Algorithms Fall 2019

Lecture : Rank, Merging, Merge Sort and Recurrence Relations

Imdad Ullah Khan

Contents

1 Karatsuba Algorithm

In the first lecture we discussed that we could add or multiply arbitrarily large integers by putting all the digits of the number in an array and then add/multiply digit by digit. So we know how to multiply two integers using the grade school algorithm but we haven't yet discussed if there's a more efficient way to multiply integers. Since we've been using the divide and conquer approach for different problems which had a brute-fore running time of n^2, lets see if we can use a similar idea for multiplication.
Let x and y be two $2n$ digit numbers then

$$x = \sum_{i=0}^{2n-1} x_i 10^i,\ y = \sum_{i=0}^{2n-1} y_i 10^i$$

And we know the following axiom for Real Numbers

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

Using these two ideas we can come up with an apporach for *dividing* the multiplication problem. In particular we can divide x and y into two n digit numbers a, b and c, d repsectively. Where

$$a = \sum_{n}^{2n-1} x_i 10^{i-n} \qquad \text{(The left half of } x\text{)}$$

$$b = \sum_{0}^{n-1} x_i 10^{i} \qquad \text{(The right half of } x\text{)}$$

$$c = \sum_{n}^{2n-1} y_i 10^{i-n} \qquad \text{(The left half of } y\text{)}$$

$$d = \sum_{0}^{n-1} y_i 10^{i} \qquad \text{(The right half of } y\text{)}$$

Then

$$\begin{aligned} xy &= (10^n a + b)(10^n c + d) \\ &= 10^{2n}(\underbrace{ac}_{\text{1 multiplication}}) + 10^n(\underbrace{ad + bc}_{\text{2 multiplications}}) + \underbrace{bd}_{\text{1 multiplication}} \end{aligned}$$

Example : If $x = 2731$ and $y = 1593$, then

$$\begin{aligned} x &= 27 \times 10^2 + 31 & y &= 15 \times 10^2 + 93 \\ a &= 27 & b &= 31 \\ c &= 15 & d &= 93 \end{aligned}$$

Giving us:

$$xy = 10^2(27 \times 15) + 10^2(27 \times 93 + 31 \times 15) + 31 \times 93$$

Now each of the numbers a, b, c, d have n digits. The multiplications by powers of 10 is just a shift operation so we won't count those. Leaving that we have four n digit numbers to multiply and then we perform three addition operations. So our recurrence relation looks like

$$T_1(n) = \begin{cases} 1 & \text{n = 1} \\ 4T_1\left(\frac{n}{2}\right) + 3n & n > 1 \end{cases}$$

We will later see, when we solve this recurrence that we get a running time that scales quadratically with n.

But we already have an n^2 algorithm which is much simpler than this. So it would seem that our approach for dividing the problem doesn't really help. However, the four multiplications we had to perform could be reduced to three by using

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

Taking from our previous example, we can see that this is true:

$$31 \times 15 + 27 \times 93 = (27 + 31)(15 + 93) - 27 \times 15 - 31 \times 93$$

$$465 + 2511 = (58)(108) - 405 - 2883$$

$$2976 = 2976$$

The Karatsuba algorithm for multiplying two integers uses this improvement. Reducing the number of multiplications by just 1 doesn't seem like much, but as we'll see this gives us a significantly better running time when this is done at every step of the recursion. Our new recurrence relation is

$$T_2(n) = \begin{cases} 1 & \text{n} = 1 \\ 3T_2\left(\frac{n}{2}\right) + 3n & n > 1 \end{cases}$$

We will see that this comes out to be about $n^{1.58}$ which is a big improvement over the n^2 algorithm. As an illustration, lets take n = 1000: For the brute force n^2 algorithm, we get:

$$1000^2 = 1000000$$

Whereas for the Karatsuba's $n^{1.58}$ algorithm, we get:

$$1000^{1.58} = 54954$$

Comparing these values shows us that Karatsuba reduces the execution time to:

$$\frac{54954}{1000000} \times 100 = 5.94\%$$

of the brute force execution

Note : Since the 3 in $3n$ is just a constant, we will ignore it and just use n from now on. This simplifies our calculations and is asymptotically equal as we will see below (the master theorem). The recurrence for this algorithm that we will refer to from now on would be

$$T_2(n) = \begin{cases} 1 & \text{n} = 1 \\ 3T_2\left(\frac{n}{2}\right) + 3n & n > 1 \end{cases}$$

