挑战。一开始，由于对算法的原理理解不够深入，我在求解线性同余方程时遇到了一些问题。在此过程中，通过和同学交流并反复查阅相关资料，我逐渐理解了 Pollard ρ算法的具体实现细节。这不仅帮助我解决了问题，还让我对离散对数问题的整体背景有了更清晰的认识。
>
> 在实验的过程中，我也对欧几里得算法和费马小定律有了更深的理解。与之前仅停留在理论层面的学习不同，这次通过代码的实现和调试，我对它们在大数运算中的实际应用有了更深的体会。这让我意识到理论知识的掌握并不是一蹴而就的，而是需要通过实践来巩固和完善。
>
> 此外，这次实验还让我弥补了之前理论学习中的一个遗憾。复习离散对数相关知识时，我把其他所有的算法都理解完了，但是因为觉得 Pollard ρ算法较为复杂而略过，但通过这次实验，我不仅彻底理解了它的原理，还掌握了课本上没有提到的一种特殊情况的处理方法。这次深入的实践让我感受到学习是一个“回旋镖”的过程，之前看似遗落的部分通过实践得到了补充，真的是受益匪浅。
>
> 最后，这次实验让我更加熟练地调试和实现了与大数运算相关的操作。这种熟练度的提升不仅帮助我更高效地完成了实验，也让我对今后的学习和研究充满了信心。通过理论与实践的结合，我深刻体会到，不管是复杂的理论知识还是实际应用，只要保持耐心和钻研精神，都可以逐步掌握并加以运用。