Complexity of recursive algorithms

Vera Sacristán

Computational Geometry
Departament de Matemàtica Aplicada II
Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

Computational Geometry makes frequent use of recursive algorithms. The analysis of their complexity is not as immediate as it is for exhaustive or incremental algorithms, for example. The goal of this document is to show in some detail the complexity analysis of some of the most classical recursive algorithms in Computational Geometry, as well as to remind the general theorem which applies to this kind of recursions.

1 Some examples from Computational Geometry

Proposition 1 *The divide-and-conquer algorithm to compute the convex hull of a set of n points in the plane, runs in $O(n \log n)$ time.*

In this algorithm, the problem gets divided into two subproblems of approximately size $n/2$, and the partition and merging steps are done in $O(n)$ time. Therefore, the running time of the algorithm is:

$$\begin{aligned} T(n) &= 2\,T(\tfrac{n}{2}) + O(n) \\ &\leq 2\,T(\tfrac{n}{2}) + c\,n \\ &\leq 4\,T(\tfrac{n}{4}) + 2\,c\,\tfrac{n}{2} + c\,n \\ &\leq 8\,T(\tfrac{n}{8}) + 4\,c\,\tfrac{n}{4} + 2\,c\,\tfrac{n}{2} + c\,n \\ &\vdots \\ &\leq 2^k\,T(\tfrac{n}{2^k}) + c\,n\,k \\ &\vdots \\ &\leq 2^{\log n}\,T(1) + c\,n \log n \\ &= n + c\,n \log n \\ &= O(n \log n) \end{aligned}$$

Observation: These calculations implicitly assume that n is a power of 2. The complete and rigorous proof of Proposition 1, valid for all n, would require some more details.

Proposition 2 *The binary search algorithm to compute the supporting (half)lines of a convex n-gon from an exterior point, runs in $O(\log n)$ time.*

In this algorithm, half of the polygon vertices are eliminated at each iteration, and deciding which half is to be rejected can be done in constant time, since the decision is based on local information.

Therefore, the running time of the algorithm is:

$$\begin{aligned}
T(n) &= T(\tfrac{n}{2}) + O(1) \\
&\leq T(\tfrac{n}{2}) + c \\
&\leq T(\tfrac{n}{4}) + 2c \\
&\leq T(\tfrac{n}{8}) + 3c \\
&\vdots \\
&\leq T(\tfrac{n}{2^k}) + c\,k \\
&\vdots \\
&\leq T(1) + c \log n \\
&= O(\log n)
\end{aligned}$$

Observation: The comment made in Proposition 1 applies to this case too.

Proposition 3 *The prune-and-search algorithm proposed by Megiddo and Dyer to solve linear programming and related problems runs in $O(n)$ time.*

At each iteration of this algorithm, the prune step eliminates a constant fraction of the input elements (halfplanes, in the linear programming problem), based on a previous search phase which runs in linear time. Therefore, the running time of the algorithm is:

$$\begin{aligned}
T(n) &= T(r\,n) + O(n), \qquad \text{where } 0 < r < 1, \\
&\leq T(r\,n) + c\,n \\
&\leq T(r^2\,n) + c\,r\,n + c\,n \\
&\leq T(r^3\,n) + c\,r^2\,n + c\,r\,n + c\,n \\
&\leq T(r^4\,n) + c\,r^3\,n + c\,r^2\,n + c\,r\,n + c\,n \\
&\vdots \\
&\leq T(r^k\,n) + c\,n \textstyle\sum_{i=0}^{k-1} r^i \\
&= T(r^k\,n) + c\,n \tfrac{1-r^k}{1-r} \\
&\leq T(r^k\,n) + c\,n \tfrac{1}{1-r} \\
&\vdots \\
&= T(1) + c'n \\
&= O(n)
\end{aligned}$$

2 The General Theorem

There is a result which allows to solve this kind of recursions in a systematic way. It is called the *master theorem.*

Theorem 4 *Let T be a increasing function between natural numbers, recursively defined by the formula*

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

where $a \geq 1$, $b > 1$. Then:

1. *If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.*
2. *If $f(n) = \Theta(n^{\log_b a} \log^k n)$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.*
3. *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and $af(\frac{n}{b}) \leq cf(n)$ for some $c < 1$ and all $n \geq n_0$, then $T(n) = \Theta(f(n))$.*

Observation: The function T models the running time of a recursive algorithm where n is the size of the problem, a is the number of subproblems in the recursion, $\frac{n}{b}$ is the size of each subproblem and f is the cost of the work that the algorithm does outside the recursive calls.