2 $Rank_A(x)$

Problem : Given an array A and an element x (which may or may not belong to A), we want to find the number of elements in A that are "less than" x. This is called the Rank of x in A and is written as $Rank_A(x)$

Example : Let A =

5	4	6	9	2	7	5	8

Then $Rank_A(5) = 2$, $Rank_A(3) = 1$, $Rank_A(1) = 0$. Notice the rank of the minimum element of A, the maximum element of A and an integer larger than the maximum of A.

Computing the Rank : One way to compute the rank of an element is just by a simple linear scan of the array A. This is a simple algorithm, taking n steps in the worst-case scenario.

Rank in a sorted array: Suppose the A is sorted, then can we do any better than n steps ? Well we have already studied the binary search algorithm for finding an element in a sorted array. Binary search takes about $\log n$ steps. We can use a slightly modified version of it to find the rank of an element. Recall what binary search returns when the search key x is not in the array.

Rank of two elements at once : Suppose now that A is sorted and instead of one element x, you're given two elements x and y and you have to compute the rank of both of these elements. Without loss of generality, we can assume $x \leq y$ (if $x > y$ just swap the two). Now we know we can compute the rank of one element using binary search. For two elements we can just repeat the same process for the second element. So the total steps taken would be $\log n + \log n = 2\log n$. Another thing we could do is that suppose $Rank_A(x)$ turns out to be i. Then instead of using binary search on the whole array, we can use binary search on $B = A[i \dots n]$. Then $Rank_A(y) = Rank_B(y) + Rank_A(x)$. Concretely, we can do the following.

Algorithm 1 : Finding Rank of Two Elements

```
$rankXA \leftarrow getRank(x, A)$
$rankYB \leftarrow getRank(y, \{A[rankXA]..A[n]\})$
return {rankXA, rankYB + rankXA}
```

How much improvement do we get by using this algorithm as compared to just using binary search on A twice? Well let's see, any improvement that we do get is because now we're searching over a smaller array. The size of that array is $n - Rank_A(x)$. So what's the largest possible value for $n - Rank_A(x)$? that would be our worst case. The largest value is when $Rank_A(x) = 0$. So in that case the size of the new array would be n and it would take us $\log n$ steps to compute $Rank_A(y)$ and a total of $\log n + \log n$ steps to compute both the ranks. So in the worst case there is no improvement by this algorithm.

Rank of n elements at once : Okay so now, instead of two elements x and y we can generalize the problem. Suppose we're given two sorted arrays, A and B. Our task is to find the rank in A of all elements of B. That is we have to compute $\{Rank_A(x)|x \in B\}$. We solve this problem by extending our algorithm for the 2 element case. The algorithm is given below.

Algorithm 2 : Computing $\{Rank_A(x)|x \in B\}$ using Binary Search

```
for i = 1 to n do
    if i == 1 then
        $rankB[i] \leftarrow getRank(B[i], A)$
    else
        $rankB[i] \leftarrow (rankB[i-1] + getRank(B[i], \{A[rankB[i-1]]...A[n]\}))$
```

Let's analyze the running time for this algorithm. For two elements x and y we calculated that the running time should be $2\log(n)$. When we extend it to n elements, again at each turn, in the worst case, the size of the remaining array that we have to look at is not reduced at all. So the total running time would be $\log n + \log n + logn + \ldots + logn = n \log n$.

Taking a step Back : Can we do any better than this ? Let's consider again, the linear scan approach that was discussed in the beginning. As it turns out, in one linear pass through the array A we can determine the rank of all elements of B. The algorithm for that is quite simple, and is described below.

Algorithm 3 : Computing $\{Rank_A(x) | x \in B\}$ using Linear Search

```
r ← 0
for i = 1 to n do
    if A[i] > B[j] then
        $rankB[j] \leftarrow r$
        $j \leftarrow j + 1$
    else
        $r \leftarrow r + 1$
```

This concludes our discussion on rank. Now we move on to a seemingly different problem. That of "merging" two sorted arrays. It is quite easy to see that this algorithm takes $2n$ comparisons.

