1 Parenthesis Matching

We first look at the parenthesis matching problem, which is defined as follows: You are given a sequence of characters such that each character is either ( or ). You want to return true if the sequence is well formed and false if the sequence is not well-formed. For instance ⟨(,(,),(,))⟩ is a well formed sequence, while ⟨),(,),(,)⟩ is not.

Question: There is a simple solution that does not use divide and conquer. Can you think of it?
Answer: You can just go through and keep a counter. When you see an open parenthesis, increment the counter. When you see a closed parenthesis, decrement it. If the counter ever goes negative, return false. If the counter is not 0 at the end of the sequence, return false. If the counter is 0 at the end of the sequence and was never negative, return true. The running time is O(n), the best we can hope for.

But just for kicks, let’s try to design a divide and conquer algorithm.

Question: Let’s try the simplest: Divide the sequence into two equal halves. What should the recursive calls return for us to be able to merge? The first thing that comes to mind might be that the function returns whether the given sequence is well-formed. Is this enough?
Answer: Clearly, if both $s_1$ and $s_2$ are well-formed expressions, $s_1$ concatenated with $s_2$ must be a well-formed expression. The problem is that we could have $s_1$ and $s_2$ such that neither of which is well-formed but $s_1s_2$ is well-formed (e.g., “((” and “))”). This is not enough information to conclude whether $s_1s_2$ is well-formed.

We need more information from the recursive calls. We’ll crucially rely on the following observations (which can be formally shown by induction):

Observation 1 If $s$ contains “)” as a substring, then $s$ is a well-formed parenthesis expression if and only if $s'$ derived by removing this pair of parenthesis “)” from $s$ is a well-formed expression.

Applying this reduction repeatedly, we can show that a parenthesis sequence is well-formed if and only if it eventually reduces to an empty string.

Observation 2 If $s$ does not contain “)” as a substring, then $s$ has the form “)ˢ⁽⁽”. That is, it is a sequence of close parens followed by a sequence of open parens.
That is to say, on a given sequence $s$, we’ll keep simplifying $s$ conceptually until it contains no substring “()” and return the pair $(i, j)$ as our result. This is relatively easy to do recursively. Consider that if $s = s_1s_2$, after repeatedly getting rid of “()” in $s_1$ and separately in $s_2$, we’ll have that $s_1$ reduces to “$i$” and $s_2$ reduces to “$k$” for some $i, j, k, \ell$. To completely simplify $s$, we merge the results. That is, we merge “$i$” with “$k$”. The rules are simple:

- If $j \leq k$ (i.e., more close parens than open parens), we’ll get “$i+k-j$”.
- Otherwise $j > k$ (i.e., more open parens than close parens), we’ll get “$i+\ell-j$”.

This directly leads to a divide and conquer algorithm.

ParenMatch($S, n$)

1. if $n = 1$ then return (0, 1) if $S[1] = (\) and (1, 0) otherwise.
2. $(i, j) \leftarrow$ ParenMatch($S_{left}, n/2$)
3. $(k, \ell) \leftarrow$ ParenMatch($S_{right}, n/2$)
4. if $j \leq k$ then return $(i + k - j, \ell)$
5. else return $(i, \ell + j - k)$

What is the running time of this algorithm?

$$W(n) = 2W(n/2) + O(1) = O(n)$$
$$S(n) = S(n/2) + O(1) = O(lg n)$$

2 Grade-school addition

In grade school, you are taught to execute several algorithms by hand to perform basic arithmetic. Although in this course we are primarily interested in a model of algorithms that roughly captures programs executing on a RAM computer, in which all arithmetic takes a single “step,” we can also apply our thinking to computing “by hand.” In this case, the natural model of running time is that a single-digit operation – reading, writing, or a single-digit arithmetic operation – takes some constant (bounded) time.

**Question 1.** Consider the algorithm you were taught for adding integers in grade school. What is the asymptotic running time of this addition algorithm in this digit-based model (as a function of the larger number of digits $n$ of the two inputs)?

**Answer:** $O(n)$
Question 2. Could there be an algorithm with a strictly asymptotically faster running time in terms of the number of input digits? How do you know?
Answer: No. Running strictly faster than \( O(n) \) means that for every possible constant \( c > 0 \) that could be under the big-O, for sufficiently large \( n \), eventually the running time is less than \( cn \). (We denote such a running time by \( o(n) \), i.e., “little-oh.”) But then, for \( c = 1/2 \), eventually the algorithm must read at most half of the digits. Then there are at least two different inputs that differ in one (unread) digit, for which the algorithm must return the same answer. But if the two inputs differ in exactly one digit, the answers must be different.

