数理工学セミナー (離散数理) Day1

harsaka

数理工学コース B4

October 18, 2019

Contents

加算による乗算アルゴリズム

加算による乗算アルゴリズム

- 愚直解:$O(n)$
- エジプト乗法 (ロシア農民のアルゴリズム):$O(\log n)$
- 加法連鎖:最小回数を極める

愚直解:multiply0

```
int multiply0(int n, int a){
    if (n == 1) return a;
    return multiply0(n - 1, a) + a;
}
```

愚直解:multiply0

```
int multiply0(int n, int a){
    if (n == 1) return a;
    return multiply0(n - 1, a) + a;
}
```

- $O(n)$
- 遅すぎる！

愚直解:multiply0(非再帰)

- 今のは関数の中で自分を呼び出す『再帰関数』だった.
- 非再帰で同じ内容のものを書けるか?

愚直解:multiply0(非再帰)

- 今のは関数の中で自分を呼び出す『再帰関数』だった.
- 非再帰で同じ内容のものを書けるか?
- 今回は書ける (と思います).

愚直解:multiply0(非再帰)

```
int non_rec_multiply0(int n, int a){
    int result = a;
    for (int i = 0; i < n - 1; i++){
        result += a;
    }
    return result;
}
```

愚直解:multiply0(非再帰)

```c
int non_rec_multiply0(int n, int a){
    int result = a;
    for (int i = 0; i < n - 1; i++){
        result += a;
    }
    return result;
}
```

- 当然 $O(n)$
- 非再帰の利点:計算量が分かりやすい！

エジプト乗法:multiply1

```
int multiply1(int n, int a){
    if (n == 1) return a;
    int result = multiply1(half(n), a + a);
    if (odd(n)) result = result + a;
    return result;
}
```

エジプト乗法:multiply1

```
int multiply1(int n, int a){
    if (n == 1) return a;
    int result = multiply1(half(n), a + a);
    if (odd(n)) result = result + a;
    return result;
}
```

- odd(x):x の最下位 (2 進数表記して一番右側の) ビットが立っているかを判定.
- half(x):x を 2 で割る演算.

エジプト乗法:multiply1

```
bool odd(int n){
    return n & 0x1;
}
int half(int n){
    return n >> 1;
}
```

エジプト乗法:multiply1

```
bool odd(int n){
    return n & 0x1;
}
int half(int n){
    return n >> 1;
}
```

- odd(x):x の最下位 (2 進数表記して一番右側の) ビットが立っているかを判定.
- half(x):x を 2 で割る演算.

エジプト乗法:multiply1

- n を 2 進数表記する.
- その i 桁目に bit が立っていれば $2^i * a$ を加える.

エジプト乗法:multiply1

- n を 2 進数表記する.
- その i 桁目に bit が立っていれば $2^i * a$ を加える.
- 計算量は $O(\log(n))$

エジプト乗法:multiply1

- n を 2 進数表記する.
- その i 桁目に bit が立っていれば $2^i * a$ を加える.
- 計算量は $O(\log(n))$
- これも非再帰で書いてみる.

エジプト乗法:multiply1(非再帰)

```
int non_rec_multiply1(int n, int a){
    int res = 0;
    while (n){
        if (odd(n)) res = res + a; // v(n)
        n = half(n);
        a = a + a; //floor(log(n))
    }
    return res;
}
```

エジプト乗法:multiply1(非再帰)

```
int non_rec_multiply1(int n, int a){
    int res = 0;
    while (n){
        if (odd(n)) res = res + a; // v(n)
        n = half(n);
        a = a + a; //floor(log(n))
    }
    return res;
}
```

- $O(\log(n))$.

エジプト乗法:multiply1(非再帰)

```
int non_rec_multiply1(int n, int a){
    int res = 0;
    while (n){
        if (odd(n)) res = res + a; // v(n)
        n = half(n);
        a = a + a; //floor(log(n))
    }
    return res;
}
```

- $O(\log(n))$.
- 累乗 a^n の計算も同様に行える.

累乗 a^n を高速計算

```
int pow(int n, int a){
    int res = 1;
    while (n){
        if(odd(n)) res = res * a;
        n = half(n);
        a = a * a;
    }
    return res;
}
```

累乗 a^n を高速計算

```
int pow(int n, int a){
    int res = 1;
    while (n){
        if(odd(n)) res = res * a;
        n = half(n);
        a = a * a;
    }
    return res;
}
```

- 積の演算を O(1) と仮定している点に注意.

累乗 a^n を高速計算

```
int pow(int n, int a){
    int res = 1;
    while (n){
        if(odd(n)) res = res * a;
        n = half(n);
        a = a * a;
    }
    return res;
}
```

- 積の演算を O(1) と仮定している点に注意.
- 競プロでたまに使います.