The idea to proof is as follows. The problem of size n is split into a problems of size $\frac{n}{b}$, in $f(n)$ time. At their turn, these a problems are split into a^2 problems of size $\frac{n}{b^2}$, in $af(\frac{n}{b})$ time. The a^2 resulting problems are split into a^3 problems of size $\frac{n}{b^3}$, in $a^2 f(\frac{n}{b^2})$ time. And so on. This process can be represented in an a-ary tree (each node of the tree has a children) where each node gets labeled with the the running time of the work that the algorithm does at that point, outside the recursion. Since at each node the size of the problem gets divided by b, the height of the tree is $log_b n$. Therefore:

$$\begin{aligned}
T(n) &= aT(\tfrac{n}{b}) + f(n) \\
&= a^2 T(\tfrac{n}{b^2}) + f(n) + af(\tfrac{n}{b}) \\
&= a^3 T(\tfrac{n}{b^3}) + f(n) + af(\tfrac{n}{b}) + a^2 f(\tfrac{n}{b^2}) \\
&\vdots \\
&= a^k T(\tfrac{n}{b^k}) + \textstyle\sum_{i=0}^{k-1} a^i f(\tfrac{n}{b^i}) \\
&\vdots \\
&= a^{\log_b n} T(\tfrac{n}{b^{\log_b n}}) + \textstyle\sum_{i=0}^{\log_b n - 1} a^i f(\tfrac{n}{b^i}) \\
&= a^{\log_b n} T(1) + \textstyle\sum_{i=0}^{\log_b n - 1} a^i f(\tfrac{n}{b^i}) \\
&= a^{\log_b n} f(1) + \textstyle\sum_{i=0}^{\log_b n - 1} a^i f(\tfrac{n}{b^i}) \\
&= \textstyle\sum_{i=0}^{\log_b n} a^i f(\tfrac{n}{b^i}).
\end{aligned}$$

Separating the first and last terms of the last expression, helps understanding the theorem:

$$T(n) = a^{\log_b n} + \sum_{i=1}^{\log_b n - 1} a^i f(\frac{n}{b^i}) + f(n).$$

Notice, in fact, that the first term is equal to $n^{\log_b a}$.

The three cases of the theorem correspond to the following three possibilities:

1. If $f(n) = O(n^{\log_b a - \epsilon})$, the order of the third term is substantially lower than the order of the first term. Therefore, the order of $T(n)$ is the same as that of the first term. This corresponds to the case in which the most time-consuming work of the algorithm is done in the leaves of the tree.

2. If $f(n) = \Theta(n^{\log_b a})$, the first and the third terms are of the same order. Therefore, the second term simply adds a $\log n$ factor to the cost of $T(n)$. This corresponds to the case in which at each level of the tree, the algorithm spends the same running time.

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$, the order of the third term is substantially higher than the order of the first term. Therefore, if f is regular enough, the order of $T(n)$ is the same as that of the third term. This corresponds to the case in which the most time-consuming work of the algorithm is done in the root of the tree.

