VII

DIVIDE AND CONQUER

Divide and Conquer (D&C) is a technique for designing algorithms that consist of decomposing the instance into a number of smaller subinstances of the same problem, solving successfully and independently each subinstance and then combining the sub solutions thus derived to obtain the solution of the original instance.

7.1 The general template:

In D&C, to be worthwhile, 3 conditions must be met. The decision when to use the basic subalgorithm rather than to make recursive calls must be taken judiciously – it must be possible to decompose an instance into subinstances and to recombine the subsolutions fairly efficiently, and the subinstance should as far as possible be of about the same size. In most D&C algorithms $l \approx \frac{n}{b}$, where l: size of subinstance, n: size of original instance, for some constant b. The running time for such D&C is almost automatic. Let $g(n)$ be the time required by D&C not counting the time needed for the recursive calls. The total time $t(n)$ taken by D&C is something like: $t(n) = l\, t(n \div b) + g(n)$, provided n is large enough. If there exists a k such that $g(n) \in \Theta(n^k)$ then we conclude that

$$t(n) \in \begin{cases} \Theta(n^k) & \text{if } l < b^k \\ \Theta(n^k \log n) & \text{if } l = b^k \\ \Theta(n^{\log_b l}) & \text{if } l > b^k \end{cases}$$

It remains to see how to determine whether to divide the instance and make recursive calls, or whether the instance is so simple that it is better to invoke the basic sub algorithm directly. Although this choice does not affect the order of the execution time of the algorithm, we are also concerned to make the multiplicative constant hidden in the Θ notation as small as possible. With most D&C algorithms, this decision is based on a simple "threshold" usually denoted n_0.

7.2. Introduction - Multiplying large integers

Classic algorithm requires $\Theta(n^2)$ to multiply n figure numbers. Can we do better? A previously seen algorithm called divide and Conquer Technique consisted of reducing the multiplication of two n-figure numbers to four (4) multiplications of $\frac{n}{2}$-figure numbers. Unfortunately, that one does not offer much help unless we do some more work. We must find a way to reduce the original multiplicat not to four (4) but to three (3) half-size multiplications.
Let's illustrate the process on the following example: 981×1234
First, pad the shorter operand with 0 to the left 0981.
Second, split each operand into two (2) halves:

0981 gives $w = 09$ and $x = 81$
1234 " $y = 12$ " $z = 34$

Third, notice that:

$$981 = 10^2 w + x \quad \text{and} \quad 1234 = 10^2 y + z$$

and then the required product can be computed as:

$$\begin{aligned} 981 \times 1234 &= (10^2 w + x)(10^2 y + z) \\ &= 10^4 wy + 10^2 (wz + xy) + xz \\ &= 1080000 + 127800 + 2754 = 1210554 \end{aligned}$$

If you think there was no gain, you're right.
This procedure still requires 4 half size multiplications
The key observation is that there is no need to compute both wz and xy: all we really need is the sum of these 2 terms. Is it possibl Yes it is. Remember, we also need the values wy and xz to apply the above formula. Now, consider this

$$r = (w + x) \times (y + z) = wy + (wz + xy) + xz$$

After only one multiplication, we obtain the sum of all three terms needed to calculate the desired product. This suggests proceeding as follows.

$$p = wy = 09 \times 12 = 108$$
$$q = xz = 81 \times 34 = 2754$$
$$r = (w+x)\times(y+z) = 90 \times 46 = 4140$$

and finally

$$981 \times 1234 = 10^4 p + 10^2 (r - p - q) + q$$
$$= 1080000 + 127800 + 2754 = 1210554$$

Thus the product can be reduced to 3 multiplications of 2 figure numbers: 09×12, 81×34, 90×46, together with a certain number of shifts (multiplications by powers of 10), additions and subtractions.

Is it worth performing 4 more additions to save one multiplication? Of course NOT, when we are multiplying small numbers. However, when the numbers to be multiplied are large, it is worthwhile. When operands get larger, the time taken by additions and subtractions and shifts become negligible compare to the time taken by a single multiplication. It seems reasonable to expect 25% cut of computing time. We will see our savings will be even better.