最適な加法連鎖

```
int multiply_by_15(int a){
    int b = (a + a) + a;
    int c = b + b;
    return (c + c) + b;
}
```

最適な加法連鎖

```
int multiply_by_15(int a){
    int b = (a + a) + a;
    int c = b + b;
    return (c + c) + b;
}
```

- $15 * a$ の計算回数はこちらの方が 1 回少ない.

最適な加法連鎖

```
int multiply_by_15(int a){
    int b = (a + a) + a;
    int c = b + b;
    return (c + c) + b;
}
```

- 15 * a の計算回数はこちらの方が 1 回少ない.
- 加法連鎖は和による積の計算回数の最小値を与える (すごい).

最適な加法連鎖

```c
int multiply_by_15(int a){
    int b = (a + a) + a;
    int c = b + b;
    return (c + c) + b;
}
```

- 15 * a の計算回数はこちらの方が 1 回少ない.
- 加法連鎖は和による積の計算回数の最小値を与える (すごい).
- 100 までの加法連鎖を求める (問題 2.1)

最適な加法連鎖

- "1"が長さ 0(加算 0 回) の加法連鎖で計算されるとして帰納的に計算.

最適な加法連鎖

- "1"が長さ 0(加算 0 回) の加法連鎖で計算されるとして帰納的に計算.
- 長さ k の加法連鎖で計算される数字の集合 chain[k] とその各数字に辿り着くまでの連鎖過程の数列 ancestor[chain[k][j]] が求まっていると仮定.

最適な加法連鎖

- "1"が長さ 0(加算 0 回) の加法連鎖で計算されるとして帰納的に計算.
- 長さ k の加法連鎖で計算される数字の集合 chain[k] とその各数字に辿り着くまでの連鎖過程の数列 ancestor[chain[k][j]] が求まっていると仮定.
- cur:=chain[k][j] とすると, ancestor[cur] の要素の和で表せる数字が長さ $k+1$ で初めて加法連鎖をなす候補となる.

最適な加法連鎖

- "1"が長さ 0(加算 0 回) の加法連鎖で計算されるとして帰納的に計算.
- 長さ k の加法連鎖で計算される数字の集合 chain[k] とその各数字に辿り着くまでの連鎖過程の数列 ancestor[chain[k][j]] が求まっていると仮定.
- cur:=chain[k][j] とすると, ancestor[cur] の要素の和で表せる数字が長さ $k+1$ で初めて加法連鎖をなす候補となる.
- ancestor[] の要素数と要素がそのまま最適な加法連鎖の長さと元になっている.

最適な加法連鎖

- "1"が長さ 0(加算 0 回) の加法連鎖で計算されるとして帰納的に計算.
- 長さ k の加法連鎖で計算される数字の集合 chain[k] とその各数字に辿り着くまでの連鎖過程の数列 ancestor[chain[k][j]] が求まっていると仮定.
- cur:=chain[k][j] とすると, ancestor[cur] の要素の和で表せる数字が長さ $k+1$ で初めて加法連鎖をなす候補となる.
- ancestor[] の要素数と要素がそのまま最適な加法連鎖の長さと元になっている.
- 長さ k=9 まであれば 100 までの任意の数の加法連鎖が求められる.

最適な加法連鎖

- 実装する.
- https://github.com/Harsaka/Srkg_seminar/blob/master/addition_chain.cpp

再帰アルゴリズムの改良 multi_acc0

```
int multi_acc0(int r, int n, int a){
    if (n == 1) return r + a;
    if (odd(n)){
        return multi_acc0(r + a, half(n), a + a);
    }else{
        return multi_acc0(r, half(n), a + a);
    }
}
```

再帰アルゴリズムの改良 multi_acc0

```
int multi_acc0(int r, int n, int a){
    if (n == 1) return r + a;
    if (odd(n)){
        return multi_acc0(r + a, half(n), a + a);
    }else{
        return multi_acc0(r, half(n), a + a);
    }
}
```

- 末尾再帰を目指す.

再帰アルゴリズムの改良 multi＿acc0

```
int multi_acc0(int r, int n, int a){
    if (n == 1) return r + a;
    if (odd(n)){
        return multi_acc0(r + a, half(n), a + a);
    }else{
        return multi_acc0(r, half(n), a + a);
    }
}
```

- 末尾再帰を目指す.
- まだ末尾再帰ではない.

