Divide and Conquer: Integer Multiplication

The standard integer multiplication routine of two n-digit numbers involves n multiplications of an n-digit number by a single digit, plus the addition of n numbers, which have at most 2n digits. All in all, assuming that each addition and multiplication between single digits takes O(1), this multiplication takes $O(n^2)$ time:

	quantity	**time**
1) multiplication n-digit by 1-digit	**n**	**O(n)**
2) additions 2n-digit by n-digit max	**n**	**O(n)**

Total time = $n*O(n) + n*O(n) = 2n*O(n) = O(n)*O(n) = O(n^2)$.

(Note: other statements necessary in the integer multiplication of large integers are minor compared to the work detailed above.)

Now, as we have done with several problems in the past, let's consider a divide-conquer solution:

Imagine multiplying an n-bit number by another n-bit number, where n is a perfect power of 2. (This will make the analysis easier.) We can split up each of these numbers into two halves.

Let the first number be I, and the second be J. Let the "left half" of the first number be I_h and the "right half" of the first number be I_l. (h is for high bits, l is for low bits.) Assign J_h and J_l similarly. With this notation, we can set the stage for solving the problem in a divide and conquer fashion.

$I \times J = [(I_h \times 2^{n/2}) + I_l] \times [(J_h \times 2^{n/2}) + J_l]$

$= I_h \times J_h \times 2^n + (I_l \times J_h + I_h \times J_l) \times 2^{n/2} + I_l \times J_l$

Written in this manner we have broken down the problem of the multiplication of 2 n-bit numbers into 4 multiplications of n/2- bit numbers plus 3 additions. (Note that multiplying any binary number by an arbitrary power of two is just a shift operation of the bits.) Thus, we can compute the running time T(n) as follows:

$T(n) = 4T(n/2) + \theta(n)$

This has the solution of $T(n) = \theta(n^2)$ by the Master Theorem.

Now, the question becomes, can we optimize this solution in any way. In particular, is there any way to reduce the number of multiplications done. Some clever guess work will reveal the following:

Let $P_1 = (I_h + I_l) \times (J_h + J_l) = I_h x J_h + I_h x J_l + I_l x J_h + I_l x J_l$
$P_2 = I_h \times J_h$, and
$P_3 = I_l \times J_l$

Now, note that

$P_1 - P_2 - P_3 = I_h x J_h + I_h x J_l + I_l x J_h + I_l x J_l - I_h x J_h - I_l x J_l$
$= I_h x J_l + I_l x J_h$

Then we have the following:

$I \times J = P_2 \times 2^n + [P_1 - P_2 - P_3] \times 2^{n/2} + P_3$.

So, what's the big deal about this anyway?

Now, consider the work necessary in computing P_1, P_2 and P_3. Both P_2 and P_3 are n/2-bit multiplications. But, P_1 is a bit more complicated to compute. We do two n/2 bit additions, (this takes O(n) time), and then one n/2-bit multiplication. (Potentially, n/2+1 bits…)

After that, we do two subtractions, and another two additions, each of which still takes O(n) time. Thus, our running time T(n) obeys the following recurrence relation:

$T(n) = 3T(n/2) + \theta(n)$.

The solution to this recurrence is $T(n) = \theta(n\text{^}(\log_2 3))$, which is approximately $T(n) = \theta(n^{1.585})$, a solid improvement.

Although this seems it would be slower initially because of some extra precomputing before doing the multiplications, for very large integers, this will save time.

Q: Why won't this save time for small multiplications?
A: The hidden constant in the θ(n) in the second recurrence is much larger. It consists of 6 additions/subtractions whereas the θ(n) in the first recurrence consists of 3 additions/subtractions.

Note: Incidentally, I decided to do this problem on my own instead of looking at the book solution. As it turns out, I solved it slightly differently than the book. Hopefully this illustrates that even if "the book" has a particular solution to a problem, that doesn't mean another equally plausible and efficient solution does not exist. Furthermore, many problems have multiple solutions of competing efficiency, so often times, there isn't a single right answer. (FYI, there were two fundamentally different algorithms that got close to full-credit on the pearl/box problem.)

Example to Illustrate Algorithm

Mutliply 11010011 x 01011001.