Let's analyze what we achieved. To multiply 2 n-figure numbers classic algorithm requires a time $h(n) = cn^2$ for some constant c that depends on the implementation (of course this is a simplification since in reality time is more like $cn^2 + bn + a$.) Similarly, let $g(n)$ be the time taken by the divide and conquer algorithm (DC A) to multiply 2 n-figure numbers, not counting the time needed to perform the three half-size multiplications. In other words $g(n)$ is the time needed for additions

shifts and various overheads.

It is easy to implement these operations so that $g(n) \in \Theta(n)$.

Let's not worry whether n is even or odd, or the 2 numbers are not of the same length.

If each of the three half size multiplications is carried out by the classic algorithm, the time needed to multiply 2 n-figure numbers is:

$$3\,h(n/2) + g(n) = 3.c.\left(\frac{n}{2}\right)^2 + g(n) = \frac{3}{4}h(n) + g(n)$$

$h(n) \in \Theta(n^2)$ and $g(n) \in \Theta(n)$ that is negligible compared to $\frac{3}{4}h(n)$ when n is large enough, which shows a gain of 25%. which is the improvement that we anticipated.

To do better than this, we ask the question "how should the subinstance be solved?" If they are small then the classic algorithm may be the best. However when they are sufficiently large, we might use our new algorithm "recursively".

Our algorithm can multiply 2 n-figure numbers in a time

$$t(n) = 3\,t\left(\frac{n}{2}\right) + g(n)$$

when n is even and sufficiently large, whose solution is

$$t(n) \in \Theta\left(n^{\log 3} \mid n \text{ is a power of } 2\right)$$

Since $\log_2 3 \approx 1.585$ is smaller than 2, this algorithm can multiply 2 large integers much faster than the classic multiplication algorithm, and the bigger n the more this improvement is worth having.

Few important issues have been left out. How do we proceed with odd (length) number? Although multiplicand and multiplier are of size $\frac{n}{2}$ it may happen that their sum overflows and is of size 1 bigger. Therefore, it was slightly incorrect to claim that:

$r = (w+x).(y+z)$

involves a $\frac{1}{2}$ size multiplication, How does this affect the analysis of running time? How do we multiply numbers of different sizes? ... etc.

Numbers of odd length are easily multiplied by splitting them as nearly down the middle as possible. If n then split into $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ numbers. The second question is trickier. Consider multiplying: 5678 by 6789. The operands split the following way, $w = 56$, $x = 78$, $y = 67$, and $z = 89$

$$p = w.y = 56 \times 67$$

$$q = xz = 78 \times 89, \text{ and}$$

$$r = (w+x) \times (y+z) = 134 \times 156$$

The 3rd multiplication involves 3-figure numbers, and thus it is not really $\frac{1}{2}$ size compared with the original multiplication. However the size of $w+x$ and $y+z$ can't exceed $1 + \lceil \frac{n}{2} \rceil$.

To simplify the analysis, let $t(n)$ denote the time taken by this algorithm in the worst case to multiply 2 numbers of size at most n. By definition, $t(n)$ is a nondecreasing function. When n is sufficiently large, our algorithm reduces the multiplication of 2 numbers of size at most n to 3 smaller multiplications: $p = wy$, $q = x.z$, and $r = (w+x)(y+z)$ of sizes at most $\lfloor \frac{n}{2} \rfloor$, $\lceil \frac{n}{2} \rceil$ and $1 + \lceil \frac{n}{2} \rceil$, respectively in addition to smaller manipulations in $O(n)$. Therefore there exists a constant c such that:

$$t(n) \leq t(\lfloor n/2 \rfloor) + t(\lceil \tfrac{n}{2} \rceil) + t(1 + \lceil \tfrac{n}{2} \rceil) + cn \quad \text{for large } n.$$

The result of this recurrence yields:

$$t(n) \in O(n^{\log 3})$$

A worst case analysis yields in fact $\Theta(n^{\log 3})$

How about different sizes? Let u and v be integers of sizes m and n, respectively. If m and n are within a factor or two of each other

then it is best to pad the smaller operand with nonsignificant zeros. However, this approach is discouraged when one operand is much larger than the other. It could be much worse than the classic algorithm! It is simple to combine both algorithms, to obtain a truly better algorith The idea is to slice the longer operand v into blocks of size m and to use the divide and conquer algorithm to multiply u by each block of v, so that the DCA is used to multiply operands of the same size.

