Course Notes for

CS 1501
Algorithm Implementation

By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

LZW vs Huffman

- In practice, which is better, LZW or Huffman?
  - For most files, LZW gives better compression ratios
  - It is also generally better for compressing archived directories of files
  - Why?

    •Files can build up very long patterns, allowing LZW to get a great deal of compression

    •As long as we have codewords available, different file types do not "hurt" each other with LZW as they do with Huffman – with each type we simply have to build up new patterns

- Let's compare
  - See (old) compare.txt handout
  - Note that for a single text file, Huffman does pretty well
  - For large archived file, Huffman does not do as well
  - gzip outperforms Huffman and LZW
    - Combo of LZ77 and Huffman
    - See: http://en.wikipedia.org/wiki/Gzip
      http://www.gzip.org/
      http://www.gzip.org/algorithm.txt

- As discussed previously, other good compression algorithms exist
  - Some are similar to those we have discussed
    - Ex: Deflate (LZ77 + Huffman) (used in gzip and pkzip)
      - http://en.wikipedia.org/wiki/DEFLATE
    - Ex: LZMA (another variation of LZ77) (used by 7-zip)
      - http://en.wikipedia.org/wiki/LZMA
  - Some are completely different approaches
    - Ex: bzip2 using multiple "stacked" algorithms
      - http://en.wikipedia.org/wiki/Bzip2

Limitations on Compression and Entropy

- Goal for new algorithms is always to improve compression (without costing too much in the run-time)
- This leads to a logical question:
  - How much can we compress a file (in a lossless way)?
    - Ex: Take a large file, and compress it K times
    - If K is large enough, maybe I can compress the entire file down to 1 bit!
  - Of course this won't work, but why not?
    - Clearly, we cannot unambiguously decompress this file – we could make an infinite number of "original" files from out 1 bit file

Limitations on Compression and Entropy

- Generally speaking, the amount we can compress a file is dependent upon the amount of **entropy** in the data
  - Informally, entropy is the amount of uncertainty / randomness in the data
    - Information entropy is very similar in nature to thermodynamic entropy, which you may have discussed in a physics or chemistry class
  - The more entropy, the less we can compress, and the less entropy the more we can compress
    - Ex: File containing all A's can be heavily compressed since all of the characters are the same
    - Ex: File containing random bits cannot be compressed at all

- When we compress a file (ex: using compress or gzip) we are taking patterns / repeated sequences and substituting codewords that have much more entropy
- Attempts to compress the result (even using a different algorithm) may have little or no further compression
  - However, in some cases it may be worth trying, if the two algorithms are very different
- For more info, see:
- http://en.wikipedia.org/wiki/Lossless_data_compression
- http://en.wikipedia.org/wiki/Information_entropy

- Integer Multiplication
  - With predefined int variables we think of multiplication as being Theta(1)
    - Is the multiplication really a constant time op?
      - No -- it is constant due to the constant size of the numbers (32 bits), not due to the algorithm
  - What if we need to multiply very large ints?
    - Ex: RSA that we will see shortly needs ints of sizes up to 2048 bits
    - We must do this in software
  - Now we need to think of good integer multiplication algorithms

Integer Multiplication

- **GradeSchool algorithm:**
  - Multiply our integers the way we learn to multiply in school
    - However, we are using base 2 rather than base 10
    - Try a long multiplication example and think how it is done
  - Run-time of algorithm?
    - We have two nested loops:
      - >Outer loop goes through each bit of first operand
      - >Inner loop goes through each bit of second operand
    - Total runtime is Theta(N^2)
  - How to implement?
    - We need to be smart so as not to waste space
    - The way we learn in school has "partial products" that can use Theta(N^2) memory

Integer Multiplication

RED digits are subproducts - each represents the product of the top number by one digit of the bottom number

GREEN digits are sums of subproducts – to save memory these can be done incrementally – add "as we go" rather than having to store all subproducts (which would require Theta(N^2) memory)

All of the rows are shown here, but note that we only really need 3 rows at time – which is Theta(N) memory

```
       10010110
       10110111
       --------
       10010110
      10010110
      ---------
      111000010
     10010110
     ----------
    10000011010
    00000000
    -----------
    10000011010
   10010110
   ------------
   110101111010
  10010110
  -------------
 10000000111010
 00000000
 --------------
 10000000111010
10010110
---------------
110101100111010
```