To simplify matters, I will do the work in decimal, and just show you the binary outputs:

Let I = 11010011, which is 211 in decimal
Let J = 01011001, which is 89 in decimal.
Then we have I_h = 1101, which is 13 in decimal, and
I_l = 0011, which is 3 in decimal
Also we have J_h = 0101, which is 5 in decimal, and
J_l = 1001, which is 9 in decimal

1) Compute $I_h + I_l$ = 10000, which is 16 in decimal
2) Compute $J_h + J_l$ = 1110, which is 14 in decimal
3) Recursively multiply $(I_h + I_l) \times (J_h + J_l)$, giving us 11100000, which is 224 in decimal. (This is P_1.)
4) Recursively mutliply $I_h \times J_h$, giving us 01000001, which is 65 in decimal. (This is P_2.)
5) Recursively multiply $I_l \times J_l$, giving us 00011011, which is 27 in decimal. (This is P_3.)
6) Compute $P_1 - P_2 – P_3$ using 2 subtractions to yield 10000100, which is 132 in decimal
7) Now compute the product as 01000001x100000000 +
10000100x 00010000 +
00011011 =

	0100000100000000	**$(P_2 x2^8)$**
	100001000000	**$((P_1- P_2- P_3) x2^4)$**
+	**00011011**	**(P_3)**

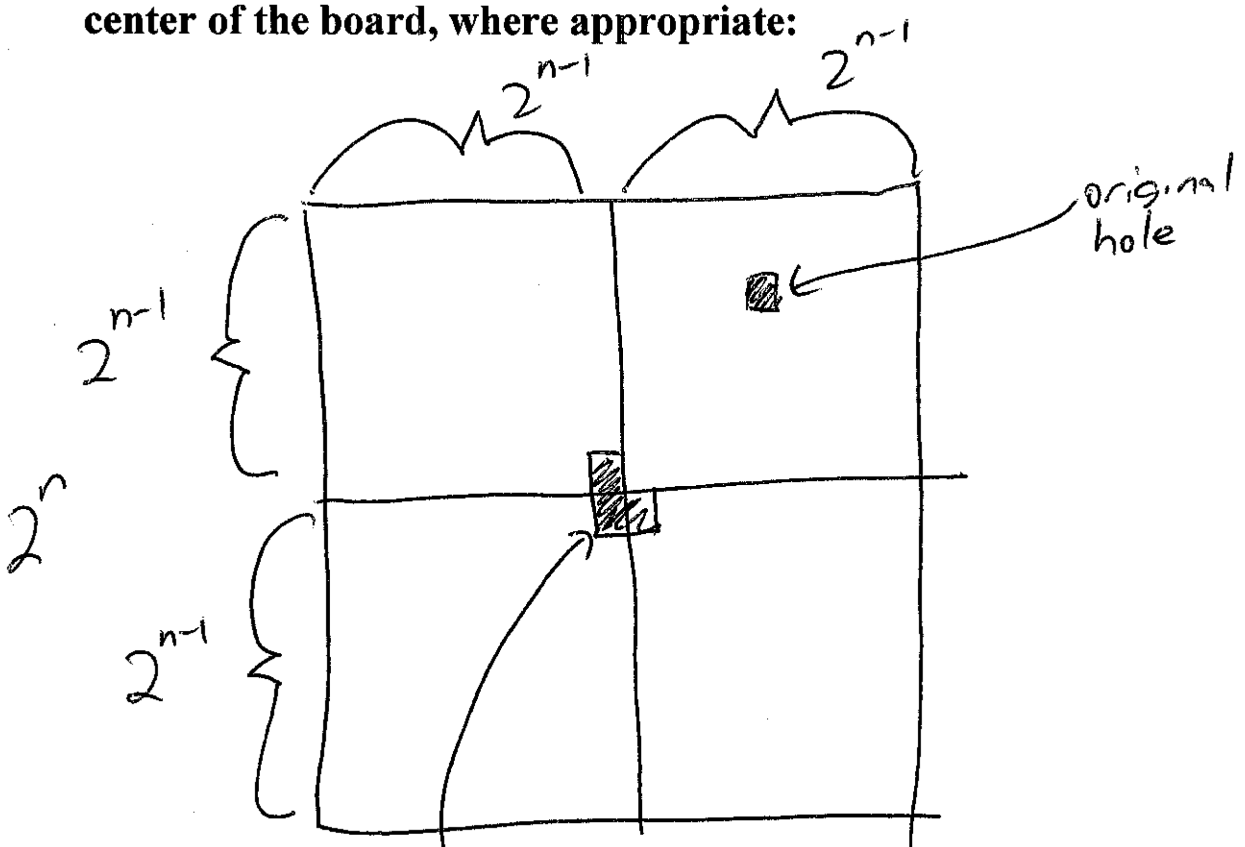
0100100101011011, which is 18779 in decimal, the correct answer. (This is also $65x2^8+132\ x2^4+27$.)

