Solving Divide-and-Conquer Recurrences

Victor Adamchik

A divide-and-conquer algorithm consists of three steps:

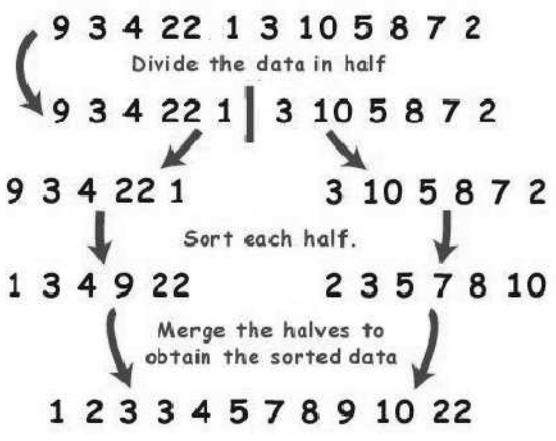
- dividing a problem into smaller subproblems
- solving (recursively) each subproblem
- then combining solutions to subproblems to get solution to original problem

We use recurrences to analyze the running time of such algorithms. Suppose T_n is the number of steps in the worst case needed to solve the problem of size n. Let us split a problem into $a \geq 1$ subproblems, each of which is of the input size $\frac{n}{b}$where $b > 1$. Observe, that the number of subproblems a is not necessarily equal to b. The total number of steps T_n is obtained by all steps needed to solve smaller subproblems $T_{n/b}$ plus the number needed to combine solutions into a final one. The following equation is called divide-and-conquer recurrence relation

$$T_n = a\, T_{n/b} + f(n)$$

As an example, consider the mergesort:

-divide the input in half
-recursively sort the two halves
-combine the two sorted subsequences by merging them.



Let $T(n)$ be worst-case runtime on a sequence of n keys:

If $n = 1$, then $T(n) = \Theta(1)$ constant time

If $n > 1$, then $T(n) = 2\, T(n/2) + \Theta(n)$

here Θ(n) is time to do the merge. Then

$$T_n = 2\,T_{n/2} + \Theta(n)$$

Other examples of divide and conquer algorithms: quicksort, integer multiplication, matrix multiplication, fast Fourier trnsform, finding conver hull and more.

There are several techniques of solving such recurrence equations:

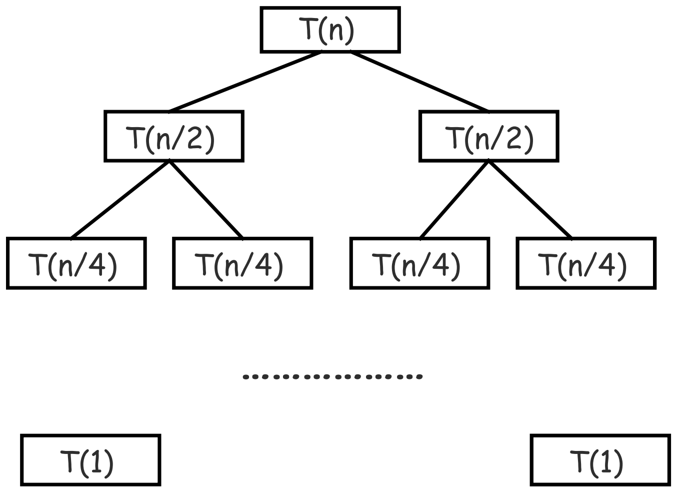
- the iteration method
- the tree method
- the master-theorem method
- guess-and-verify

Tree method

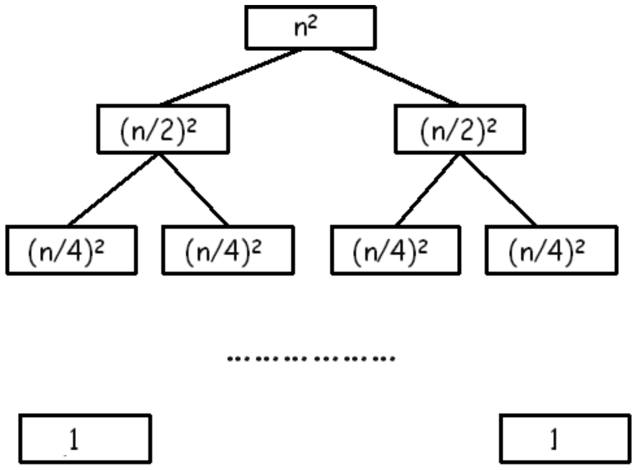
We could visualize the recursion as a tree, where each node represents a recursive call. The root is the initial call. Leaves correspond to the exit condition. We can often solve the recurrence by looking at the structure of the tree. To illustrate, we take this example