3 Merging two arrays

Problem : The merge problem is as follow. We're given two arrays A and B, each of size n. Both of these arrays are in sorted order. We have to make another sorted array C of size $2n$, that contains all the elements of A and B.

Example : Let A = $\begin{array}{|c|c|c|c|}\hline 1 & 3 & 4 & 7 \\ \hline\end{array}$, B = $\begin{array}{|c|c|c|c|}\hline 2 & 5 & 6 & 8 \\ \hline\end{array}$ Then C = $\begin{array}{|c|c|c|c|c|c|c|c|}\hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline\end{array}$

Solution : One way to do this is to just combine the two arrays into one big array C and then just sort C. Since C is of size $2n$ this would take about $2n\log(2n)$ comparisons, if we use insertion sort with binary search. Using selection sort or bubble sort it would take about $\frac{2n(2n-1)}{2}$ comparisons. But since the two arrays A and B are sorted perhaps we can do better. If you really think about this problem is exactly equivalent to determining the rank of all elements of B in A (or vice versa). If we know the ranks we know in which position the elements of A and B would go in the combined array. Like in the example described above we know

$$Rank_A(2) = 1, Rank_A(5) = 3, Rank_A(6) = 3, Rank_A(8) = 4$$

So we know that 2 would come after one element of A has been inserted (which one?). 5 would come after three elements of A have been inserted into C and so on. And we know that finding the ranks of all elements of B in A takes just n steps. So we should be able to solve this problem in about n steps. Typically though, the way the solution is implemented doesn't require explicitly finding the ranks. We just maintain two running pointers for A and B, compare the elements at those indices, insert the smaller one into C and increment the corresponding pointer. The exact algorithm is given below

Algorithm 4 : Merging A and B

```
$p \leftarrow 1$
$q \leftarrow 1$
for i = 1 to 2n do
    if ($p > n$ or $A[p] > B[q]$) then
        $C[i] \leftarrow B[q]$
        $q \leftarrow q + 1$
    else
        $C[i] \leftarrow A[p]$
        $p \leftarrow p + 1$
```

4 Merge Sort

The above strategy allows us to define another sorting algorithm, called the Merge Sort. Merge Sort works as follows. Given an array, A of size n we divide the array into two halves $A[1 \dots n/2$ and $A[n/2+1 \dots n]$, recursively sort the two halves and then merge the sorted halves using the merge algorithm we just discussed. The pseudocode is given below.

Algorithm 5 : Merge Sort

```
function MergeSort(A)
    if size(A) == 1 then
        return $A$
    else
        $L \leftarrow MergeSort(A[1 \dots \text{size}(A)/2])$
        $L \leftarrow MergeSort(A[\text{size}(A)/2 + 1 \dots \text{size}(A)])$
        return Merge($L, R$)
```

Runtime Analysis : To analyze the running of this algorithm we can set up a recurrence relation. We know that to sort an array of size m we have to sort two arrays of size $m/2$ and then merge the two. We know that the time taken to merge two arrays of size $m/2$ is m. Denote by $T(m)$ the number of comparisons MergeSort takes to sort an array of m elements. It is clear from the algorithm and preceding discussion that

$$T(m) = 2T\left(\frac{m}{2}\right) + m \tag{1}$$

.

We want to find a closed form solution to this recurrence (rather than this recursive/inductive definition). One way to find such a closed form solution is to just extend the recursive relation. Suppose we want to find what is $T(n)$, then repeatedly applying (1) we get

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n \\ &= 2\left(2\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right)\right) + n + n \\ &= 2\left(2\left(2\left(2T\left(\frac{n}{16}\right) + \frac{n}{8}\right)\right)\right) + n + n + n \end{aligned}$$

In general the k^{th} line of the above sequence of equation is

$$T(n) = 2^k \cdot T\left(\frac{n}{2^k}\right) + \overbrace{n + n + \ldots + n}^{k \text{ times}}$$

As we know from the algorithm that $T(1) = 1$ and it is easy to see that the maximum value of k is $\log n$ (assuming n is a power of 2 to rid ourselves from dealing with floors and ceilings), we get that the last equation will be

$$\begin{aligned} T(n) &= 2^{\log n} \cdot T\left(\frac{n}{2^{\log n}}\right) + \overbrace{n + n + \ldots + n}^{\log n \text{ times}} \\ &= n \cdot T\left(\frac{n}{n}\right) + \overbrace{n + n + \ldots + n}^{\log n \text{ times}} \\ &= n \cdot 1 + n \log n \\ &= n \log n + n \end{aligned}$$

We will now discuss recurrence relations, and ways to solve them.

