Multiplication in threes

Assume that \( x \) and \( y \) are \( n \) digit numbers, and, assume for convenience, that \( n \) is divisible by 3. Then we can write \( x = x_310^{2n/3} + x_110^{n/3} + x_0 \), and \( y = y_210^{2n/3} + y_110^{n/3} + y_0 \), where each \( x_i, y_i \) is a block of consecutive digits in \( x \) or \( y \). Then multiplying these expressions gives:

\[
x y = x_3y_210^{4n/3} + (x_2y_1 + x_1y_2)10^{n/3} + (x_2y_0 + x_1y_1 + x_0y_2)10^{2n/3} + (x_1y_0 + y_0x_1)10^{n/3} + x_0y_0.
\]

Let the co-efficient of \( 10^{n/3} \) be called \( c_1 \), i.e., \( c_0 = x_0y_0, c_1 = x_1y_0 + x_0y_1 \), etc. If we do this recursion in the obvious way, we have to do 9 recursive calls to multiply numbers with \( n/3 \) digits, for a recurrence relation:

\[
T(n) = 9T(n/3) + cn \text{ for some constant } c.
\]

In the general format for divide-and-conquer recurrences, we have \( A = 9, B = 3, C = 1 \), so since \( A/B^C > 1 \), the time is \( O(n^{\log_A B}) = O(n^{\log_9 3}) = O(n^2) \), which is no improvement over the grade-school method or obvious divide-and-conquer in blocks of 2.

However, we can compute the terms in a clever way like we did in pairs:

\( a = x_0y_0, b = x_1y_1, c = x_2y_2, d = (x_0 + x_1)(y_0 + y_1), e = (x_0 + x_2)(y_0 + y_2) \) and \( f = (x_2 + x_0)(y_2 + y_0) \). Then \( c_0 = a, c_1 = d - a - b, \)
\( c_2 = e - a - c + b, c_3 = f - a - c, \) and \( c_4 = c \). So we get a recurrence of approximately the form:

\[
T(n) = 6T(n/3) + O(n) \text{ (which gives us the time } T(n) = O(n^{\log_9 3}) = O(n^{1.63..}) \text{, which is worse than when we did the clever multiplication in pairs. So, as almost everyone concluded, it is better to do divide-and-conquer multiplication in pairs than to do it this way.)}
\]

But we can be REALLY tricky. Consider the polynomials \( p_x(z) = x_2z^2 + x_1z + x_0 \) and \( p_y(z) = y_2z^2 + y_1z + y_0 \). Then \( p_x(z)p_y(z) = e_4z^4 + e_3z^3 + e_2z^2 + e_1z + e_0 \) is a polynomial of degree 4. We can determine a polynomial of degree 4 by its values at 5 points: pick for simplicity \( z = -2, -1, 0, 1, 2 \). This suggests doing the following computations:

\[
a = (4x_2 - 2x_1 + x_0)(4y_2 - 2y_1 + y_0) = 16c_4 - 8c_3 + 4c_2 - 2c_1 + c_0, b = (x_2 - x_1)(y_2 - y_1 + y_0) = c_4 - c_3 + c_2 - c_1 + c_0 = x_0y_0, d = (x_2 + x_1)(y_2 + y_1 + y_0) = c_4 + c_3 + c_2 + c_1 + c_0, \]
\( e = (4x_2 + 2x_1 + x_0)(4y_2 + 2y_1 + y_0) = 16c_4 + 8c_3 + 4c_2 + 2c_1 + c_0 \). Then we get back \( c_4...c_0 \) as follows:

1. \( c_0 = c \)
2. $f = (d - b)/2 = e_1 + e + 3,$
3. $g = (e - a)/2 = 8e_3 + e_1$
4. $e_3 = (g - f)/7,$
5. $e_1 = f - e_3,$
6. $h = (b + d)/2 - e_0 = e_4 + e_2$
7. $i = (a + e - 2e_0)/8 = 4e_4 + e_2,$
8. $e_4 = (i - h)/3,$
9. $e_2 = h - e_4.$

Then we get a recursion: $T(n) = 5T(n/3) + O(n)$, which comes out to $O(n^{\log_3 5}) = O(n^{1.51})$ which is better than in twos (for REALLY big $n$.)