$$T(n) = 2\,T\left(\frac{n}{2}\right) + n^2$$
$$T(1) = 1$$

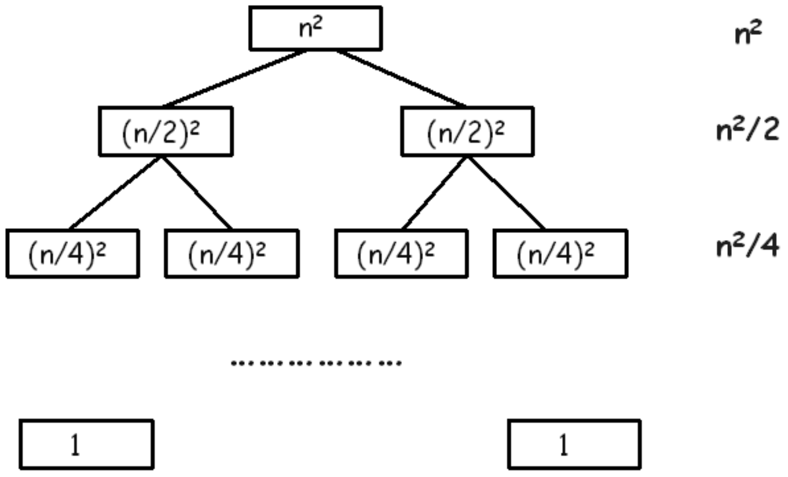
Here is a recursion tree that diagrams the recursive function calls



Using a recursion tree we can model the time of a recursive execution by writing the size of the problem in each node.



Using a recursion tree we can model the time of a recursive execution by writing the size of the problem in each node.



The last level corresponds to the initial condition of the recurrence. Since the work at each leaf is constant, the total work at all leaves is equal to the number of leaves, which is

$$2^h = 2^{\log_2 n} = n$$

To find the total time (for the whole tree), we must add up all the terms

$$T(n) = n + n^2\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots\right) = n + n^2 \sum_{k=0}^{-1+\log_2 n} \left(\frac{1}{2}\right)^k$$

The sum is easily computed by means of the geometric series

$$\sum_{k=0}^{h} x^k = \frac{x^{h+1} - 1}{x - 1}, \;\; x \neq 1$$

This yeilds

$$T(n) = 2\,n^2 - 2\,n + n \;=\; 2\,n^2 - n$$

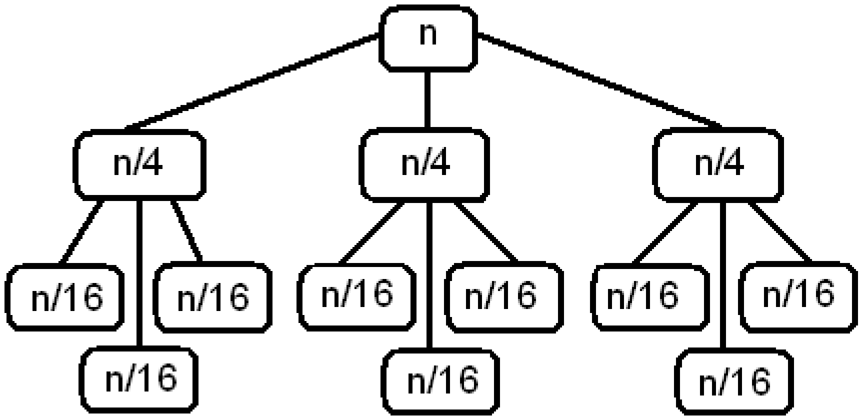
Check with *Mathematica*

RSolve[{T[n] == 2 T[n / 2] + n², T[1] == 1}, T[n], n]

```
{{T[n] → n (-1 + 2 n)}}
```

