
 **penguin72487** · Follow
Last edited by penguin72487 on May 23, 2024
Contributed by 

心得

我覺得BCD64比起10¹⁸進位一點都不效率，不管是編碼、運算、實現都沒有10¹⁸進位有效率。我學到要運算子多載"="來達成賦予值得操作要使用T& operator=()，之前都搞不清楚要怎麼賦予多載。還學到怎麼用BCD64可以用位元遮罩來運算。還發現現在的GPT好方便喔，不僅給我BCD64的想法，還幫我用py寫測試，不然我根本看不出來最後一個測資有錯誤，而且學到一個reverse跟遞迴都不用的大數運算。

Design of your linked list: 10%

有分輸入並編碼，運算，輸出，我完全不使用reverse或是雙向鏈，
編碼區:我的大數最低位是head，最高位是tail，輸入的時候切16個字元給BCD64。
期中我有用一個優化，c-'0'改成c^48。

運算區:使用運算子多載，在使用上比較直觀，從最低位先算，有沒有進位從大數這裡控制，如果兩個數字位數不一樣，就先算完位數一樣的部分，後面全部push_back就好。

輸出區:原本使用遞迴到最高位再輸出回來
但是遞迴叫函數，但是這樣耗stack區的空間，只能算10^6位左右，而且效率低，耗時是編碼的100倍，換成從頭把所有數字都輸到string，再用index控制從尾巴輸出，快到起飛，跟編碼差不多快。

Review/Improve class BCD64

how does BCD64 work?

BCD (Binary Coded Decimal) 是一種數字儲存格式，每一位十進位數字都被單獨編碼成四位二進位數。例如，十進位數字 93 在 BCD 中會被表示為 1001 0011，如果是BCD64，把16個BCD串在一起，總共耗64bit可以存在unsigned long long裡面

如果要做加法的話
就要用位元遮罩一位BCD一位BCD加起來，把進位傳下去。

在加法中，如果有進位，就要+6
1 8 9 7
0001 1000 1001 0111
&
0000 0000 0000 1111
+
2 9 0 5
0010 1001 0000 0101
&
0000 0000 0000 1111
+
Carry
0
=
0000 0000 0000 1100
+
0000 0000 0000 0110

=
0000 0000 0001 0010
Carry
1
然後把兩個數字右移4bit，繼續算。

Analyze the space used to store BCD large numbers using linked lists? Is it optimal (or ideal)? Why?

如果有n位數(10進位下)，每一位用4bit，
64bit可以存儲16位數字，每16位數字要再用64bit存指標，然後大數自己要存頭尾的指標，所以有n位數，需要的空間是
 $\lceil \frac{n}{16} \rceil * 2 * 64 + 2 * 64(\text{bit})$ 的空間。
如果要是省空間的話可以使用連續的動態陣列空間，可以不用存指標，可以省一半的空間。

還有可以用8Byte 做10¹⁸次方進位，
需要
 $\lceil \frac{n}{18} \rceil * 2 * 64 + 2 * 64(\text{bit})$ 的空間。
或是乾脆用成2⁶⁴進位
 $\lceil \frac{n}{\log_{10}(2^{64})} \rceil * 2 * 64 + 2 * 64(\text{bit})$ 的空間。
log₁₀(2⁶⁴)約等於19.26

how to improve BCD64 so that the performance can be raised

```
BCD64 FA(BCD64 &num, unsigned char *carryout = nullptr, unsigned char carryin) {
    static const uint64_t bcd_add6[2] = {0, 6};
    uint64_t add_6s = 0;
    unsigned char carry = carryin;
    for (int i = 0; i < sizeof(bcd) << 1; i++) {
        uint8_t bcd_4bit = ((bcd & MASK1111(i)) >> (i << 2));
        bcd_4bit += ((num.bcd & MASK1111(i)) >> (i << 2));
        bcd_4bit += carry;
        carry = (bcd_4bit >= 10);
        add_6s |= bcd_add6[carry] << (i << 2);
    }
    BCD64 sum(bcd + num.bcd + carryin + add_6s);
    if (carryout)
        *carryout = carry;
    return sum;
}
```

我覺得助教寫的還蠻優化的，指示carryout可以優化出去，使用傳參考，會比傳地址更快，這個地址算是一種value，傳進來的時候會copy一次，carryout跟carryin可以合併。
變成carry，還有sizeof(bcd) << 1，要注意編譯器有沒有優化，沒優化要把常數提出來。

(bcd_4bit >= 10);也可以改成>9，會再快一點，雖然編譯器優化也可能會優化就是了。

```
BCD64 FA(BCD64 &num, unsigned char &carry) {
    static const uint64_t bcd_add6[2] = {0, 6};
    uint64_t add_6s = 0;
    for (int i = 0; i < sizeof(bcd) << 1; i++) {
        uint8_t bcd_4bit = ((bcd & MASK1111(i)) >> (i << 2));
        bcd_4bit += ((num.bcd & MASK1111(i)) >> (i << 2));
        bcd_4bit += carry;
        carry = (bcd_4bit >9);
        add_6s |= bcd_add6[carry] << (i << 2);
    }
    BCD64 sum(bcd + num.bcd + carry + add_6s);
    return sum;
}

BCD64 FM(BCD64 &num, unsigned char &borrow) {
    static const uint64_t bcd_sub10[2] = {0, 10};
    uint64_t sub_10s = 0;
    for (int i = 0; i < sizeof(bcd) << 1; i++) {
        uint8_t bcd_4bit = ((bcd & MASK1111(i)) >> (i << 2));
        bcd_4bit -= ((num.bcd & MASK1111(i)) >> (i << 2));
        bcd_4bit -= borrow;
        borrow = (bcd_4bit >9);
        sub_10s |= bcd_sub10[borrow] << (i << 2);
    }
    BCD64 diff(bcd - num.bcd - borrow - sub_10s);
    return diff;
}
```

但是這些優化只是壓常數而已，沒有什麼本質上的加快。

Last changed by 

 **penguin72487** · Follow

💬 0 ❤️ 👁 9 📌