Tromino Tiling

A tromino is a figure composed of three 1x1 squares in the shape of an L. Given a 2^nx2^n checkerboard with 1 missing square, we can recursively tile that square with trominoes.

Here's how we do it:

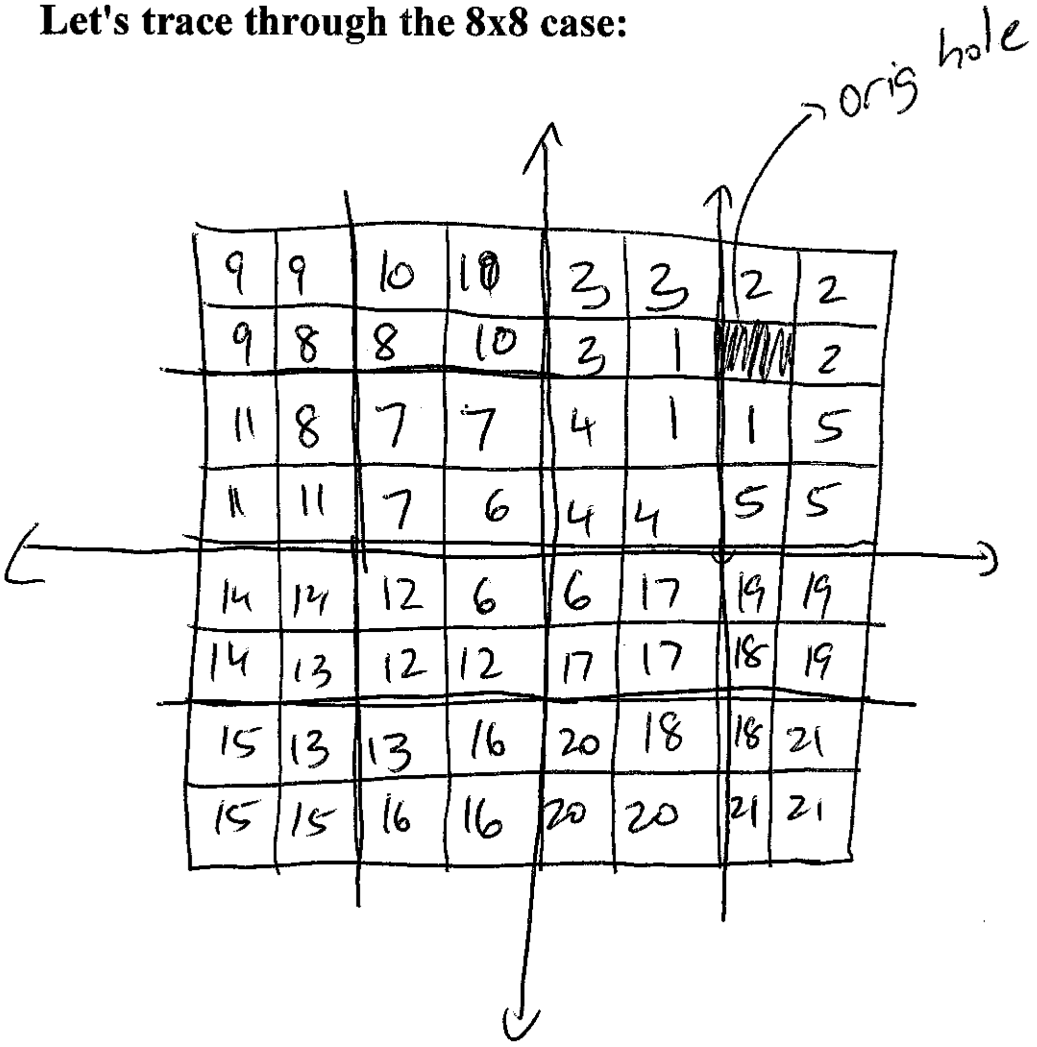
1) Split the board into four equal sized squares.
2) The missing square is in one of these four squares. Recursively tile this square since it is a proper recursive case.
3) Although the three other squares aren't missing squares, we can "create" these recursive cases by tiling one tronimo in the center of the board, where appropriate:



