Introduction to Divide and Conquer

Sorting with $O(n \log n)$ comparisons and integer multiplication faster than $O(n^2)$

Periklis A. Papakonstantinou

York University

Consider a problem that admits a straightforward polynomial-time algorithm. Divide and Conquer algorithms are used to reduce the computational time to a smaller polynomial. We begin by reviewing the well-known sorting algorithm MergeSort. Then, we give a simple *Divide and Conquer* algorithm for integer multiplication.

1 Sorting with $O(n \log n)$ comparisons

The input consists of a sequence of n numbers $\langle a_1, \dots, a_n \rangle$ and the output is the elements of the input in non-decreasing order.

Is it possible to sort n integers without comparing every two of them? The basic, albeit misleading intuition suggests that this should be false; i.e. for sorting we must do $\Theta(n^2)$ comparisons. Divide and Conquer (DnC) algorithms for sorting oppose to this intuition, since they manage to sort n elements by comparing much fewer pairs of elements. MergeSort is such an algorithm. At a high level it works as follows: suppose that we are given a sequence of n integers. Now assume that we break this sequence in two halves of equal (or almost equal) length. If we have solved the problem *independently* on each of these two halves then we can combine them into a sorted sequence in linear time. This concept of assuming that you have solutions to independent subproblems and by *combining* you can build solutions for bigger problems is the key point behind every Divide and Conquer algorithm.

1.1 The algorithm

You probably have already come across MergeSort several times during your studies. Here is a space inefficient way to describe this algorithm.

```
MergeSort[$A$]
1 if $length(A) = 1$
2     then
3           Return $A$
4 $q \leftarrow \lfloor \frac{length(A)}{2} \rfloor$
5 $A_1 \leftarrow$ MergeSort[$A[1..q]$]
6 $A_2 \leftarrow$ MergeSort[$A[q+1..]$]
7 Return Merge[$A_1, A_2$]
```

Notation: $A[i..j]$, where $i \leq j$ denotes the subarray of A from the i-th to the j-th element. Similarly, $A[i..]$ denotes the subarray of A which starts from the i-th element and goes to the end of A.

The procedure Merge$[A_1, A_2]$ is not defined. Let us describe it here at an abstract level. The procedure takes as parameters two arrays A_1 and A_2 which are sorted and returns one array that contains the elements of A_1, A_2 (including multiplicity, i.e. if an element appears more than once it will appear the same number of times in the returned value) sorted. The procedure maintains two pointers each moving once from left to right. As the pointers move to the right end we create the output in linear time. For most of this section we will assume that Merge is correct for the (sub)problem where given two ordered arrays we return the elements that appear in both sorted.

1.2 Running time

Let n be the number of elements in the input. Let $T(n)$ denote the number of comparisons of MergeSort on an input of n elements. We have that MergeSort makes two calls to itself for a problem of size almost $n/2$; to be more precise $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$. Also, the *combine step* takes $O(n)$ comparisons. Also, when the input is of length 1 then the answer is returned without comparisons. That is, $T(n) \leq 2T(n/2) + cn$, for some constant c and $n > 1$ and with boundary condition $T(1) = 0$. Using the Master Theorem or any other method for solving recurrence relations we have that $T(n) = O(n \log n)$.

Remark 1. It is important to understand *how* we avoided to compare every pair of elements. The main idea is in the combine step. Assuming that we have solved the subproblems we can get along by comparing only $O(k)$ elements instead of $O(k^2)$, where k is the length of the subarrays. So where does the $\log n$ factor come from? This factor does not come from the combine step, but rather of how many times this combine step should be executed. This has to do with the number of subproblems; i.e. how we break the problem into equal sized subproblems.

Exercise 1. Suppose that the integers in the input are represented in binary. Let N be the total number of bits in the input. Explain why the (worst-case) running time of MergeSort is $O(N \log N)$.

1.3 Proof of correctness

Proofs of correctness of DnC algorithms are much easier than proofs of correctness of other types of algorithms. The reason is that the main difficulty of the algorithm is in its description, and more importantly because the description of the algorithm reflects directly to a recursive definition. Correctness is almost transparent in the description of the algorithm. The most complicated part is the correctness of the "combine step". And in our proof we are going to ignore this since at this level of 3101 students have already dealt with correctness of more complicated iterative algorithms than the Merge (e.g. Assignment 1 question 1(b)). So suppose that MERGE is correct for the (sub)problem where given two ordered arrays we return the elements that appear in both sorted.