Example. Solve the recurrence

$$T(n) = 3\,T\left(\frac{n}{4}\right) + n$$



The work at all levels is

$$n\left(1 + \frac{3}{4} + \frac{9}{16} + \ldots\right)$$

Since the height is $\log_4 n$, the tree has $3^{\log_4 n}$ leaves. Hence, the total work is given by

$$T(n) = n \sum_{k=0}^{-1+\log_4 n} \left(\frac{3}{4}\right)^k + \,3^{\log_4 n}\, T(1)$$

By means of the geometric series and taking into account

$$3^{\log_4 n} = n^{\log_4 3}$$

the above sum yields

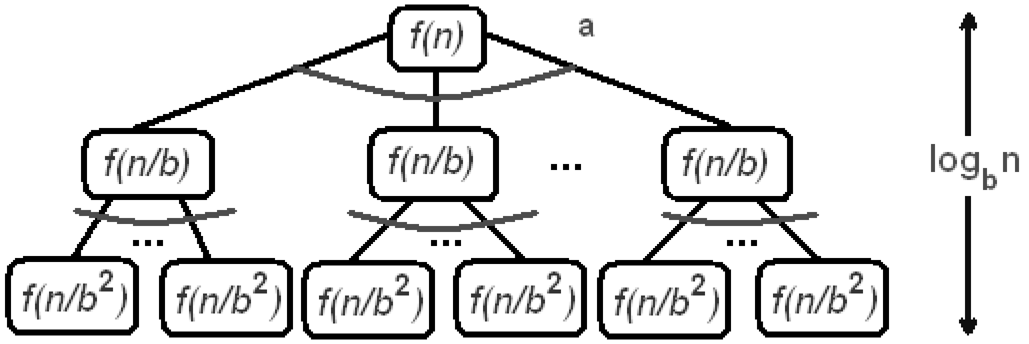
$$T(n) = 4\,n\; - 4\,n^{\log_4 3} + n^{\log_4 3}\, T(1) \;=\; O(n)$$

The Master Theorem

The master theorem solves recurrences of the form

$$T(n) \;= a\,T\left(\frac{n}{b}\right) + f(n)$$

for a wide variety of function $f(n)$ and $a \geq 1$, $b > 1$. In this section we will outline the main idea. Here is the recursive tree for the above equation



It is easy to see that the tree has $a^{\log_b n}$ leaves. Indeed, since the height is $\log_b n$, and the tree branching factor is a, the number of leaves is

$$a^h = a^{\log_b n} = a^{\frac{\log_a n}{\log_a b}} = n^{\frac{1}{\log_a b}} = n^{\log_b a}$$

Summing up values at each level, gives

$$T(n) = f(n) + a\, f\left(\frac{n}{b}\right) + a^2\, f\left(\frac{n}{b^2}\right) + \ldots + n^{\log_b a}\, T(1)$$

Therefore, the solution is

$$T(n) = n^{\log_b a}\, T(1) + \sum_{k=0}^{-1+\log_b n} a^k\, f\left(\frac{n}{b^k}\right)$$

Now we need to compare the asymptotic behavior of $f(n)$ with $n^{\log_b a}$. There are three possible cases.

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & \text{if } f(n) = O\left(n^{\log_b a}\right) \\ \Theta\left(n^{\log_b n} \log^{k+1} n\right) & \text{if } f(n) = \Theta\left(n^{\log_b a} \log^k n\right),\ k \ge 0 \\ \Theta(f(n)) & \text{if } f(n) = \Omega\left(n^{\log_b a}\right) \end{cases}$$

The following examples demonstrate the theorem.

Case 1. $T(n) = 4\,T\left(\frac{n}{2}\right) + n$

We have $f(n) = n$ and $n^{\log_b a} = n^{\log_2 4} = n^2$, therefore $f(n) = O\left(n^2\right)$. Then the solution is $T(n) = \Theta\left(n^2\right)$ by case 1.

Case 2. $T(n) = 4\,T\left(\frac{n}{2}\right) + n^2$

In this case $f(n) = n^2$ and $f(n) = \Theta\left(n^2\right)$. Then $T(n) = \Theta\left(n^2 \log n\right)$ by case 2.

