

## Analyzing divide-and-conquer algorithms

A divide-and-conquer algorithm has the following structure:

- **Divide** input problem (of size  $n$ ) into independent subproblems of the same type with smaller sizes (say  $n_1, \dots, n_a$ ).
- **Conquer** by solving the subproblems *recursively*.
- **Combine** solutions to subproblems to obtain solution to original problem.

For example, in MERGE-SORT the divide step is trivial, the conquer step is performed by the two recursive calls to MERGE-SORT and the combine step is the call to MERGE.

**Recurrence Relations.** Let  $T(n)$  denote the running time of a divide-and-conquer algorithm as described above. If the divide step takes  $t_D(n)$  time and the combine step takes  $t_C(n)$  time, then,

$$T(n) = t_D(n) + \sum_{1 \leq i \leq a} T(n_i) + t_C(n) \text{ if } n > 1.$$

$T(1)$  refers to the base case, which takes constant time, so  $T(1)$  is a constant. (Sometimes the base case may occur at a large value than 1, but it occurs at some constant value for  $n$ , and the recurrence relation holds for values of  $n$  larger than that constant value.)

Usually (but not always) the subproblems in the conquer step are all of the same size  $n/b$ ,  $b > 1$ . In this case, we have a simpler recurrence relation (for  $n > 1$ ):

$$T(n) = a \cdot T(n/b) + f(n), \text{ where } f(n) = t_D(n) + t_C(n)$$

For MERGE-SORT we have  $t_D(n) = 0$  and  $t_C(n) = \Theta(n)$ . Hence,

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$$

If  $n$  is a power of 2, i.e.,  $n = 2^k$  for some positive integer  $k$ , then the two subproblems have the same size and

$$T(n) = 2T(n/2) + \Theta(n) \quad \text{or}$$

$$T(n) \geq 2T(n/2) + c_1 \cdot n \quad \text{and} \quad T(n) \leq 2T(n/2) + c_2 \cdot n \quad \text{for suitable constants } c_1, c_2$$

Being able to solve recurrence relations is important when analyzing the running time of divide-and-conquer algorithms. Recurrence relations may be solved by the *iteration method* or the *substitution method*. The ‘recursion tree’ described in the textbook is a useful tool to visualize the solution of recurrence relations.

- In the **iteration method**, we repeatedly expand the recurrence relation using the same given form for the recursive terms on the right-hand side, until we reach the base case, for which we can substitute the value given for the base case.

When the iteration method is used to solve a recurrence relation, one typically obtains a summation that must be bounded. So it is important to learn to bound summations.

- In the **substitution method**, we ‘guess’ that a certain function is a solution (or an upper or lower bound on the solution) to the recurrence relation, and we verify that this is correct by induction.

The substitution method does not have the need to bound summations, but one needs to come up with a good guess to the correct solution. One also should be careful about establishing correctness by induction — one needs to re-establish the *exact* form of the inductive assumption, including all constants.

## The Master Theorem

The *Master theorem* in section 4.3 in the textbook is a useful theorem specifically tailored to solve the type of recurrence relations that arise when analyzing divide-and-conquer algorithms. It gives an asymptotically tight solution to the ‘divide and conquer recurrence’ relation for fairly general classes of functions for the divide and combine steps.

Here is a restatement of the Master Theorem from the textbook. (As is common practice, we have omitted the base case — we assume that  $T(n) \leq c$  for all  $n \leq n_0$ , for suitable constants  $c, n_0 \geq 0$ .)

**Master Theorem.** Let

$$T(n) = a \cdot T(n/b) + f(n)$$

be a recurrence relation on the nonnegative integers, where  $a \geq 0$  and  $b > 1$  are constants, and  $f(n)$  is an asymptotically non-negative function of  $n$ . Then,

1. If  $f(n) = O(n^{(\log_b a) - \epsilon})$  for some constant  $\epsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$ .

2. If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \cdot \log n)$ .
3. If  $f(n) = \Omega(n^{(\log_b a) + \epsilon})$  for some constant  $\epsilon > 0$ , and an additional technical condition holds, then  $T(n) = \Theta(f(n))$ .

**Proof** (of parts 1 and 2 of the Master theorem for the case when  $n = b^i$ ):

Setting  $r = \log_b n$  we have

$$\begin{aligned}
 T(n) &= a \cdot T(n/b) + f(n) && \text{(given)} \\
 &= a \cdot (aT(n/b^2) + f(n/b)) + f(n) && \text{(applying the recurrence to } T(n/b)) \\
 &= a^2T(n/b^2) + af(n/b) + f(n) && \text{(simplifying)} \\
 &\dots \\
 &= a^i T(n/b^i) + \sum_{j=0}^{i-1} a^j f(n/b^j) && \text{(prove correctness by induction on } i) \\
 &\dots \\
 &= a^r T(1) + \sum_{j=0}^{r-1} a^j f(n/b^j) && \text{(since } b^r = n)
 \end{aligned}$$

By noting that  $a^r = n^{\log_b a}$ , we can now establish parts 1 and 2 of the Master Theorem by substituting the appropriate condition on  $f(n)$  in the terms in the summation, and then bounding the summation.

Here are some points to note about the Master theorem:

- The Master theorem continues to hold when floors and ceilings are used in the recurrence relation (see your textbook for a proof).
- Part 3 of the Master theorem is somewhat less interesting for our purposes because in terms of a divide-and-conquer algorithm, it represents one in which the cost of dividing and combining dominates the cost of solving the problem recursively.
- There are several natural divide and conquer recurrence relations that do not satisfy the requirements of the Master theorem, such as the following recurrence relation (*why?*):

$$T(n) = 2T(n/2) + \Theta(n \log n); \quad T(1) = 1$$

If you obtain such a recurrence relation for the running time of your divide-and-conquer algorithm, then you will have to solve it using the substitution or iteration method; you cannot use the Master theorem.