7.4 Sorting

Let T[1..n] be an array of n elements, to be sorted in an ascending order. We saw before different sorting techniques - Selection - Insertion - and Heapsort - The average case, the worst case of the latter is $\Theta(n \log n)$, whereas the former take $\Theta(n^2)$. There are several other techniques for sorting that follow the divide and conquer template.

7.4.1 Sorting by merging:

It consists on separating the array T into 2 parts whose sizes are as nearly equal as possible, sorting these parts by recursive call and then merging the solutions for each part, being careful to preserve the order. To do this, we need an efficient algorithm for merging two sorted arrays U and V into a single array T. For efficiency and ease we assume we have additional storage available at the end of both U and V to be used as a sentinel. (This technique is garanteed to work only if the value of the sentinel is set to be bigger than every element in U and V, which we'll denote by "∞".)

```
procedure merge (U[1.. m+1], V[1.. n+1], T[1.. m+n])
  {merges sorted arrays U[1..m] and V[1..n] into T[1..m+n];
    U[m+1] and V[n+1] are used as sentinels}
  i, j ← 1
  U[m+1], V[n+1] ← ∞
  for k ← 1 to m+n do
    if U[i] < V[j]
      then T[k] ← U[i]; i ← i+1
      else T[k] ← V[j]; j ← j+1
```

The merge sorting is as follows, where we use insertion sort (insert) as the basic subalgorithm.

```
procedure mergesort (T[1..n])
  if n is sufficiently small then insert (T)
  else
    array U[1.. 1+⌊n/2⌋], V[1.. 1+⌈n/2⌉]
    U[1.. ⌊n/2⌋] ← T[1.. ⌊n/2⌋]
    V[1.. ⌈n/2⌉] ← T[1+⌊n/2⌋.. n]
    mergesort (U[1.. ⌊n/2⌋])
    mergesort (V[1.. ⌈n/2⌉])
    merge (U, V, T)
```

Let's look at the following example -

1. array to be sorted 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9
2. split into 2 halves

3	1	4	1	5	9

2	6	5	3	5	8	9

3. One recursive call on mergesort for each half

1	1	2	3	3	4

5	5	5	6	8	9	9

4- One call on merge

1	1	2	3	3	4	5	5	5	6	8	9	9

5- The array is now sorted.

This sorting algorithm illustrates well all facets of DVC. When the number of elements to be sorted is small, a relatively simple algorithm is used. On the other hand, when this is justified by the number of elements, "mergesort" separates the instance into 2 subinstances half the size, solves each of these recursively, and then combines the 2 sorted half-arrays to obtain the solution to the original instance.

Let $t(n)$ the time taken by this algorithm to sort n elements. Separating T into U and V takes linear time. It easy to see that merge takes linear. Consequently:

$$t(n) = t(\lfloor \tfrac{n}{2} \rfloor) + t(\lceil \tfrac{n}{2} \rceil) + g(n)$$

where $g(n) \in \Theta(n)$. When n is even this recurrences becomes

$$t(n) = 2t(\tfrac{n}{2}) + g(n)$$

which yields:

$$t(n) \in \Theta(n \log n)$$

Thus efficiency is similar to that of heapsort.

Merge sorting may be slightly faster in practice, but requires significantly more storage for the intermediate arrays U and V. Recall that heapsort can sort in place, in the sense that it needs only a small constant number of working variables.

7.4.2 Quicksort:

invented by Hoare, and based on DVC principle. Unlike mergesort, most of the nonrecursive part of the work to be done is spent constructing the subsinstances rather than combining their solutions.

- First, this algorithm chooses a pivot one of the items in the array to be sorted. The array is then partitioned on either side of the pivot, greater elements to the right, smaller to the left. If now the sections of the array on either side of the pivot are sorted independently by recursive calls of the algorithm, the final result is a completely sorted array, no subsequent merge steps being necessary. To balance the sizes of the 2 subsinstances to be sorted, we would like to use the median element as the pivot. Unfortunately, finding the median takes more time than it is worth. For this reason, we use arbitrary element of the array as the pivot, hoping for the best.