3 Faster multiplication

Now, let’s turn to the problem of multiplication. Again, all of us were trained to execute an algorithm for multiplying two integers together.

Question 3. What is the asymptotic running time of the algorithm for integer multiplication you were taught in grade school?
Answer: Certainly at least \( n^2 \). There are some subtleties in how carries are handled. Note that the sum of \( n \) one-digit numbers is at most \( O(\log n) \) digits, so at worst we are solving \( n \) addition problems with numbers of at most \( O(\log n) \) digits for each column, so the time is no more than \( O(n^2 \log^2 n) \). The key point here is that it is surely much greater than \( O(n) \), so our optimality argument for addition does not carry over to multiplication.

Now, we’ll develop a digit-based multiplication algorithm that is significantly asymptotically faster than the one you were taught in grade school. Let’s focus on binary arithmetic to keep it simple (but we could do this for decimal as well).

Question 4. At a high level, how would you currently try to structure such an algorithm?
Answer: Right now, divide-and-conquer is the only major technique we have got in the toolbox, so let’s use that. (Note that this is not the only possible answer, nor even the best answer. But it is the best we have available at this point.)

Question 5. Now, in more detail, how would you apply this structure to develop a skeleton of an algorithm for the problem of multiplication?
Hint: In divide-and-conquer, we need to choose subproblems that are “smaller” instances of the same problem. In this case, this means smaller integer multiplication problems.
Answer: Let’s try to divide the numbers in half. That is, we write the first number as \( a \cdot 2^{n/2} + b \) and the second number as \( c \cdot 2^{n/2} + d \) where \( a, b, c, \) and \( d \) are \( n/2 \)-digit numbers.
Question 6. Now, fill out the algorithm so that you get a correct algorithm for multiplication.

Answer: Our multiplication problem is now to compute:

\[(a \cdot 2^{n/2} + b) \cdot (c \cdot 2^{n/2} + d) = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd\]

So, if we solve the four subproblems \(a \cdot c\), \(a \cdot d\), \(b \cdot c\), and \(b \cdot d\), we only need to add four numbers together; note that multiplying by \(2^k\) just involves writing the number in the appropriate location, “shifted” \(k\) digits up. (We already use this fact in the grade-school multiplication algorithm.)

Thus, we need only solve three addition problems (say, add \(ad + bc\), \(ac \cdot 2^n + bd\), and then shift the result of \(ad + bc\) \(2^{n/2}\) digits and add it to \(ac \cdot 2^n + bd\)).

Question 7. What is the running time of this algorithm?

Answer: we can see that the combine step takes time \(O(n)\), and we are solving four subproblems of size (at most) \(n/2\) each. So our running time is given by \(T(n) = 4T(n/2) + O(n)\). Drawing the tree, etc., we can see that this is dominated by the bottom level wherein we have \(4 \log_2 n = n^2\) nodes of \(O(1)\) work each, which is at least \(n^2\) steps. So, there is not much improvement here.

Question 8. How can we improve this algorithm to obtain a significantly faster algorithm?

Answer: Let’s try to mimic the trick Strassen used: let’s try to re-use the sub-problems, so that the tree of subproblems branches less. If we can get down to three subproblems, then this will give a running time of \(T(n) = 3T(n/2) + O(n) = 3^{\log_2 n} = n^{\log_3 3} \approx n^{1.585}\).

Strassen was very clever to find his seven subproblems, but we can crack integer multiplication with just a little effort. Notice that if you simply multiply \((a + b) \cdot (c + d)\), which is essentially the product of two \(n/2 + 1\) digit numbers, you get \(ac + ad + bc + bd = ac + bd + (ad + bc)\). So we could obtain \(ad + bc\) in just one multiplication rather than two, if only we could get rid of the \(ac\) and \(bd\) terms. But happily, we wished to compute these terms anyway. So, if our three subproblems are to compute \((a + b) \cdot (c + d)\), \(ac\) and \(bd\), then we can compute \((ad + bc)\) by \((a + b) \cdot (c + d) - ac - bd\), and then continue as before. So this is surely correct. This uses extra addition steps, but since that merely increases the constant under the big-O, we see that this is still an asymptotically faster multiplication algorithm, as promised. This algorithm was originally proposed by Karatsuba in the 1960s.