Let us see why:

1. If $f(n) = O(n^{\log_b a - \epsilon})$, then

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n} a^i f(\frac{n}{b^i}) = O\left(\sum_{i=0}^{\log_b n} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}\right) = O\left(n^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^i\right) \\ &= O\left(n^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n} (b^\epsilon)^i\right) = O\left(n^{\log_b a - \epsilon} \frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1}\right) = O\left(n^{\log_b a - \epsilon} \frac{n^\epsilon - 1}{b^\epsilon - 1}\right) \\ &= O\left(n^{\log_b a - \epsilon} n^\epsilon\right) = O\left(n^{\log_b a}\right). \end{aligned}$$

   On the other hand, the lower bound is given by the first term of the sum:

$$T(n) = n^{\log_b a} + \sum_{i=1}^{\log_b n - 1} a^i f(\frac{n}{b^i}) + f(n) = \Omega\left(n^{\log_b a}\right).$$

2. If $f(n) = \Theta(n^{\log_b a})$, then

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n} a^i f(\frac{n}{b^i}) = \Theta\left(\sum_{i=0}^{\log_b n} a^i \left(\frac{n}{b^i}\right)^{\log_b a}\right) = \Theta\left(\sum_{i=0}^{\log_b n} a^i \frac{n^{\log_b a}}{b^{(\log_b a) i}}\right) \\ &= \Theta\left(\sum_{i=0}^{\log_b n} n^{\log_b a}\right) = \Theta\left(n^{\log_b a} \sum_{i=0}^{\log_b n} 1\right) = \Theta\left(n^{\log_b a} \log n\right). \end{aligned}$$

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$, then

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n} a^i f(\frac{n}{b^i}) = \Omega\left(\sum_{i=0}^{\log_b n} a^i \left(\frac{n}{b^i}\right)^{\log_b a + \epsilon}\right) = \Omega\left(n^{\log_b a + \epsilon} \left(\frac{a}{b^{(\log_b a + \epsilon)}}\right)^i\right) \\ &= \Omega\left(n^{\log_b a + \epsilon} \left(\frac{1}{b^\epsilon}\right)^i\right) = \Omega\left(n^{\log_b a + \epsilon}\right) = \Omega(f(n)). \end{aligned}$$

If, in addition, $af(\frac{n}{b}) \leq cf(n)$ for some $c < 1$ and all $n \geq n_0$, then $a^i f(\frac{n}{b^i}) \leq c^i f(n)$ and then:

$$T(n) = \sum_{i=0}^{\log_b n} a^i f(\frac{n}{b^i}) \leq \sum_{i=0}^{\log_b n} c^i f(n) = f(n) \sum_{i=0}^{\log_b n} c^i \leq f(n) \sum_{i=0}^{\infty} c^i = f(n) \frac{1}{1-c} = O(f(n)).$$

Observation: What we have developed so far is not a formal proof of the theorem, but only its approximate scheme, since in case 2 we have not studied what happens when $k > 0$ and, in all cases, we are assuming that n is a power of b. The theorem is true even when n is not divisible by b, but in such case the formula must be stated in terms of the floor of the involved numbers. A complete proof can be found in the following book: T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, The MIT Press - McGraw Hill Book Company, 1990 (2nd ed. 2001).

3 Applying the theorem

In the following, some examples are shown of how the theorem applies.

Example 1 The recursion of Proposition 1 is $T(n) = 2\,T(\frac{n}{2}) + O(n)$. It corresponds to the case where $a = b = 2$, therefore $\log_b a = 1$ and $f(n) = O(n) = O(n^{\log_b a} \log^k n)$ with $k = 0$. According to the master theorem, $T(n) = O(n^{\log_b a} \log^{k+1} n) = O(n \log n)$, which coincides with the result obtained in Proposition 1.

Example 2 The recursion of Proposition 2 is $T(n) = T(\frac{n}{2}) + O(1)$. It corresponds to the case where $a = 1$ and $b = 2$, therefore $\log_b a = 0$ and $f(n) = O(1) = O(n^{\log_b a} \log^k n)$ with $k = 0$. According to the master theorem, $T(n) = O(n^{\log_b a} \log^{k+1} n) = O(\log n)$, which coincides with the result obtained in Proposition 2.

Example 3 The recursion of Proposition 3 is $T(n) = T(r\,n) + O(n)$. It corresponds to the case where $a = 1$ and $b = \frac{1}{r}$, therefore $\log_b a = 0$ and $f(n) = O(n) = O(n^{\log_b a + \epsilon})$. According to the master theorem, $T(n) = O(f(n)) = O(n)$, which coincides with the result obtained in Proposition 3.