Case 3. $T(n) = 4\,T\left(\frac{n}{2}\right) + n^3$

In this case $f(n) = n^3$ and $f(n) = \Omega\left(n^{\log_b a}\right) = \Omega\left(n^2\right)$. Then $T(n) = \Theta\left(n^3\right)$ by case 3.

Karatsuba Algorithm

Multiplication of large integers

The brute force approach ("grammar school" method)

```
    1 2 3
      4 5
   ------
    6 1 5
  4 9 2
---------
  5 5 3 5
```

We say that multiplication of two *n*-digits integers has time complexity at worst $O(n^2)$.

We develop an algorithm that has better asymptotic complexity. The idea is based on divide-and-conquer technique.

Consider the above integers and split each of them in two parts

```
123 = 12 * 10 + 3

 45 =  4 * 10 + 5
```

and then multiply them:

```
123*45 = (12*10 + 3)(4*10 + 5) =
```

$$12*4*10^2 + (12*5 + 4*3)*10 + 3*5$$

In general, the integer which has *n* digits can be represented as

$$\text{num} = x * 10^m + y$$

where

$$m = \text{floor}\left(\frac{n}{2}\right)$$

$$x = \text{ceiling}\left(\frac{n}{2}\right)$$

$$y = \text{floor}\left(\frac{n}{2}\right)$$

Example,

$$154\,517\,766 = 15\,451 * 10^4 + 7766$$

Consider two *n*-digits numbers

$$\text{num}_1 = x_1 * 10^p + x_0$$

$$\text{num}_2 = y_1 * 10^p + y_0$$

Their product is

$$\text{num}_1 * \text{num}_2 = x_1 * y_1 * 10^{2p} + (x_1 * y_0 + x_0 * y_1) * 10^p + x_0 * y_0$$

Just looking at this general formula you can say that just instead of one multiplication we have 4.

Where is the advantage?

numbers x_1, x_0 and y_1, y_0 have twice less digits.

The worst-case complexity

Let $T(n)$ denote the number of digit multiplications needed to multiply two n-digits numbers.

The recurrence (since the algorithm does 4 multiplications on each step)

$$T(n) = 4\,T\left(\frac{n}{2}\right) + O(n), \; T(c) = 1$$

Note, we ignore multiplications by a base!!! Its solution is given by

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

The algorithm is still quadratic!

The Karatsuba Algorithm

1962, Anatolii Karatsuba, Russia.

$$\text{num}_1 * \text{num}_2 = x_1 * y_1 * 10^{2p} + (x_1 * y_0 + x_0 * y_1) * 10^p + x_0 * y_0$$

The goal is to decrease the number of multiplications from 4 to 3.
We can do this by observing that

$$(x_1 + x_0) * (y_1 + y_0) = x_1 * y_1 + x_0 * y_0 + (x_1 * y_0 + x_0 * y_1)$$

It follows that

$$\text{num}_1 * \text{num}_2 = x_1 * y_1 * 10^{2p} + \Big((x_1 + x_0) * (y_1 + y_0) - x_1 * y_1 - x_0 * y_0\Big) * 10^p + x_0 * y_0$$

and it is only 3 multiplications (see it ?).

The total number of multiplications is given by (we ignore multiplications by a base)

$$T(n) = 3\,T\left(\frac{n}{2}\right) + O(n), \quad \text{T(c)} = 1$$

Its solution is

$$T(n) = 3^{\log_2 n} = n^{\log_2 3} = n^{1.58...}$$

■ Toom-Cook 3-Way Multiplication

1963, A. L. Toom, Russia.
1966, Cook, Harvard, Ph.D Thesis

The key idea of the algorithm is to divide a large integer into 3 parts (rather than 2) of size approximately $n/3$ and then multiply those parts.

Here is the equation of for the total number of multiplications

$$T(n) = 9\,T\left(\frac{n}{3}\right) + O(n), \qquad T(c) = 1$$

and the solution

$$T(n) = 9^{\log_3 n} = n^2$$

Let us reduce the number of multiplications by one

$$T(n) = 8\,T\left(\frac{n}{3}\right) + O(n)$$

$$T(n) = 8^{\log_3 n} = n^{\log_3 8} = n^{1.89...}$$

No advantage. This does not improve the previous algorithm, that runs at $O(n^{1.58...})$
How many multiplication should we eliminate?