One thing that changes from the case of the iterative algorithms is that termination is now being shown that by showing that the recursive calls correspond to subproblems of length strictly decreasing in the natural numbers. Therefore, if we represent the computation by a recursive-call tree, where in the root is the initial call, this means that every branch of the tree is bounded, and thus the algorithm terminates.

Here is the straightforward proof of correctness for MERGESORT.

Remark 2. Unlike iterative algorithms when we prove correctness of a recursive algorithm we do induction on the length of the subproblem. Usually the subproblems are not of size one less than the original subproblem. Therefore, we can use strong/complete induction to simplify the presentation of the proof.

The inductive claim is $P(n)$: MERGESORT is correct for the Sorting problem when the input has n elements.

(Basis) $P(1)$ is true by the way the algorithm works.

(Induction Step) Suppose that for every $k < n$, $P(k)$ is true (I.H.). Wts that $P(n)$ is also true. BTWAW we have that since $n > 1 \implies n/2 < n$ by the (I.H.) we have that A_1 and A_2 are sorted. Finally, the correctness of MERGE for the Merge problem implies that the returned value is sorted.

The only thing that we have not proved is the correctness of MERGE. We leave this as an exercise to the reader.

Exercise 2. Show that the following implementation of MERGE merges two sorted arrays $A_1[1..q]$ and $A_2[1..r]$ in a sorted array $A[1..q + r]$.

```
MERGE[$A_1, A_2$]
 1  $i \leftarrow 1$ (index for the first subarray)
 2  $j \leftarrow 1$ (index for the second subarray)
 3  $A[1..length(A_1) + length(A_2)] \leftarrow$ empty auxiliary array
 4  $k \leftarrow 1$ (index for the auxiliary array)

 5  while $i \le length(A_1)$ and $j \le length(A_2)$
          do
 6            if $A_1[i] \le A_2[j]$
                then
 7                    $A[k] \leftarrow A_1[i]$
 8                    $i \leftarrow i + 1$
 9              else
10                    $A[k] \leftarrow A_2[j]$
11                    $j \leftarrow j + 1$
12            $k \leftarrow k + 1$

13  if $i > length(A_1)$ (i.e. we are "done" with $A_1$)
        then
14            for $k \leftarrow k$ to $length(A_2) - length(A_1) + 1$
                  do
15                    $A[k] \leftarrow A_2[j]$
16                    $j \leftarrow j + 1$
17      else
18            for $k \leftarrow k$ to $length(A_1) - length(A_2) + 1$
                  do
19                    $A[k] \leftarrow A_1[i]$
20                    $i \leftarrow i + 1$
21  Return $A$
```

Hint: prove partial correctness by defining a predicate on $i + j$.

2 Fast Integer multiplication

Given two n-bit numbers a and b output ab.

2.1 Elementary school multiplication

People are familiar with a quite efficient (actually $\Theta(n^2)$ worst case running time) algorithm for integer multiplication from the grade-school years. The same algorithm applies when the numeric system is of base 2. Here is an example.

$$\begin{array}{r}
1\,1\,0\,1 \\
\times\,1\,1\,0\,0 \\
\\
0\,0\,0\,0 \\
0\,0\,0\,0 \\
1\;\,1\,0\,1 \\
+\,1\,1\;\,0\,1 \\
\\
1\,0\,0\;\,1\,1\,1\,0\,0
\end{array}$$

Remark 3. As far as it concerns their time complexity, multiplication and addition are two quite different arithmetic operations. The straightforward algorithm for addition has a linear $\Theta(n)$ worst-case running time, where n is the number of bits of each number (i.e. its representational length). The straightforward multiplication algorithm runs in $\Theta(n^2)$. It is also interesting to note that adding two n-bit numbers can result a number of $n+1$ (but no more) bits, where multiplying two n-bit numbers may result a number of $2n$ (but no more) bits.

