The MERGE_SORT algorithm is a classic example of the power of the divide and conquer technique. This technique is a general approach for solving problems with a three-step approach:

- **Divide:** the original problem into smaller subproblems,
- **Conquer:** each smaller subproblem separately, and
- **Combine:** the results back together.

With MERGE_SORT, we divided the initial array in two, conquered the sorting problem on the two smaller arrays with recursive calls to MERGE_SORT, and combined the results using MERGE. In the next few lectures, we will see other problems where this approach is applicable.

Note. Before discussing divide-and-conquer algorithms in class, we reviewed the Master theorem. See the notes for Lecture 3 for all the details on this topic.

1. **Counting inversions**

Here are two sequences of numbers:

- 1 2 5 4 6 7 9
- 9 1 5 2 7 6 4

Which one is closer to sorted? It seems obvious that the first sequence is, but how can we measure the “unsortedness” of a sequence to make this notion precise? One way to do this is by counting inversions.

**Definition 4.1** (Inversion). The pair of indices \((i, j) \in \{1, 2, \ldots, n\}\) with \(i < j\) forms an inversion in the sequence \(a_1, a_2, \ldots, a_n\) if \(a_i > a_j\). The number of inversions of the sequence \(a_1, \ldots, a_n\) is

\[
K(a) = |\{i, j \in [n] : i < j \text{ and } a_i > a_j\}|,
\]

the total number of pairs of indices that form an inversion in the sequence.

In the counting inversions problem, we are given a sequence \(a_1, \ldots, a_n\) of \(n\) integers and our task is to determine the number \(K(a)\) of inversions in the sequence.

The brute force algorithm for counting inversions checks all \(\binom{n}{2}\) pairs of indices \(i < j\) to see if they form an inversion. This algorithm has time complexity \(\Theta(n^2)\). Can we do better? With divide-and-conquer algorithms, yes!

The first step in applying the divide-and-conquer algorithm is to figure out how we can divide the problem into smaller subproblems. In this case, there’s a very natural option: let’s simply divide the sequence \(a_1, \ldots, a_n\) into the smaller subsequences \(L = (a_1, \ldots, a_{\frac{n}{2}})\) and \(R = (a_{\frac{n}{2}+1}, \ldots, a_n)\).
Can we conquer the smaller subproblems obtained with the division approach we identified above? Certainly! We can count the numbers $K(L)$ and $K(R)$ of inversions within $L$ and within $R$ with recursive calls to our algorithm.

So now we are left with the final task: to combine the solutions to the smaller subproblems into a solution for the original problem. And in this case it is not entirely trivial: the values $K(L)$ and $K(R)$ computed in the conquer step do not account for all of the inversions because there can also be inversions in $a$ where $i \leq \frac{n}{2}$ and $j > \frac{n}{2}$ that are not in either $L$ or $R$. So we have that

$$K(a) = K(L) + K(R) + r$$

with $r$ denoting the number of inversion pairs that include one element from $L$ and one from $R$.

So how can we compute $r$? Well, for each element $a_j \in R$, we need to determine how many elements in $L$ are smaller than $a_j$. Call this number $r_j$. Then $r = \sum_{j=n/2+1}^{n} r_j$. So now our task is to compute $r_j$ efficiently, and at this point we might notice that there’s a very close connection to the MergeSort algorithm: if we had sorted $L$ and $R$ and merge the sorted list, $r_j$ is exactly the number of elements left in $L$ at the moment when we moved $a_j$ to the merged list.

This observation suggests that we can compute $r$ efficiently with a simple modification to our standard MergeSort algorithm.

**Algorithm 1: SortAndCountInversions(A)**

```pseudo
if n = 1 return;
(L, r_L) ← SortAndCountInversions(A[1, ..., |n/2]|);
(R, r_R) ← SortAndCountInversions(A[|n/2| + 1, ..., n]);
r ← 0, S ← ∅;
while L, R are not empty do
    if the minimum element left in R is smaller than minimum left in L then
        Move the minimum element of R to the merged list S;
        $r_L = |L|;$
    else
        Move the minimum element of L to the merged list S;
return (S, r_L + r_R + r);
```

As an exercise, you should verify that the merge step of this algorithm can be turned into more precise pseudocode that runs in time $O(n)$. When this is done, we are left with an algorithm that has time complexity $T$ satisfying the recurrence relation

$$T(n) = 2T(\frac{n}{2}) + O(n),$$
which as we already have seen with the analysis of MergeSort (or as we can verify again with the Master theorem) satisfies \( T(n) = O(n \log n) \).

2. INTEGER MULTIPLICATION

Let’s consider one of the most fundamental arithmetic problems around: how to multiply two positive integers.