Let us consider that equation in a general form, where parameter $p > 0$ is arbitrary

$$T(n) = p\,T\left(\frac{n}{3}\right) + O(n)$$

$$T(n) = p^{\log_3 n} = n^{\log_3 p}$$

Therefore, the new algoritnm will be faster than $O(n^{1.58})$ if we reduce the number of multi-plications to five

$$T(n) = 5^{\log_3 n} = n^{\log_3 5} = n^{1.47...}$$

This is an improvement over Karatsuba.

Is it possible to reduce a number of multiplications to 5?

Yes, it follows from this system of equations:

$$\begin{aligned}
&x_0\,y_0 = Z_0\\
&12\,(x_1\,y_0 + x_0\,y_1) = 8\,Z_1 - Z_2 - 8\,Z_3 + Z_4\\
&24\,(x_2\,y_0 + x_1\,y_1 + x_0\,y_2) = -30\,Z_0 + 16\,Z_1 - Z_2 + 16\,Z_3 - Z_4\\
&12\,(x_2\,y_1 + x_1\,y_2) = -2\,Z_1 + Z_2 + 2\,Z_3 - Z_4\\
&24\,x_2\,y_2 = 6\,Z_0 - 4\,Z_1 + Z_2 - 4\,Z_3 + Z_4
\end{aligned}$$

where

$$\begin{aligned}
Z_0 &= x_0\, y_0\\
Z_1 &= (x_0 + x_1 + x_2)\,(y_0 + y_1 + y_2)\\
Z_2 &= (x_0 + 2\,x_1 + 4\,x_2)\,(y_0 + 2\,y_1 + 4\,y_2)\\
Z_3 &= (x_0 - x_1 + x_2)\,(y_0 - y_1 + y_2)\\
Z_4 &= (x_0 - 2\,x_1 + 4\,x_2)\,(y_0 - 2\,y_1 + 4\,y_2)
\end{aligned}$$

■ Further Generalization

It is possible to develop a **faster** algorithm by increasing the number of splits.

Let us consider a 4-way splitting. How many multiplications should we have on each step so this algorithm will outperform the 3-way splitting?

$$T(n) = p\,T\left(\frac{n}{4}\right) + O(n)$$

$$T(n) = p^{\log_4 n} = n^{\log_4 p}$$

We find parameter p from

$$\log_4 p < \log_3 5$$

which yields

$$p = 7$$

The following table demonstrates a relationship between splits and the number of multiplications:

split	number of *
2	3
3	5
4	7

Intuitively we see that the ***k***-way split requires $\mathbf{2\,k - 1}$ multiplications.

This means that instead of k^2 multiplications we do only $2\,k - 1$.

The recurrence equation for the total number of multiplication is given by

$$T(n) = (2\,k - 1)\,T\left(\frac{n}{k}\right) + O(n)$$

and its solution is

$$T(n) = (2\,k - 1)^{\log_k n} = n^{\log_k (2\,k - 1)}$$

Here is the sequence of the k-way splits when k runs from 2 to 10:

$$n^{1.58},\ n^{1.46},\ n^{1.40},\ n^{1.36},\ n^{1.33},\ n^{1.31},\ n^{1.30},\ n^{1.28},\ n^{1.27}\ \ldots$$

We can prove that asymptotically multiplication of two n-digits numbers requires $O\left(n^{1+\epsilon}\right)$ multiplications, where $\epsilon \to 0$.

Note, we will NEVER get a linear performance (prove this!)

Is it always possible to find such $2\,k-1$ multiplications?

Consider two polynomials of $k-1$ degree

$$\text{polyn}_1 = a_{k-1}\,x^{k-1} + a_{k-2} * x^{k-2} + \ldots + a_1 * x + a_0$$

$$\text{polyn}_2 = b_{k-1}\,x^{k-1} + b_{k-2} * x^{k-2} + \ldots + b_1 * x + b_0$$

when we multiply them we get a polynomial of $2\,k-2$ degree

$$\text{polyn}_1 * \text{polyn}_2 = a_{k-1}\,b_{k-1} * x^{2\,k-2}\ +\ \ldots\ + (a_1\,b_0 + b_1\,a_0) * x\ +\ a_0\,b_0$$

The above polynomial has exactly $2\,k-1$ coefficients, therefore it's uniquely defined by $2\,k-1$ values.