Say that the bit representation of $x, y \in \mathbb{N}$ is $x = x_n x_{n-1} \dots x_1$ and $x = y_n y_{n-1} \dots y_1$. When multiplying xy with the above algorithm we do the following: $2^0 x_1 y + 2^1 x_2 y + 2^2 x_3 y + \ldots + 2^{n-1} x_n y$. Therefore, the worst-case running time of this algorithm is $\Theta(n^2)$. We shall see that a simple Divide and Conquer algorithm achieves worst-case running time $O(n^{1.59})$.

2.2 A failed attempt

Can we do any better than $\Theta(n^2)$?

For presentational simplicity suppose that the number of bits of each integer is a power of 2. Extending the idea of the above (elementary-school) algorithm we observe that we can view the multiplication as follows. Say that $x = x_1 x_0$, where x_1 is the $n/2$ higher order bits of x and x_0 is the $n/2$ lower order bits of x. Similarly, $y = y_1 y_0$. Therefore, $x = 2^{n/2} x_1 + x_0$ and $y = 2^{n/2} y_1 + y_0$. That is,

$$xy = 2^n x_1 y_1 + 2^{n/2}(x_1 y_0 + x_0 y_1) + x_0 y_0 \tag{2.1}$$

Therefore, to compute xy it suffices to compute $x_1 y_1$, $x_1 y_0$, $x_0 y_1$ and $x_0 y_0$. We managed to *reduce the multiplication of two n-bit numbers to four multiplications of* $n/2$ *bit numbers.* Equation 2.1 directly implies a Divide and Conquer algorithm for integer multiplication.

```
FIRST-ATTEMPT-MULTIPLY[$x, y$]
1  if the length of $x$ (and $y$) is 1 bit
       then
2          return $x \cdot y$


3  $x = x_1 x_0$ (extract $x_1$, $x_0$ from $x$)
4  $y = y_1 y_0$
5  $p_1 \leftarrow$ FIRST-ATTEMPT-MULTIPLY[$x_1, y_1$]
6  $p_2 \leftarrow$ FIRST-ATTEMPT-MULTIPLY[$x_1, y_0$]
7  $p_3 \leftarrow$ FIRST-ATTEMPT-MULTIPLY[$x_0, y_1$]
8  $p_4 \leftarrow$ FIRST-ATTEMPT-MULTIPLY[$x_0, y_0$]
9  return $2^n p_1 + 2^{n/2}(p_2 + p_3) + p_4$
```

What is the running time of the above algorithm? We associate the worst-case running time with the representational length of the integer n. That is, $T(n) = 4T(n/2) + \Theta(n)$, $T(1) = \Theta(1)$. Applying, Master Theorem we have that $T(n) = \Theta(n^2)$.

Remark 4. FIRST-ATTEMPT-MULTIPLY is just a complicated way to achieve the same running time as the elementary school algorithm! Therefore, we haven't done anything interesting yet. Below, we will see how to modify this algorithm into a much more efficient one.

2.3 A smart idea

We will borrow an idea from C.F. Gauss (a great mathematician of 18th century). Gauss observed that we can multiply two complex numbers $(a + bi)$ and $(c + di)$ by doing only 3 instead of 4 multiplications.

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

Clearly, we have to multiply ac, bd, ad, bc. We observe that:

$$(a + b)(c + d) = ac + ad + bc + bd \Rightarrow ad + bc = (a + b)(c + d) - ac - bd$$

Therefore, with only three multiplications $(a + b)(c + d)$, ac, bd we can determine $ad + bc$, ac, bd. We adopt this idea to save one recursive call from FIRST-ATTEMPT-MULTIPLY.

2.4 A "fast" integer multiplication algorithm

To adopt the idea of Gauss in the context of integer multiplication we observe that we can compute $xy = 2^n x_1 y_1 + 2^{n/2}(x_1 y_0 + x_0 y_1) + x_0 y_0$ by doing only three multiplications: $(x_1 + x_0)(y_1 + y_0)$, $x_1 y_1$ and $x_0 y_0$. This is because

$$(x_1 y_0 + x_0 y_1) = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0 \tag{2.2}$$