5 Recurrence Relations

The following two paragraphs are mostly copy pasted from `https://courses.engr.illinois.edu/cs573/fa2010/notes/99-recurrences.pdf`

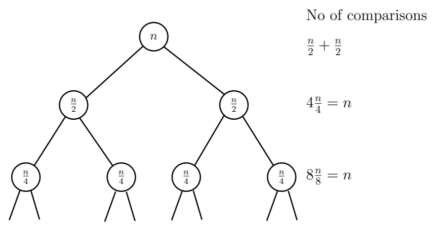
A recurrence is a recursive description of a function, or in other words, a description of a function in terms of itself. Like all recursive structures, a recurrence consists of one or more base cases and one or more recursive cases. Each of these cases is an equation or inequality, with some function value $f(n)$ on the left side. The base cases give explicit values for a (typically finite, typically small) subset of the possible values of n. The recursive cases relate the function value $f(n)$ to function value $f(k)$ for one or more integers $k < n$. For example, the following recurrence describes the identity function $f(n) = n$:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n-1) + 1 & \text{otherwise} \end{cases}$$

Recurrences arise naturally in the analysis of algorithms, especially in divide and conquer algorithms. In many cases, we can express the running time of an algorithm as a recurrence, where the recursive cases of the recurrence correspond exactly to the recursive cases of the algorithm. Recurrences are also useful tools for solving counting problems — How many objects of a particular kind exist?
By itself, a recurrence is not a satisfying description of the running time of an algorithm. Instead we would like a closed-form solution to the recurrence; this is a non-recursive description of a function that satisfies the recurrence. We will disscuss now, how to solve some types of these recurrence relations.

5.1 Solving Recurrence Relations

One way to think about what's going on in a recurrence relation is to visualize it as a tree diagram. We will consider the Mergesort algorithm and Karatsuba algorithm and see how we can visualize them and see what's happening. This would hopefully provide us with some insight as to how much work our algorithm is doing. We represent each instance of a called function as a node in the tree and calls to a function represent edges. This allows us to figure out how many function calls are made and helps us figure out the amount of work that's being done. Below is a tree diagram for the merge sort algorithm. As a reminder, mergesort splits the array to be sorted into two halves, sorts each half independently and then merges the two halves outside of the recursion. For merging it takes time equal to length of the merged array. For a more detailed description, refer to notes on mergesort.

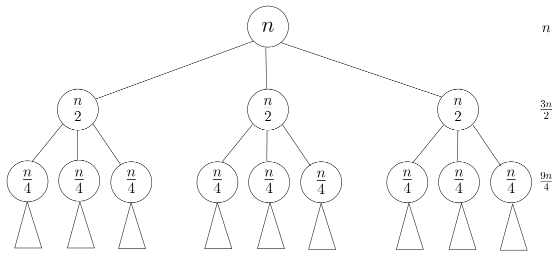


In the beginning there is a call to mergesort with a problem of size n. That functions then calls two other instances of mergesort with problem sizes of $\frac{n}{2}$, each of which make two calls to mergesort with problem sizes of $\frac{n}{4}$ and so on. We know that the calls would stop when the the size of the array is 1 (or two.. that's just an implementation issue). So we know there are about $\log n$ *levels* of the tree. Another thing we can see from this diagram is that at every level n comparisons are made for merging. That means the total time for this algorithm is $n \log n$.

5.2 Analysis Of the Karatsuba Algorithm

Let's now see how the recursion tree for the Karatsuba Algorithm looks like. In the Karatsuba algorithm at each step the problem is divided into 3 subproblems of size $\frac{n}{2}$. So it's recursion tree looks as follows

The number of levels of this tree would be the same as in mergesort (i.e $\log n$) since at each level the problem size is halved, just like in mergesort. However the number of problems grows faster in Karatsuba. And as a result we can see in the diagram that the amount of work being done at each level is not the same. The amount of work at the top (0_{th}) level is n, at the first level $1.5n$ and $2.25n$ at the second level. So it's not immediately clear what the total running time of the algorithm would be. But we can sort of see a pattern. At each level the problem size gets split into 2 and the number of problems grows by a factor of 3. So at level i the number of problems is 3^i and the size of each problem is $\frac{n}{2^i}$. So the total work at each level is $\frac{3^i \times n}{2^i}$.