It is crucial in practice, when designing a linear pivoting algorithm that the hidden constant be small if quicksort is to be competitive with others such as heapsort.

Suppose $T[i..j]$ is to be pivoted around $p = T[i]$. One good way of pivoting consists of scanning the subarray just once, but starting at <u>both</u> ends. Pointers k and l are initialized at i and $j+1$, respectively. Pointer k is then incremented until $T[k] > p$ and pointer l is decremented until $T[l] \leq p$. Now $T[k]$ and $T[l]$ are interchanged. This process continues as long as $k < l$. Finally, $T[i]$ and $T[l]$ are interchanged to put the pivot in its correct position.

The following procedure shows how to permute the elements in array $T[i..j]$ and returns a value l such that at the end; $i \leq l \leq j$; $T[k] \leq p$ for all $i \leq k \leq l$, $T[l] = p$, and $T[k] > p$ for all $l < k \leq j$, where p is the initial value of $T[i]$ }

```
procedure pivot (T[i..j]; var l)
    p ← T[i]
    k ← i ; l ← j+1
    repeat k ← k+1 until T[k] > p or k ≥ j
    repeat l ← l-1 until T[l] ≤ p
    while k < l do
        swap T[k] and T[l]
        repeat k ← k+1 until T[k] > p
        repeat l ← l-1 until T[l] ≤ p
    swap T[i] and T[l]
```

Now here is the sorting algorithm. To sort the entire array T, simply call quicksort $(T[1..n])$.

```
procedure quicksort (T[i..j])
    {sorts subarray T[i..j] into nondecreasing order}
    if j-i is sufficiently small then insert (T[i..j])
    else
        pivot (T[i..j], l)
        quicksort (T[i..l-1])
        quicksort (T[l+1..j])
```

Example on how pivot and quicksort work.

1. array to be sorted

3	1	4	1	5	9	2	6	5	3	5	8	9

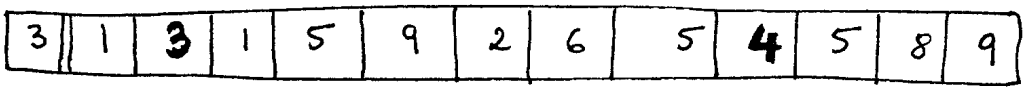
2. Array pivoted about its first element $p = 3$

3	1	4	1	5	9	2	6	5	3	5	8	9

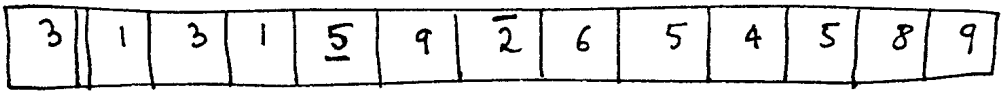
3. Find 1st element larger than pivot (underlined) and last element not larger than pivot (overline)

3	1	$\underline{4}$	1	5	9	2	6	5	$\overline{3}$	5	8	9

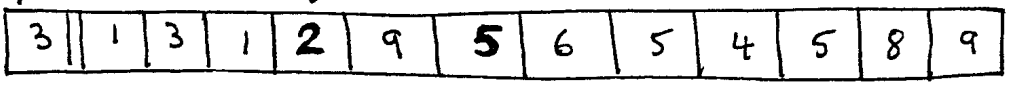
4. Swap those elements (4 and 3).



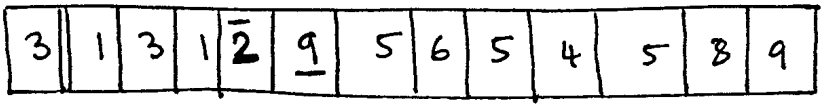
5. Scan again in both directions



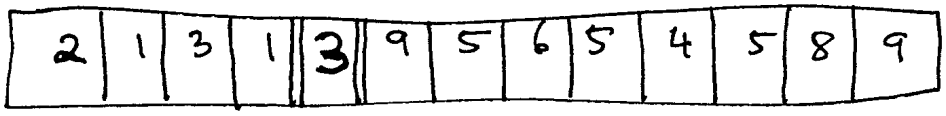
6- Swap (5 and 2)



7- Scan



8- Pointers have crossed (overline on left of underline) swap pivot with overlined -



9- Pivoting is now complete - Recursively sort subarrays on each side of pivot.