**Definition 4.2** (Integer multiplication problem). Given two \( n \)-bit positive integers \( x \) and \( y \), output their product \( xy \).

Humans invented an algorithm for this problem thousands of years ago. It is now known as the grade school algorithm. With this algorithm, we multiply \( x \) by \( y \), and shift the result \( n - i \) positions to the left. As an example, when we run this algorithm to solve \( 13 \times 11 \), we obtain

\[
\begin{array}{c}
1101 \\
\times 1011 \\
1101 \\
1101 \\
0000 \\
+ 1101 \\
\hline
10001111 (= 143)
\end{array}
\]

Notice that in the computations, we produce \( O(n^2) \) intermediate bits, and this is indeed the time complexity of this algorithm. This begs the question: can we use the Divide & Conquer approach to obtain a more efficient algorithm?

To apply the Divide & Conquer approach, we need to first determine how we can break up the original multiplication problem into problems on smaller inputs. The natural way to do this is to split the \( n \)-bit integer \( x \) into two \( n/2 \)-bit integers \( x_L \) (containing the \( n/2 \) most-significant bits) and \( x_R \) (containing the least-significant bits) and to do the same with \( y \). Then

\[
x = 2^{n/2}x_L + x_R,
\]

\[
y = 2^{n/2}y_L + y_R,
\]

and

\[
xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.
\]

This is exactly what we were hoping to see! We have divided up the problem of multiplying two \( n \)-bit integers into four instances of the smaller problem of multiplying two \( n/2 \)-bit integers. The resulting algorithm is as follows.

At first, it might be worrisome to see that we have also added extra multiplications by \( 2^n \) and \( 2^{n/2} \), but in fact these are just shifts (by \( n \) and \( n/2 \) bits, respectively), so these operations have time complexity \( O(n) \) and the total time complexity of the Multiply algorithm is defined by the recursion

\[
T(n) = 4T(n/2) + O(n).
\]
Algorithm 2: Multiply \((x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_n)\)

```plaintext
if n = 1 return xy;
(x_L, x_R) ← Split (x);
(y_L, y_R) ← Split (y);
P_{LL} ← Multiply (x_L, y_L);
P_{LR} ← Multiply (x_L, y_R);
P_{RL} ← Multiply (x_R, y_L);
P_{RR} ← Multiply (x_R, y_R);
return 2^n P_{LL} + 2^{n/2}(P_{LR} + P_{RL}) + P_{RR};
```

We can apply the Master Theorem with parameters \(a = 4, b = 2\), and \(c = 1\) and the observation that \(1 = c < \log_b a = 2\) to obtain
\[
T(n) = O(n^2).
\]

This is exactly the same as the grade school algorithm! So while the Divide & Conquer approach gives an elegant algorithm, it does not appear to have led to any efficiency improvement.

3. Fast integer multiplication

... And yet if that was the end of the story, we probably wouldn’t be covering the integer multiplication problem at this point in the class! Recall that
\[
xy = 2^n x_L y_L + 2^{n/2} (x_L y_R + x_R y_L) + x_R y_R.
\]

There’s a deceptively simple observation that can be traced back to Gauss which will be immensely useful to us: the middle term \(x_L y_R + x_R y_L\) satisfies the identity
\[
x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.
\]

We can use this identity to express the term \(x_L y_R + x_R y_L\) with a difference of three multiplications. This would be a step back for us (we only need 2 multiplications to compute the same term directly!) except that two of the multiplications, \(x_L y_L\) and \(x_R y_R\) are multiplications that we already need to do in the multiplication algorithm anyways! As a result, we can now implement a Divide & Conquer multiplication algorithm that performs only 3 multiplications instead of 4.

Algorithm 3: FastMultiply \((x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_n)\)

```plaintext
if n = 1 return xy;
(x_L, x_R) ← Split (x);
(y_L, y_R) ← Split (y);
P_{LL} ← FastMultiply (x_L, y_L);
P_{RR} ← FastMultiply (x_R, y_R);
P_{sum} ← FastMultiply (x_L + x_R, y_L + y_R);
return 2^n P_{LL} + 2^{n/2}(P_{sum} - P_{LL} - P_{RR}) + P_{RR};
```
The time complexity of this algorithm now satisfies
\[ T(n) = 3T(n/2) + O(n). \]
We can again apply the Master Theorem, this time with parameters \( a = 3, \ b = 2, \) and \( c = 1. \) Since \( 1 = c < \log_{b} a = \log_{2} 3 < 1.59, \) we obtain
\[ T(n) = O(n^{1.59}). \]
This result is a great illustration of the power of algorithmic thinking: despite countless mathematicians (and students of all ages) using multiplication algorithms over thousands of years, it was only in 1960 that Karatsuba showed with the above algorithm that it was possible to solve the integer multiplication in subquadratic time.