- Can we improve on Theta(N^2)?
  - How about if we try a divide and conquer approach?
  - Let's break our N-bit integers in half using the high and low order bits:

```
X = 1001011011001000
  = 2^(N/2)(X_H) + X_L
where X_H = high bits = 10010110
      X_L = low bits  =         11001000
                         ----------------
      X =                1001011011001000
```

$$X = 2^{N/2}(X_H) + X_L$$

More Integer Multiplication

- Let's look at this in some more detail
  - Recall from the last slide how we break our N-bit integers in half using the high and low order bits:

$$X = 1001011011001000$$
$$= 2^{N/2}(X_H) + X_L$$
$$\text{where } X_H = \text{high bits} = 10010110$$
$$X_L = \text{low bits} = 11001000$$

  - Given two N-bit numbers, X and Y, we can then re-write each as follows:

$$X = 2^{N/2}(X_H) + X_L$$
$$Y = 2^{N/2}(Y_H) + Y_L$$

  - Note that these are both **binomials**

- Now, the product, X*Y, can be written as:

$$\mathbf{XY} = (2^{N/2}(X_H) + X_L) * (2^{N/2}(Y_H) + Y_L)$$
$$= 2^N X_H Y_H + 2^{N/2}(X_H Y_L + X_L Y_H) + X_L Y_L$$

- Note the implications of this equation
  - The multiplication of 2 N-bit integers (XY) is being defined in terms of
    - 4 multiplications of N/2 bit integers (note sub-products)
    - Some additions of ~N bit integers
    - Some shifts (up to N positions, for powers of 2)
  - But what does this tell us about the overall runtime?

- How to analyze the divide and conquer algorithm?
  - Analysis is more complicated than iterative algorithms due to recursive calls
  - For recursive algorithms, we can do analysis using a special type of mathematical equation called a Recurrence Relation
    - see http://en.wikipedia.org/wiki/Recurrence_relation
    - Idea is to determine two things for the recursive calls
      1) How much work is to be done during the current call, based on the current problem size?
      2) How much work is "passed on" to the recursive calls?

- Let's examine the recurrence relation for the divide and conquer multiplication algorithm
  - We will assume that the integers are divided exactly in half at each recursive call
    - Original number of bits must be a power of 2 for this to be true

1) Work at the current call is due to shifting and binary addition. For an N-bit integer this should require operations proportional to N
2) Work "passed on" is solving the same problem (multiplication) 4 times, but with each of half the original size
   - $X_H Y_H$, $X_H Y_L$, $X_L Y_H$, $X_L Y_L$

- So we write the recurrence as

T(N) = 4T(N/2) + Theta(N)

  - Or, in words, the operations required to multiply 2 N-bit integers is equal to 4 times the operations required to multiply 2 N/2-bit integers, plus the ops required to put the pieces back together
- Now we need to **solve this recurrence**
  - The recurrence gives a relationship between terms in a sequence of terms
  - We want a Theta run-time in direct terms of N – i.e. we want to remove the recursive component
  - There are a number of techniques that can accomplish this

- Let's use a variant of the **recursion tree** technique to solve this recurrence
  - Idea is that we progress down a recursion execution tree, adding at each level of the tree
  - Assume that N is 2^K for some K

$T(N) = 4T(N/2) + N ==$

$T(2^K) = 4T(2^{K-1}) + 2^K$ *but* $T(2^{K-1}) = 4T(2^{K-2})+2^{K-1}$

$= 4[4T(2^{K-2}) + 2^{K-1}] + 2^K$

$= 4^2T(2^{K-2})+2^{K+1}+2^K$ *but* $T(2^{K-2}) = 4T(2^{K-3})+2^{K-2}$

$= 4^2[4T(2^{K-3})+2^{K-2}]+2^{K+1}+2^K$

$= 4^3T(2^{K-3})+2^{K+2}+2^{K+1}+2^K$

$= \ldots$ *let's now generalize for level i*

$T(2^K) = 4^i T(2^{K-i}) + \sum_{j=0}^{i-1} 2^{K+j}$ *now let i = K*

$T(2^K) = 4^K T(2^{K-K}) + \sum_{j=0}^{K-1} 2^{K+j}$ *assume T(1) = 1*

we can also factor 2^K out of the sum

$= 4^K + 2^K \sum_{j=0}^{K-1} 2^j$

we can now eval. geometric sum

$= 4^K + 2^K(2^K - 1)$ *but $N=2^K$*

$= 4^K + N(N-1)$

$= 4^K + N^2 - N$ *but $4^K = (2^2)^K = (2^K)^2 = N^2$*

so finally

$T(N) = N^2 + N^2 - N = 2N^2 - N$

➔ **Theta(N^2)**

More Integer Multiplication

- Note that this is the SAME run-time as Gradeschool
  - Further, the overhead for this will likely make it slower overall than Gradeschool
  - So why did we bother?
    - If we think about it in a more "clever" way, we can improve the divide and conquer solution so that it is in fact better than Gradeschool
  - We will do this next lecture