CSCI203
Algorithms and Data Structures

Big Numbers

Lecturer: Dr. Xueqiao Liu
Room 3.117
Email: xueqiao@uow.edu.au

Standard Integers in C/C++

type	Storage size	Value range
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned int	4 bytes	0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

Big Numbers

- Big numbers, numbers larger than 2^{64}, are important in many applications in computing:
  - Perfect hashing – needs a prime number > |U| the size of the universe of keys.
  - Cryptography – operates with numbers of 512, 1024 or even more bits.
  - Arbitrary precision arithmetic, e.g. calculating π to a million decimal digits.
- Efficient calculation using big numbers is worth examining as it is useful in its own right and provides some useful methods that have wider application.
  - Divide and Conquer.

Big Numbers

- We will start by looking at:
  - How to represent large numbers;
  - How to add them;
  - How to multiply them;
  - How to raise them to large powers.

Large Number Representation

- This is the easiest question to address:
- We simply break the number into an ordered sequence of manageable chunks.
- We do this already…
- …It is called decimal notation.
  - e.g. we represent 2^{20} as 1048576
  - where each digit is a chunk in a sequence of powers of ten.

Decimal Numbers

- 1048576
  - $1 \times 10^6 + 0 \times 10^5 + 4 \times 10^4 + 8 \times 10^3 + 5 \times 10^2 + 7 \times 10^1 + 6 \times 10^0$
- We can do the same using any numeration base:
- The same number can be written as:
  - 100000000000000000000 in base 2 (digits are 01);
  - 1222021101011 in base 3 (digits are 012);
  - 232023301 in base 5 (digits are 01234);
  - 11625034 in base 7 (digits are 0123456);
  - $c974g$ in base 17 (digits are 0123456789abcdefg);
  - $18p2g$ in base 30 (digits are 0123456789abcdefghijklmnopqrst).

Big Numbers - Big bases

- If we wish to represent large numbers we can break then up into chunks, each of which fits a computer word:
  - 32 bits;
  - 64 bits.
- In this way a 1024-bit number can be represented as a sequence of:
  - 32 32-bit integers;
  - 16 64-bit integers.
- We choose the largest integer for which we can easily and accurately calculate sums and products.

Addition

- To add two large integers we note:
- Integers x and y can be written in base b as follows:
  - $x = x_k \times b^k + x_{k-1} \times b^{k-1} + \ldots + x_0 \times b^0$
  - $y = y_k b^k + y_{k-1} \times b^{k-1} + \ldots + y_0 \times b^0$
- We can the write $x + y$ as:
  - $x + y = (x_k + y_k) \times b^k + (x_{k-1} + y_{k-1}) \times b^{k-1} + \ldots + (x_0 + y_0) \times b^0$
- We simply add the corresponding chunks of the two numbers (plus any possible carry) to get the equivalent chunk of the result.

Addition's Efficiency

- If we add two k-chunk integers this involves calculating k additions, each of b-digit integers.
- We can add two integers in k operations.
  - $k = \log_b n$.
- This is pretty good.

Multiplication

- Multiplication is a bit harder.
- The product $x \times y$ involves calculating products of all of the chunks of each number taken in pairs, each product being multiplied by an appropriate power of the base:

$$\begin{aligned} x \times y &= \left(x_k \times b^k + x_{k-1} \times b^{k-1} + \ldots + x_0 \times b^0\right) \times \left(y_k \times b^k + y_{k-1} \times b^{k-1} + \ldots + y_0 \times b^0\right) \\ &= x_k \times y_k \times b^{2k} + (x_k \times y_{k-1} + x_{k-1} \times y_k) \times b^{2k-1} + \cdots + x_0 \times y_0 \times b^0 \end{aligned}$$

- Note: each product of two numbers has up to twice as many bits as the original numbers.

An Example

- Let us calculate the product of 12345×56789 using base- 100 chunks.

- 12345 consists of 3 chunks: $01\ 23\ 45$
- 56789 consists of 3 chunks: $05\ 67\ 89$

- We calculate the product as follows:


01 05 00 05
12345
56789
01 67 00 67
01 23 45 05 67 89
01 89 00 89
23 05 01 15
23 67 15 41
23 89 20 47
45 05 02 25
45 67 30 15
45 89 40 05
01 02 01
00 07 01 06 02 05
701060205

56789 X 12345

			05	67	89
		×	01	23	45
				40	05
			30	15	
		02	25		
			20	47	
		15	41		
	01	15			
		00	89		
	00	67			
00	05				
	01	02	01		
00	07	01	06	02	05

56789			12345		
05	67	89	01	23	45

701060205

Analysis

- To multiply two 3-chunk integers involved:
  - 9 multiplications;
  - A similar number of additions.
- In general multiplying two n-digit integers together involves:
  - $O(n^2)$ multiplications;
  - $O(n^2)$ additions.
- Can we do better?
  - Multiplication is much slower than addition.