末尾再帰 multi_acc1

```
int multi_acc1(int r, int n, int a){
    if (n ==1) return r + a;

    if (odd(n)) r = r + a;
    return multi_acc1(r, half(n), a + a);
}
```

末尾再帰 multi_acc1

```
int multi_acc1(int r, int n, int a){
    if (n ==1) return r + a;

    if (odd(n)) r = r + a;
    return multi_acc1(r, half(n), a + a);
}
```

- 末尾再帰になった.

末尾再帰 multi_acc1

```
int multi_acc1(int r, int n, int a){
    if (n ==1) return r + a;

    if (odd(n)) r = r + a;
    return multi_acc1(r, half(n), a + a);
}
```

- 末尾再帰になった.
- 繰り返し (for 文, while 文) で書き直しやすくなる！ (つまり構造としてシンプルで良い形式)

末尾再帰 multi_acc2

```
int multi_acc2(int r, int n, int a){
    if (odd(n)){
        r = r + a;
        if(n == 1)return r;
    }
    return multi_acc2(r, half(n), a + a);
}
```

末尾再帰 multi_acc2

```
int multi_acc2(int r, int n, int a){
    if (odd(n)){
        r = r + a;
        if(n == 1)return r;
    }
    return multi_acc2(r, half(n), a + a);
}
```

- n == 1 の判定は偶数に対しては不要.

末尾再帰 multi_acc2

```
int multi_acc2(int r, int n, int a){
    if (odd(n)){
        r = r + a;
        if(n == 1)return r;
    }
    return multi_acc2(r, half(n), a + a);
}
```

- n == 1 の判定は偶数に対しては不要.
- 初めに偶奇判定することで計算回数を半減させられる！

正確な末尾再帰 multi_acc3

```
int multi_acc3(int r, int n, int a){
    if(odd(n)){
        r = r + a;
        if (n == 1)return r;
    }
    n = half(n);
    a = a + a;
    return multi_acc3(r, n, a);
}
```

正確な末尾再帰 multi_acc3

```
int multi_acc3(int r, int n, int a){
    if(odd(n)){
        r = r + a;
        if (n == 1)return r;
    }
    n = half(n);
    a = a + a;
    return multi_acc3(r, n, a);
}
```

- 再帰呼び出しの引数の形が一緒.

正確な末尾再帰 multi_acc3

```
int multi_acc3(int r, int n, int a){
    if(odd(n)){
        r = r + a;
        if (n == 1)return r;
    }
    n = half(n);
    a = a + a;
    return multi_acc3(r, n, a);
}
```

- 再帰呼び出しの引数の形が一緒.
- ここまでくると簡単に繰り返し (非再帰) で書き換えられる.

非再帰 multi_acc4

```
int multi_acc4(int r, int n, int a){
    while (true){
        if (odd(n)){
            r = r + a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a + a;
    }
}
```

改善 multi_acc4 を multiply2 に適用

```
int multiply2(int n, int a){
    if (n == 1) return a;
    return multi_acc4(a, n - 1, a);
}
```

改善 multi_acc4 を multiply2 に適用

```
int multiply2(int n, int a){
    if (n == 1) return a;
    return multi_acc4(a, n - 1, a);
}
```

- multi_acc4(a, n - 1, a) というように r=a で呼ぶことで加算を 1 回減らしている．

改善 multi_acc4 を multiply2 に適用

```
int multiply3(int n, int a){
    while (!odd(n)){
        a = a + a;
        n = half(n);
    }
    return multiply2(n, a);
}
```

改善 multi_acc4 を multiply2 に適用

```
int multiply3(int n, int a){
    while (!odd(n)){
        a = a + a;
        n = half(n);
    }
    return multiply2(n, a);
}
```

- multi_acc4(a, n - 1, a) と関数では n-1 を呼ぶので，計算回数を減らすために n が偶数のときには先にシフト演算をしてしまう．

改善 multi_acc4 を multiply2 に適用

```
int multiply4(int n, int a){
    while (!odd(n)){
        a = a + a;
        n = half(n);
    }
    if (n == 1) return a;
    return multi_acc4(a, half(n - 1), a + a);
}
```

改善 multi_acc4 を multiply2 に適用

```
int multiply4(int n, int a){
    while (!odd(n)){
        a = a + a;
        n = half(n);
    }
    if (n == 1) return a;
    return multi_acc4(a, half(n - 1), a + a);
}
```

- n-1 を偶数に調整したので n-1 は 1 にならない.

改善 multi_acc4 を multiply2 に適用

```
int multiply4(int n, int a){
    while (!odd(n)){
        a = a + a;
        n = half(n);
    }
    if (n == 1) return a;
    return multi_acc4(a, half(n - 1), a + a);
}
```

- n-1 を偶数に調整したので n-1 は 1 にならない.
- multi_acc4 を呼び出すときにあと 1 回だけシフト演算をして完成.