1, 1, 2, 3, [3] 4, 5, 5, 5, 6, 8, 9, 9

10- The array is now sorted.

Quicksort is inefficient if it happens systematically on most recursive calls that the subinstances $T[i..l-1]$ and $T[l+1..j]$ are severely unbalanced. In the worst case, for example if T is already sorted before the call to quicksort, we get $l = i$ each time, which means a recursive call executes in an instance of size 0 and another one instance whose size is reduced by only 1. The time is quadratic and in $\Omega(n^2)$ in the worst case to sort n elements.

Quicksort takes a time in $O(n \log n)$ to sort n elements on the average. In practice the hidden constant is smaller than those involved in heapsort or in mergesort.

7.5 Finding the Median:

Given $T[1..n]$ an array of integers, let s be an integer between 1 and n. The s-th smallest element of T is defined as the element that would be in the s^{th} position if T were sorted into nondecreasing order. Given T and s, the problem of finding the s-smallest element of T is known as the selection problem. When $s = \lceil \frac{n}{2} \rceil$ it is known as the median.

Exple: the median of $[3, 1, 4, 1, 5, 9, 2, 6, 5]$ is 4 since $3, 1, 1, 2$ are smaller than 4 and $5, 6, 9, 5$ are larger.

The naive way to find the median would be to sort the array then to extract its $\lceil \frac{n}{2} \rceil$-th entry, if we used mergesort or heapsort, this takes a time in $\Theta(n \log n)$. Can we do better?

To answer the question, we study the interrelation between finding the median and selecting the s-th smallest element.

Assume the availability of an algorithm "median($T[1..n]$)" that returns median of T. Given a T and s (integer), how could we determine the s^{th} smallest (s) of T? Let p be the median of T. Now pivot T around p using the new pivot procedure that we call "pivotbis" -

procedure pivotbis ($T[i..j]$, p; var k, l)

that partitions T into 3 sections using p as a pivot: after pivoting, the elements in $T[1..k]$ are smaller than p, those in $T[k+1..l-1]$ are equal to p, and those in $T[l..j]$ are larger than p. The values of "k" and "l" are returned by pivotbis. After a call on pivotbis (T, p, k, l) we are done if $k < s < l$, as s is simply equal to p. If $s \leq k$, the s-th smallest element of T is the s-smallest of $T[1..k]$. Finally, if $s \geq l$, the s-smallest element of T is now the $(s-l+1)$st smallest element of $T[1..n]$. In any case, we have made progress

Since either we are done, or the sub array to be considered contains less than half the elements, by virtue of p being the median of the original array. (This approach is very similar to the binary search

The key idea is to use 2 variables i and j, initialized to 1 and n respectively, and to ensure that at every moment $i \leq s \leq j$, and the elements in $T[1..i-1]$ are smaller than those in $T[i..j]$, which are in turn smaller than those in $T[j+1..n]$. The immediate consequence of this is that the desired element resides in $T[i..j]$. When all the elements in $T[i..j]$ are equal, we are done

• The following example illustrates the process. For simplicity we assume "pivotbis" is implemented in a way that is intuitively simple eventhough a really efficient implementation would proceed differently

Exple :

1 • Given the following array, find the 4th smallest element.

3	1	4	1	5	9	2	6	5	3	5	8	9

2 • Pivot array around its median p = 5 using "pivotbis"

3	1	4	1	2	3	5	5	5	9	6	8	9

3 • only part left of pivot is still relevant since $4 \leq 6$

3	1	4	1	2	3	·	·	·	·	·	·	·

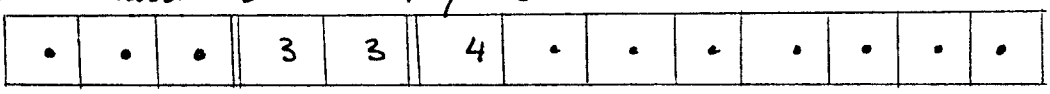
4 • pivot that part around its median p = 2