Karatsuba Multiplication

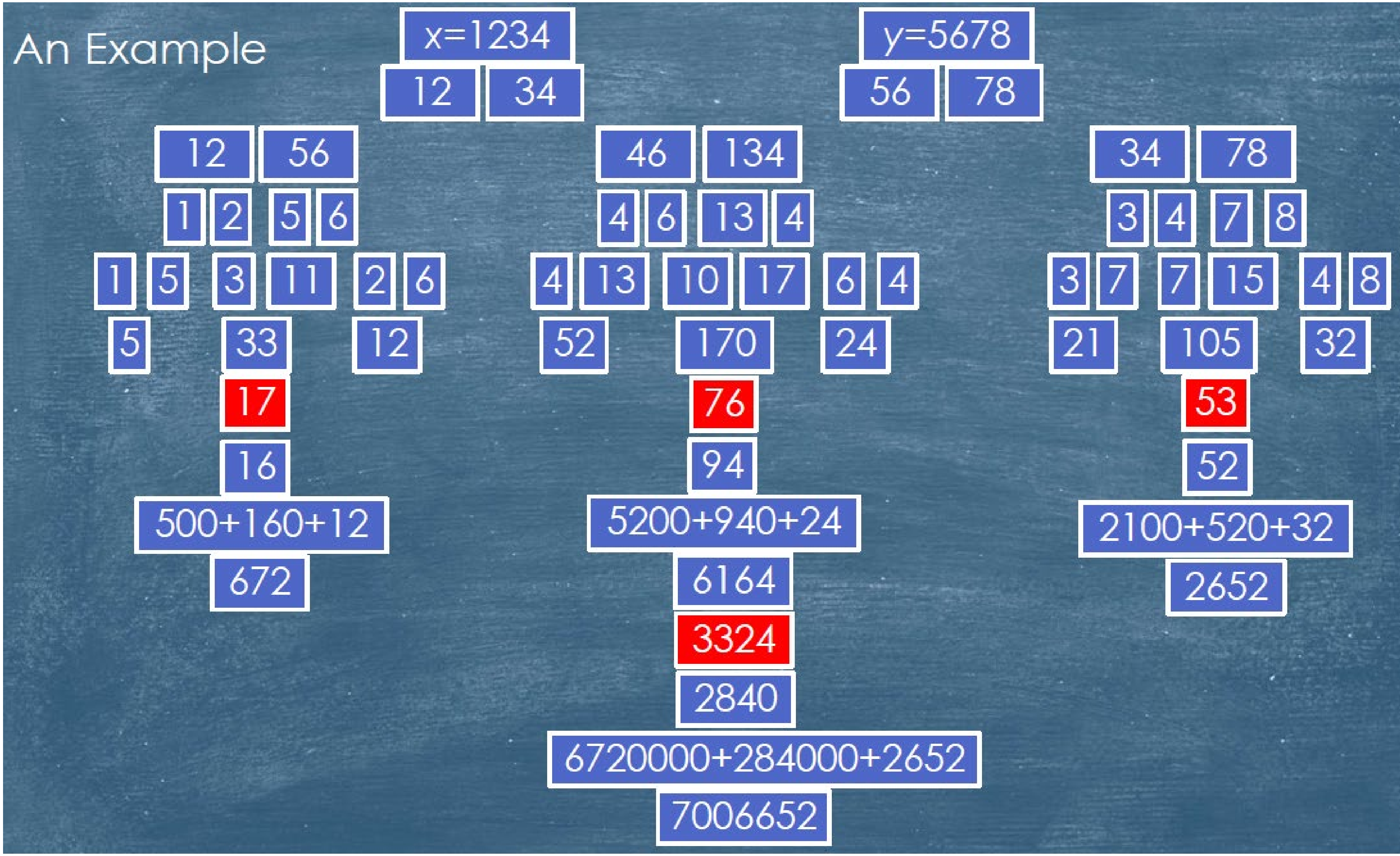
- Instead of breaking each number into chunks (with length n) let us simply split them into two pieces.
  - $x = b^{n/2} \times x_H + x_L$
  - $y = b^{n/2} \times y_H + y_L$
  - Then $x \times y =$
    - $b^n (x_H \times y_H) + b^{\frac{n}{2}} \times ((x_H \times y_L) + (x_L \times y_H)) + (x_L \times y_L)$
  - This involves 4 multiplications, 2 additions and 2 shifts (assuming b is a power of 2).
  - If we keep dividing into smaller chunks we still end up with $O(n^2)$.
- Let us look at the multiplications in more detail.