We can now figure out the total running time that this algorithm would take by summing up the running time over all levels.

$$T_2(n) = \sum_{i=0}^{\log n} \frac{3^i \times n}{2^i} = n \sum_{i=0}^{\log n} \left(\frac{3}{2}\right)^i = n \left(\frac{1 - \frac{3}{2}^{\log n + 1}}{1 - \frac{3}{2}} \right)$$

$$\in \mathcal{O}\left(n \times \left(\frac{3}{2}\right)^{\log_2 n}\right) \in \mathcal{O}\left(n \times \left(\frac{3^{\log_2 n}}{n}\right)\right) \in \mathcal{O}\left(3^{\frac{\log_3 n}{\log_3 2}}\right)$$

$$\in \mathcal{O}\left(n^{\frac{1}{\log_3 2}}\right) \in \mathcal{O}\left(n^{\frac{1}{0.6309}}\right) \in \mathcal{O}\left(n^{1.58}\right)$$

5.3 Analysis Of the Naive Divide & Conquer Multiplication Algorithm

We can similarly imagine a tree for this algorithm. At each step the number of problems would be multiplied by 4 and the problem size would be halved. So at each level the amount of work would be $\frac{4^i \times n}{2^i}$. And the total work would be

$$T_1(n) = \sum_{i=0}^{\log n} \frac{4^i \times n}{2^i} = n \sum_{i=0}^{\log n} 2^i = n \times 2^{log_2 n + 1}$$

$$\in \mathcal{O}\left(n \times 2^{log_2 n}\right) \in \mathcal{O}\left(n \times n\right) \in \mathcal{O}\left(n^2\right)$$

So we see that the running time in this case turns out to be n^2 as was mentioned previously.

6 Substitution method for Solving Recurrences

While we have seen that the recursion tree method works pretty well for solving recurrences, we would like to have a method that is simpler and does not require much drawing. Such a method is the substitution method, which simply requires to substitute the value of the function for smaller values from the recurrence and continue until we see a pattern and can't manage it. Let us use the recurrence relation for merge sort, which is:

$$T(n) = 2T(\frac{n}{2}) + n$$

$$T(n) = 2T(\frac{n}{2}) + n = 2(2T(\frac{n}{4}) + \frac{n}{2}) + n = 2 * 2 * (2T(\frac{n}{8}) + \frac{n}{4}) + n = 2 * 2 * 2(T(\frac{n}{8}) + n + n + n$$

$$\vdots \qquad \vdots$$

$$= \underbrace{2 * 2 * 2... * 2}_{\text{k}} * T(\frac{n}{2^k}) + \underbrace{n + n + ... + n}_{\text{k}}$$

$$\vdots \qquad \vdots$$

$$= \underbrace{2 * 2 * 2... * 2}_{\text{log n}} * T(\frac{n}{2^{\log n}}) + \underbrace{n + n + ... + n}_{\text{log n}}$$

$$= 2^{\log n} * 1 + n \log n$$

$$= n \log n + n$$

Hence we get $O(n \log n)$ for the merge sort algorithm.

6.1 Using the substitution method for solving the naive Divide and Conquer multiplication problem

As stated previously, the naive divide and conquer algorithm for multiplication has the recurrence relation

$$T_1(n) = 4T_1(\frac{n}{2}) + n$$

$$T_1(n) = 4T_1(\frac{n}{2}) + n = 4(4T_1(\frac{n}{4}) + \frac{n}{2}) + n$$

$$= 4(4(4T_1(\frac{n}{8}) + \frac{n}{4}) + \frac{n}{2}) + n = 4 * 4 * 4T_1(\frac{n}{8}) + 4 * 4 * \frac{n}{4} + 4 * \frac{n}{2} + n$$