1	1	2	3	4	3	·	·	·	·	·	·	·

5 • Only the part right ~~of~~ pivot is still relevant since $4 \geq 4$

·	·	·	3	4	3	·	·	·	·	·	·	·

6. Pivot around its median $p = 3$



7. Answer is 3 because the pivot is the 4th position.

```
function selection (T[1..n], s)
  {finds the s-smallest element in T, 1 ≤ s ≤ n}
  i ← 1 ; j ← n
  repeat
    {Answer lies in T[i..j]}
    p ← median (T[i..j])
    pivotbis (T[i..j], p, k, l)
    if s ≤ k then j ← k
    else if s ≥ l then i ← l
      else return p
```

The above algorithm select the required elements of T after going round the loop (repeat loop) a logarithmic number of time in the worst case. However trips round the loop no longer take constant time, and indeed this algorithm cannot be used until we have an efficient way to find the median, which was our original problem. Can we modify the algorithm to avoid resort to the median?

First, observe that our algorithm still works regardless of which element of T is chosen as pivot (the value of p). The choice of p affects only the efficiency of the algorithm: The median assures that only half the elements are considered each time round the loops. If we are willing to sacrifice speed in the worst case and obtain a reasonably fast algorithm on average we simply choose T[i] as pivot. So first instruction in the loop $p \leftarrow T[i]$.
This causes the algorithm to spend quadratic time in the worst case

(For example: array in decreasing order and search for smallest element.) Nevertheless, this modified algorithm run in linear time on the average, under the assumption that elements of T are distinct and that each of the n! possible permutations of the element are equally likely. This is much better than sorting the array on the average but the worst-case behavior is unacceptable for many applications. The quadratic worst case can be avoided by choosing other techniques that would choose the pivot reasonably closer to the median, that is refered to as the pseudomedian.

7.6 Matrix Multiplication

Let A and B be two n x n matrices to be multiplied, and A x B = C. where:

$$c_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Assuming that scalar multiplication, and addition are elementary operations. Since there are n additions and n multiplications each entry in C is calculated in $\theta(n)$. There are n^2 entries, thus the product AB can be calculated in $\theta(n^3)$.

In the late 1960's Strassen improved this algorithm with an idea similar to that of "multiplying large numbers" which was discovered a decade earlier.

Basic idea: Strassen showed that 2 x 2 matrices can be multiplied using less than the eight scalar multiplications, required by the original definition. Let A and B to be multiplied.

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

Consider the following operations, each of which involves one multiplication.

- $m_1 = (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11})$
- $m_2 = a_{11} b_{11}$
- $m_3 = a_{12} b_{21}$
- $m_4 = (a_{11} - a_{21})(b_{22} - b_{12})$
- $m_5 = (a_{21} + a_{22})(b_{12} - b_{11})$
- $m_6 = (a_{12} - a_{21} + a_{11} - a_{22}) b_{22}$
- $m_7 = a_{22}(b_{11} + b_{22} - b_{12} - b_{21})$

and AB is given by the following matrix

$$C = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}$$

It is therefore possible to multiply two 2×2 matrices using only seven scalar multiplications. Like multiplying large numbers, the algorithm does not very interesting: it uses a large number of additions and subtractions compared to the four additions that are sufficient for the classic algorithm.

If we replace each entry of A and B by an $n \times n$ matrix, we obtain an algorithm that can multiply two $2n \times 2n$ matrices by carrying out seven multiplications of $n \times n$ matrices, as well as a number of additions and subtractions of $n \times n$ matrices. Given that large matrices can be added much faster than they can be multiplied, saving one multiplication more than compensates for the supplementary additions.

Let $t(n)$ be the time to multiply two $n \times n$ matrices using the changes above. For simplicity assume is a power of 2. Since matrices can be added and subtracted in a time in $\Theta(n^2)$ then

$$t(n) = 7\, t\left(\frac{n}{2}\right) + g(n) \quad , \quad \text{where } g(n) \in \Theta(n^2)$$

whose solution is $t(n) \in \Theta(n^{\log_2 7})$, since $\log_2 7 < 2.81$ it is thus possible to multiply two $n \times n$ matrices in a time $O(n^{2.81})$ provided scalar operations are elementary. Square matrices whose sizes are not power of 2

are easily handled by padding them with rows and columns of zeros, at least doubling their size, which does not affect the asymptotic running time.