① Add this tromino

② Recursively tile all 4 quadrants, each has "1 missing square"

Let's trace through the 8x8 case:

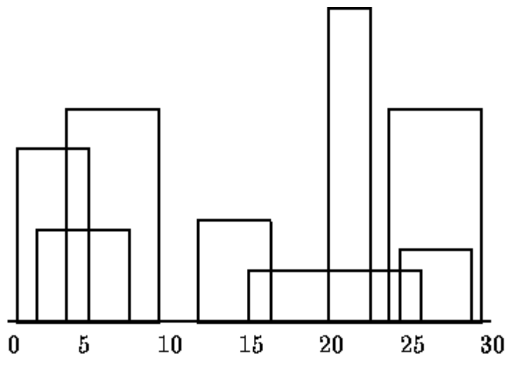


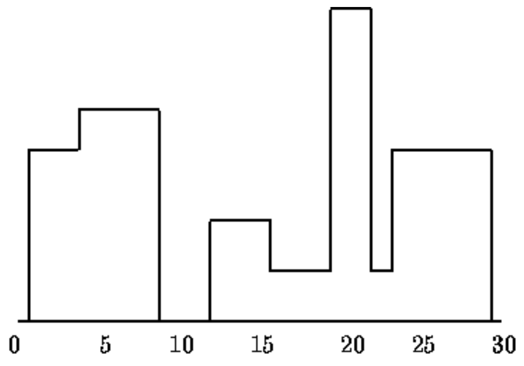
Now, let's do the analysis. Let T(n) be the running time of tiling a nxn square, where n is a perfect power of 2. Then we form the following recurrence relation:

$T(n) = 4T(n/2) + O(1)$, since the extra work involves putting a tile in the middle. Using the master theorem, we have $A = 4$, $B= 2$, $k=0$, and $B^k = 1 < A$. Thus, the running time is $O(n^2)$. This makes sense since we have n^2 to tile and tile at least once each recursive call.

Skyline problem

You are to design a program to assist an architect in drawing the skyline of a city given the locations of the buildings in the city. To make the problem tractable, all buildings are rectangular in shape and they share a common bottom (the city they are built in is very flat). The city is also viewed as two-dimensional. A building is specified by an ordered triple (L_i, H_i, R_i) where L_i and R_i are left and right coordinates, respectively, of building *i* and H_i is the height of the building. In the diagram below buildings are shown on the left with triples (1,11,5), (2,6,7), (3,13,9), (12,7,16), (14,3,25), (19,18,22), (23,13,29), (24,4,28) the skyline, shown on the right, is represented by the sequence: (1, 11, 3, 13, 9, 0, 12, 7, 16, 3, 19, 18, 22, 3, 23, 13, 29, 0)





You need to Merge two skylines——similar to the merge sort

For instance: there are two skylines,

Skyline A: $a_1, h_{11}, a_2, h_{12}, a_3, h_{13}, \ldots, a_n, 0$
Skyline B: $b_1, h_{21}, b_2, h_{22}, b_3, h_{23}, \ldots, b_m, 0$

merge (list of a’s, list of b’s) form into (c_1, h_{11}, c_2, h_{21}, c_3, …, c_{n+m}, 0)

Clearly, we merge the list of a's and b's just like in the standard Merge algorithm. But, it addition to that, we have to properly decide on the correct height in between each set of these boundary values. We can keep two variables, one to store the current height in the first set of buildings and the other to keep the current height in the second set of buildings. Basically we simply pick the greater of the two to put in the gap.

After we are done, (or while we are processing), we have to eliminate redundant "gaps", such as 8, 15, 9, 15, 12, where there is the same height between the x-coordinates 8 and 9 as there is between the x-coordinates 9 and 12. (Similarly, we will eliminate or never form gaps such as 8, 15, 8, where the x-coordinate doesn't change.)

Since merging two skylines of size n/2 should take O(n), letting T(n) be the running time of the skyline problem for n buildings, we find that T(n) satisfies the following recurrence:

$T(n) = 2T(n/2) + O(n)$

Thus, just like Merge Sort, for the Skyline problem T(n) = O(nlgn).