This method generalizes, to get theoretically better and better algorithms as the number of pieces increases. The limit of this seems to be the Fast Fourier Multiplication Algorithm, which multiplies in time $O(n\log^2 n)$.

The secret of the Fast Fourier is to pick the points at which to evaluate the polynomial carefully, to allow much of the work for different points to be identical.

**Hamiltonian path** Consider the following algorithm for deciding whether a graph has a Hamiltonian Path from $x$ to $y$, i.e., a simple path in the graph from $x$ to $y$ going through all the nodes in $G$ exactly once. ($N(x)$ is the set of neighbors of $x$, i.e. nodes directly connected to $x$ in $G$).

1. `HamPath(G, x : node, y : node)`
2. If $x = y$ is the only node in $G$ return True.
3. If no node in $G$ is connected to $x$, return false.
4. For each $z \in N(x)$ do:
5. If `HamPath(G - {x}, z, y)`, return true.
6. Return false

a. Explain (informally) why this algorithm is correct. (5 points) b. If every node of the graph $G$ has degree (number of neighbors) at most 3, how long will this algorithm take at most? (15 points) (Hint: you can get a tighter bound than the most obvious one.)

We want to find a Hamiltonian path from $x$ to $y$. The back-tracking algorithm branches on the first node visited after $x$. The choices are all the neighbors of $x$ in the graph. We can’t return to $x$ because the path must be simple. If we can find a Hamiltonian path from some neighbor $z$ of $x$ to $y$ in $G - \{x\}$, we can append $x$ to the front of the path and get a
Hamiltonian path from $x$ to $y$. On the other hand, if we cannot for any
neighbor $z$ of $x$, then we cannot find a Hamiltonian path from $x$ to $y$, since
the second node along such a path must be such a $z$.

The time of the algorithm is proportional to the number of recursive calls
that get made, since each time through the loop makes one recursive call
and then does $O(1)$ work. The recursion tree for the program has depth $n$,
since each time step we delete exactly one node from $G$. In general graphs,
the time could be as much as $(n-1)!$, since the first node could have $n-1$
neighbors, each possibility for the second node $n-2$ neighbors, and so on.
However, if no node in $G$ has more than 3 neighbors, an obvious upper
bound for the total time is $O(3^n)$, since no node in the recursion tree has
more than 3 children, and the depth is $n$.

In fact, since after the first step, we only call $y$ right after one of its
neighbors $x$ has been deleted from the tree, we can see that the total size
of the tree is at most $3 + 2^{n-1} = O(2^n)$.

There is an even more subtle argument that improves this bound: A graph
with $n$ nodes none of which has more than 3 edges can have at most $3n/2$
edges total, since each edge touches two nodes. Every time we branch
in the recursion, i.e., $x$ has more than one neighbor, we delete at least 2
edges from the graph. Thus, any path in the recursion tree can have at
most $(3n/2)/2 = 3n/4$ places where there was any branching. Thus, the
time is actually bounded by $O(2^{0.75n})$ for such graphs.

**Implementation: 20 pts** Implement a back-tracking algorithm for maximum
independent set (such as from class). Run your algorithm on random
graphs with edge probability 1/2 (as in the greedy algorithms assignment)
for $n$ as many different powers of 2 as you can (without using more than
an one hour computer time on any one instance). How does the actual
maximum independent set size compare to the size found by the greedy
heuristic last assignment?

While the exact optimal threshold depends on implementation details
(language used, operating system), a general observation is that the best
threshold for the hybrid is well smaller than the place where the two al-
gorithms have equal performance. With a divide-and-conquer algorithm,
improving performance by a fixed factor at the threshold size will improve
it proportionally for larger sizes. So we want to find a threshold value
where the naive algorithm time is the smallest fraction of the d&c time.