$$\vdots \qquad \vdots$$

$$= \underbrace{4 * 4 * 4... * 4}_{\text{k}} * T_1(\frac{n}{2^k}) + \sum_{i=0}^{k} (4^i * \frac{n}{2^i})$$

$$\vdots \qquad \vdots$$

$$= \underbrace{4 * 4 * 4... * 4}_{\log n} * T_1(\frac{n}{2^{\log n}}) + \sum_{i=0}^{\log n} 2^i n = 4^{\log n} * 1 + n(2^{\log n + 1})$$

$$= 2^{2 \log n} + n * n = n^2 + n^2 = 2n^2$$

Hence we get $O(n^2)$ runtime for the naive divide and conquer multiplication.

6.2 Using substitution for Karatsuba

Finally, we look into the Karatsuba algorithm and determine its runtime via the substitution method. So we have:

$$T_2(n) = 3T_2(\frac{n}{2}) + n = 3(3T_2(\frac{n}{4}) + \frac{n}{2}) + n = 3(3(3T_2(\frac{n}{8}) + \frac{n}{4}) + \frac{n}{2}) + n$$

$$\vdots \qquad \vdots$$

$$= \underbrace{3 * 3 * 3 ... * 3}_{\text{k}} T_2(\frac{n}{2^k}) + \sum_{i=0}^{k} (3^i * \frac{n}{2^i})$$

$$\vdots \qquad \vdots$$

$$= \underbrace{3 * 3 * 3 ... * 3}_{\log n} T_2(\frac{n}{2^{\log n}}) + \sum_{i=0}^{\log n} (3^i * \frac{n}{2^i}) = 3^{\log n} * 1 + n * \sum_{i=0}^{\log n} (\frac{3^i}{2^i})$$

$$= 3^{\log n} + n * (\frac{1 - (\frac{3}{2})^{\log n}}{1 - (\frac{3}{2})}) = 3^{\log n} + n * (\frac{1 - (\frac{3^{\log n}}{2^{\log n}})}{-(\frac{1}{2})})$$

$$= 3^{\log n} + n * 2(\frac{3^{\log_2 n}}{n}) = 3(3^{\frac{\log_3 n}{\log_3 2}}) = 3(3^{\log_3 n})^{\frac{1}{\log_3 2}} = 3n^{\frac{1}{\log_3 2}} = 3n^{1.58}$$

Hence, we get $O(n^{1.58})$ as the runtime for the Karasuba algorithm.

7 Master Theorem

We've had to analyze the running times of a bunch of divide and conquer algorithms over the past 2,3 classes which had pretty similar recurrence relations. Setting up a recurrence relation, and then figuring out how much work is being done at each level of the recursion tree and then summing up over all levels is a pain and we would like to have some general results that can be used to directly get the running time of a divide and conquer algorithm. As it turns out, most divide and conquer algorithms tend to have the following type of recurrence relation.

$$T(n) = aT\left(\frac{n}{b}\right) + \mathcal{O}(n^d)$$

Where a corresponds to the number of subproblems, n/b corresponds to the size of each subproblem and $\mathcal{O}(n^d)$ is the work performed in the conquer and combine step. There's a theorem called the Master theorem which gives us the following results for these types of recurrence relations

$$T(n) = \begin{cases} \mathcal{O}(n^d) & d > \log_b(a) \\ \mathcal{O}(n^d \log(n)) & d = \log_b(a) \\ \mathcal{O}(n^{\log_b(a)}) & d < \log_b(a) \end{cases}$$

7.1 Analyzing running times using the Master Theorem

Let's now apply to master theorem to the various recurrence relations we've discussed uptil now. For the merge sort algorithm $a = 2$, $b = 2$ and $d = 1$. So $\log_b(a) = \log_2(2) = 1$ which is the same as d. So case 2 applies here and we get a running time of $n \log n$, the same as when we computed using the recursion tree.

For the naive divide and conquer multiplication algorithm we have $a = 4$, $b = 2$ and $d = 1$. Which means $\log_b(a) = \log_2(4) = 2$ which is greater than d. So case 3 applies and we get a running time of

$$n^{\log_b(a)} = n^{\log_2(4)} = n^2$$

For Karatasuba too, case 3 applies. You can put in the relevant values and see that the running time comes out to be about $n^{1.58}$ same as we calculated earlier.