Divide-and-Conquer

We now start introducing the first algorithm-design technique: divide-and-conquer. A typical divide-and-conquer algorithm follows the framework below.

1. partition the original problem into smaller problems;
2. recursively solve all subproblems;
3. combine the solutions of the subproblems to obtain the solution of the original problem.

We use sorting as the first problem to demonstrate designing divide-and-conquer algorithms. Recall that the *sorting* problem is to find the sorted array (say, in increasing order) S' of a given array S. We now design a divide-and-conquer algorithm for it. For any recursive algorithm, we always need to clearly define the recursion. In this case, we define function merge-sort (S) returns the sorted array (in ascending order) of S.

The idea is to sort the first half and second half of S, by recursively calling the merge-sort function. How to obtain the sorted array of S then with the two sorted half-sized arrays? We have introduced such an algorithm to merge two sorted arrays into a single sorted array. That's exactly the algorithm we need here in the combining step. The pseudo-code for the merge-sort function is given below.

```
Algorithm merge-sort (S[1 ··· n])
    if n ≤ 1: return S;
    S'_1 = merge-sort (S[1 ··· n/2]);
    S'_2 = merge-sort (S[n/2 + 1 ··· n]);
    return merge-two-sorted-arrays (S'_1, S'_2);
end algorithm;
```

Analysis of Merge-Sort

The correctness of the algorithm can be proved by induction, as the natural structure of above algorithm is recursive. Specifically, we want to prove the statement that the merge-sort function with A as input returns the sorted array of A if $|A| = n$. The induction is w.r.t. n. The base case, i.e., $n = 1$, is clearly correct. In the inductive step, we assume that above statement is true for $|A| = 1, 2, \cdots, n-1$, and we aim to prove it is correct for $|A| = n$. As the algorithm is correct for $|A| = 1, 2, \cdots, n-1$, in particular, it is correct for $|A| = n/2$, i.e., S'_1 and S'_2 store the sorted array of S_1 and S_2, respectively. Combinining the correctness of merge-two-sorted-arrays that we have already proved, we have that merge-sort returns the sorted array of S.

To see its runing time, we define $T(n)$ as the running time of merge-sort (S) when $|S| = n$. Clearly, both merge-sort ($S[1 \cdots n/2]$) and merge-sort ($S[n/2+1 \cdots n]$) take $T(n/2)$ time. As merge-two-sorted-arrays takes $\Theta(|S'_1| + |S'_2|) = \Theta(n)$ time, we have the recurrence $T(n) = 2T(n/2) + \Theta(n)$. We will use *master's theorem* to get the closed form for $T(n)$, described below.

Master's Theorem

Master's theorem gives closed form for the following recurrence:

$$T(n) = \begin{cases} aT(n/b) + \Theta(n^d) & \text{if } n \geq 2 \\ 1 & \text{if } n \leq 1 \end{cases}$$

Master's theorem is widely used to analyze the running time of divide-and-conquer algorithms. Recall that a divide-and-conquer algorithms first partition the original problems into subproblems, solve all subproblems recursively, and then combine them to answer the original question. Hence, above recurrence precisely describes the running time of such algorithms. Specifically, a refers to the number of subproblems that the original problem is partitioned, n/b refer to the input-size of each subproblem, and $\Theta(n^d)$ refer to the running time of the combining step. In merge-sort, $a = 2$, as it calls merge-sort twice in the algorithm, $b = 2$, as the input-size of each subproblem becomes $n/2$, and $d = 1$, as the merge-two-sorted-arrays takes $\Theta(n)$ time. Note that, it is not always the case that $a = b$ (in merge-sort though, $a = b$). Such example includes matrix multiplication, where $a = 4$ and $b = 2$.

We now solve above recurrence. Without loss of generality, we assume that n is a power of b, i.e., $n = b^k$ for some k. To further simplify, we use n^d instead of $\Theta(n^d)$. We therefore need to add $\Theta(\cdot)$ to the resulting formular of $T(n)$.

$$\begin{aligned} T(n) &= aT(n/b) + n^d \\ &= a(aT(n/b^2) + (n/b)^d) + n^d \\ &= a^2T(n/b^2) + a(n/b)^d + n^d \\ &= a^2(aT(n/b^3) + (n/b^2)^d) + a(n/b)^d + n^d \\ &= a^3T(n/b^3) + a^2(n/b^2)^d + a(n/b)^d + n^d \\ &= \cdots \\ &= a^kT(n/b^k) + \sum_{i=0}^{k-1} a^i(n/b^i)^d \end{aligned}$$

As we assume that $n = b^k$ and $T(1) = 1$, we have

$$T(n) = a^k + \sum_{i=0}^{k-1} a^i(n/b^i)^d = \sum_{i=0}^{k} a^i(n/b^i)^d = n^d \sum_{i=0}^{k} (a/b^d)^i.$$

Consider the following 3 cases.

1. If $a = b^d$, i.e., $d = \log_b a$, then $T(n) = n^d k = n^d \log_b n = \Theta(n \log n)$.
2. If $a < b^d$, i.e., $d > \log_b a$, then the series decreases exponentially, and therefore the item of $i = 0$ dominates. $T(n) = n^d a/b^d = \Theta(n^d)$.
3. If $a > b^d$, i.e., $d < \log_b a$, then the series increases exponentially, and therefore the item of $i = k$ dominates. $T(n) = n^d(a/b^d)^k = n^d a^k/b^{dk} = n^d a^k/n^d = a^k = a^{\log_b n} = n^{\log_b a}$.

We have used one facts about logarithmic function above: $a^{\log_b n} = n^{\log_b a}$.

Master's theorem can be summarized as below. For recurrence $T(n) = aT(n/b) + n^d$ with $T(1) = 1$, we have the following:

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

In the case of merge-sort, $a = b = 2$ and $d = 1$. So $d = \log_b a = 1$. Hence, $T(n) = \Theta(n \log n)$.

A more generalized form of master's theorem is to solve this recurrence: $T(n) = aT(n/b) + n^d \log^s n$ with $T(1) = 1$. The closed form is given below:

$$T(n) = \begin{cases} \Theta(n^d \log^{s+1} n) & \text{if } d = \log_b a \\ \Theta(n^d \log^s n) & \text{if } d > \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$