Karatsuba Multiplication…

- We calculated four results;
  - $x_H \times y_H$;
  - $x_H \times y_L$;
  - $x_L \times y_H$;
  - $x_L \times y_L$.
- We actually only need three;
  - $x_H \times y_H$;
  - $(x_H + x_L) \times (y_H + y_L)$;
  - $x_L \times y_L$.
- Why is this?

Karatsuba Multiplication…

- Consider the three terms we need to calculate the product:
  - $x_H \times y_H$;
  - $x_H \times y_L + x_L \times y_H$;
  - $x_L \times y_L$.
- Our three multiplications allow us to evaluate all of these terms as follows:
  - $x_H \times y_H$;
  - $(x_H + x_L) \times (y_H + y_L) - (x_H \times y_H + x_L \times y_L)$;
  - $x_L \times y_L$.
- Now, if we keep dividing down, our multiplication takes $O(n^{\log_2 3})$, instead of $O(n^2)$


An Example
x=1234
12
34
y=5678
56
78
12
56
46
134
34
78
1
2
5
6
4
6
13
4
3
4
7
8
1
5
3
11
2
6
4
13
10
17
6
4
3
7
7
15
4
8
5
33
12
52
170
24
21
105
32
17
76
53
16
94
52
500+160+12
5200+940+24
2100+520+32
672
6164
2652
3324
2840
6720000+284000+2652
7006652

Analysis

- Our example required 9 multiplications.
  - $4^{\log_2 3} = 9$
- Brute force would need 16 multiplications
  - $4^2 = 16$
- If we have longer numbers, the advantage becomes greater:
  - 10 digits: 100 vs. ~38
  - 100 digits: 10000 vs. ~1479

Further Analysis

- Note that, although we perform fewer multiplications we perform many more additions.
- The break-even point will vary, depending on the processor characteristics.
- Generally, when the numbers to be multiplied are more than 320 bits long (10 words) we get an advantage.
- Typically, multiplications of this type are calculated modulo a number which is also of similar size.
- This further reduces the number of operations required.

Powers

- To evaluate x^y where x and y are both large integers is a daunting task.
- We recall that:
  - $x^y = x \times x \times x \times \cdots \times x$
  - $y - 1$ multiplications.
- This will take an unacceptable number of operations to complete.
- Can we improve on $O(y)$ multiplications?

Fast Powers

- We can represent x^y recursively as follows:
  - $x^y = \left(x^{\frac{y}{2}}\right)^2$ if y is even;
  - $x^y = x \times x^{y-1}$ if y is odd.
- Thus, for example,

$$
\begin{aligned}
a^{29} &= a \times a^{28} = a \times (a^{14})^2 \\
&= a \times \left((a^7)^2\right)^2 \\
&= a \times \left((a \times a^6)^2\right)^2 \\
&= a \times \left((a \times (a^3)^2)^2\right)^2 \\
&= a \times \left(\left(a \times \left(a \times (a^2)\right)^2\right)^2\right)^2
\end{aligned}
$$

Analysis

- We note that at least half of the operations involved in fast_power reduce the power by a factor of two.
  - If y is odd at some iteration, it is even next time.
  - If y is even, there is 50% chance that it will be even next time.
  - Even in the worst case, alternating odd and even values, the value of x^y will be computed in $O(\log y)$ multiplications.
- This is a big improvement on $O(y)$, e.g. for 1000-bit numbers:
  - Conventional power computation would take $O(2^{1000})$ operations;
  - Fast power computation would take $O(1000) = O(2^{10})$ operations;
  - This is a factor of 2^{990} times faster!

Even Faster

- We can also create an iterative version:

```
procedure fast_power_iter(x, y)
    i = y
    result = 1
    a = x
    while i > 0 do
        if i is odd result = big_mult(result, a)
        a = big_mult(a, a)
        i = ⌊i/2⌋
return result
```

- This version removes the cost of the recursive calls.

Modular Powers

- In practice, we compute all of these results modulo m, where m is yet another large integer.
- $(ab) \, mod \, m = ((a \, mod \, m)(b \, mod \, m)) \, mod \, m$
- This gives us the modular power procedure:

```
procedure mod_power(x, y)
    i = y
    result = 1
    a = x
    while i > 0 do
        if i is odd result = mod(big_mult(result,
        a),m)
        a = mod(big_mult(a, a), m)
        i = ⌊i/2⌋
return result
```