Recall that the "bottleneck" in our algorithm is the multiplication and not the additions/subtractions. Additions and subtractions cost $\Theta(n)$. Therefore, adding (to FIRST-ATTEMPT-MULTIPLY a *constant* number of additions and subtractions does not hurt the linear running time (when excluding the cost of the recursive calls).Our algorithm is as follows:

```
D & C-MULTIPLY[$x, y$]
1  if the length of $x$ (and $y$) is 1 bit
       then
2          return $x \cdot y$


3  $x = x_1 x_0$ (extract $x_1$, $x_0$ from $x$)
4  $y = y_1 y_0$
5  $p_1 \leftarrow$ D & C-MULTIPLY[$x_1, y_1$]
6  $p_2 \leftarrow$ D & C-MULTIPLY[$x_1, y_0$]
7  $p_3 \leftarrow$ D & C-MULTIPLY[$x_1 + x_0, y_1 + y_0$]
8  return $2^n p_1 + 2^{n/2}(p_3 - p_1 - p_2) + p_2$
```

Remark 5. At a first glance it seems that there is no big deal since we save only one multiplication. This intuition is not correct. Note that we do not merely save one multiplication. We save one multiplication *in every level of the recursion.* As mentioned in class, reducing the number of recursive calls has a significant effect in the running time of a Divide and Conquer algorithm (even when increasing the cost of the "combine-step" - in this example the "combine-step" has asymptotically the same cost in both algorithms). To gain a better understanding you may wish to draw the recursion tree and convince yourself.

The recurrence relation which associates the worst-case running time with the length of each number is $T(n) = 3T(n/2) + \Theta(n)$, $T(1) = \Theta(1)$. By applying Master Theorem we have that $T(n) = \Theta(n^{\log_2 3}) = O(n^{1.59})$.

2.5 Correctness of the algorithm

Most of the Divide and Conquer algorithms have a straightforward proof of correctness. The structure of the algorithm is such that it resembles to an inductive definition. Therefore, a straightforward structural induction does the job. Since we have only mentioned simple and strong(complete) induction we will use strong induction to prove correctness. The whole "difficulty" lies in the correctness of the combine step. The term "correctness of the combine step" is a bit vague since we must have a Problem associated with the "combine-step" of the Divide and Conquer algorithm. Furthermore, the definition of this Problem must be useful for the correctness of the whole algorithm (w.r.t. the -other- Problem that this algorithm solves). Well, most of the times the definition of the Problem where the "combine-step" is correct is self-evident. That's why most of the times we omit the details of defining it explicitly.

Where is the "combine-step" of this Divide and Conquer algorithm? We wrote the combine-step on the **return** since the combine-step is nothing more than a simple calculation (additions/subtractions). In this case the Problem associated with the combine-step is that given the results of the multiplications $(x_1 + x_0)(y_1 + y_0)$, x_1y_1 and x_0y_0 determine xy. The correctness of the combine-step is immediate from equations 2.2 and 2.1. To prove the correctness of D & C-MULTIPLY we have to prove that the algorithm terminates and that if it terminates the preconditions imply the postconditions.

To show termination we associate the length of the numbers in each recursive call with a strictly decreasing sequence in $\mathbb{N}$. Therefore, by standard arguments the algorithm terminates.

We do strong induction on the length of the integers in the input. (Basis) If the length of x is 1 (and the length of y is 1) then the algorithm multiplies x and y and therefore it returns the correct value. (Inductive Step) Suppose that the algorithm is correct for every integer of (representational) length $n < k$. We want to show that it is correct for integers of length k. Since the recursive calls are made to integers of length less than k by the inductive hypothesis and by the fact that the combine-step is correct we have that it holds for k.

Remark 6. For simplicity, in the above proof of correctness we miss details regarding the fact that not all integers are powers of two. Even if we restrict the input such that the integers are powers of two then not all the integers involved in the recursive calls are powers of two. Why?

Exercise 3. Modify the D & C-MULTIPLY algorithm by adding ceilings and floors so that it works correctly for all integers. In your modification you should always consider the fact that the numbers in the input might be of length 3. Prove in detail termination for this case. Modify the rest of the proof of correctness. You have to modify the Induction Basis (extend your basis when the length is more than 1 - what happens if $x_1 = 1$ and $x_0 = 1$? why you cannot use the inductive hypothesis in this case?)