Introduction to Algorithms

Algorithm, Problem, and Instance

What's an algorithm? Algorithm is a sequence of unambiguous specifications/instructions for solving a problem. Such specifications can be followed by a human, or implemented with a machine program. The *problem* studied in theoretical computer science needs to be formally defined, usually described using input and output.

For example, consider the *sorting problem*: its input is an array $A = [a_1, a_2, \cdots, a_n]$, and its output is the sorted array, say in ascending order, of A.

A problem describes a template. It's concrete, instantiated version is called an *instance*. For example, an instance for above sorting problem would be $A = [4, 2, 10, -5, 8]$. The relationship between problem and instance is very similar to that between *class* and *object* in object-oriented programming languages. You may also think a problem is the set of all its possible instances.

Principles of Algorithm Design

There is always a straightforward algorithm to solve a problem, which is just enumerating all possible solution (and then pick the correct/best one). This is called *brute-force*. Brute-force may work for instances of size small, but in general it is *not efficient*.

We focus on designing *efficient* algorithms in this course. Here, "efficient" means "runs in polynomial-time", which we will formally define. There are two basic principles in designing efficient algorithms. Recall that an *algorithm* produces a *solution* for a *problem*.

1. *Partitioning problem into smaller subproblems.* This principle is from the perspective of problem. There are two strategies to reduce the size of a problem. First, we can partition a problem of size n into smaller subproblems with size of $n-1$; this strategy leads to *dynamic programming* algorithms. Second, we can partition a problem of size n into smaller subproblems with size of $n/2$; this strategy leads to *divide-and-conquer* algorithms.

2. *Iteratively Improving.* This principle is from the perspective of solution, i.e., we start with an initial/trivial solution, and we then gradually improve it to finally obtain the (optimal) solution. There are two strategies. First, we can start with a feasible but non-optimal solution, and then gradually improve it to achieve an optimal solutions; *linear programming* and *network flow* algorithms follow this strategy. Second, we can start with a partial-solution, and then gradually make it to a complete-solution; *greedy* algorithms follow this strategy.

Relationship between Problems

We not only design algorithms for individual problems, but also study the relationship between problems. This is very important. Studying how problems relate help in the following three folds.

1. Allow us to build the hierarchy (i.e., classes of problems) and understand them. This is the targets of *theory of computational complexity*.

2. Allow us to solve new problems using existing algorithms. For example, if we know that problem X can be transformed into problem Y (we will formally define what does this mean) and we know a good algorithm for problem Y, and we know that we can use this algorithm for Y to solve X.

3. Allow us to prove that a problem is hard to solve. For example, if we know that problem X can be transformed into problem Y, and that there does not exist any efficient algorithhm for X, then we know that there does not exist efficient algorithm for problem Y.

Example of Algorithm Design and Analysis

Let's consider the problem of *merging two sorted arrays*. The input of this problem is two sorted arrays, in ascending order, A and B, and the output is a sorted array C that consists of all elements in A and B.

An instance of this problem is: $A = [-4, 2, 5, 8]$ and $B = [-3, 2, 3, 4]$. For this instance, the output should be $C = [-4, -3, 2, 2, 3, 4, 5, 8]$.

An algorithm usually requires *data structures* to store key intermediate information. In this case, we maintain two pointers, k_A and k_B. Throughout the algorithm, we guarantee that k_A and k_B point to the smallest number in A and B that haven't been added to C.

The idea of the algorithm is to iteratively construct C (so you may call it a greedy algorithm). In each step, we compare the two numbers at the two pointers: the smaller one of the two will be the smallest one in *all* numbers that haven't been added to C; we then add it to C and update the pointers accordingly. The pseudo-code for this algorithm is given below.

```
function merge-two-sorted-arrays (A[1 ··· m], B[1 ··· n])
    init an empty array C; (#units = 1)
    init pointers k_A = 1 and k_B = 1; (#units = 2)
    add a big number M (larger than any number in A and B) to the end of A and B: A[m+1] = M and
    B[n+1] = M; (think: why we do it?); (#units = 2)
    for k = 1 → m+n (#units = m+n)
        if A[k_A] ≤ B[k_B] (#units = m+n)
            C[k] = A[k_A]; (#units = a_1)
            k_A = k_A + 1; (#units = a_2)
        else
            C[k] = B[k_B]; (#units = m+n−a_1)
            k_B = k_B + 1; (#units = m+n−a_2)
        end if;
    end for;
    return C; (#units = 1)
end algorithm;
```

To analyze an algorithm, we need to (1) prove it's correct, and (2) analyze its running time.

We first prove this algorithm is correct, i.e., the resulting C is sorted and includes all numbers in A and B. This seems obvious. To give a formal and complete proof, we define the following *invariant*.

Invariant: For any $k = 0, 1, 2, \cdots, m+n$, at the end of the k-th iteration of above algorithm, we must have that C stores the smallest k numbers in A and B, k_A and k_B points to the smallest number of $A \setminus C$ and $B \setminus C$, respectively.

We now prove above invariant is correct, by *induction*. The base case is $k = 0$. At this time (i.e., before the for-loop and after the initialization), C is empty and k_A and k_B points to the first number in A and B respectively (and therefore the smallest one, as the given A and B are sorted). We now show the inductive step: assume that the invariant is correct for $k = 0, 1, \cdots, i$, we prove that the invariant is correct for $k = i+1$. The inductive assumption tells that at the end of the i-th iteration, $C[1 \cdots i]$ stores the first i smallest numbers. Consider what the algorithm does in the $(i+1)$-th iteration: it compare the numbers at the pointers, and adds the smaller one to C. As each pointer now points to the smallest one in $A \setminus C$ and $B \setminus C$, we have that the smaller one is the $(i+1)$-th smallest number in A and B. Besides, how the algorithm updates the pointers guarantees the pointers always point to the smallest one in $A \setminus C$ and $B \setminus C$, respectively. Therefore, the invariant holds after $(i+1)$-th iteration.

We then analyze its running time. The running time is measured with the *basic computing units* used by the algorithm in the *worst case*. The definition of basic computing units depends on the computing model. Usually, we think each operation like assignment, basic arithmetic operation, comparison, etc, takes 1 unit. The worst case is w.r.t. all instances, i.e., the instance takes largest number of computing units.

The required units in each step of the algorithm is marked with blue text. Let $T(m,n)$ be the running time of this algorithm when $|A| = m$ and $|B| = n$. We can write $T(m,n) = 4m+4n+6$.

Note that the running time of an algorithm is a function of *input size*, rather than the *input*. The input size is usually the amount of memory needed to store the actually input. In this example, array of size m can be stored in a block of memory with m memory units. An implicit assumption here is that, every number in the given arrays can be stored in a single memory unit (for example, 32 bits to store an integer in C++). This is reasonable and widely used. In certain cases, where we study very large number (which cannot be stored in 32 bits for example), then we cannot assume that such number takes 1 memory unit; instead, a (large) number x uses $\log_2 x$ bits in memory.

The running time of $T(m,n) = 4m+4n+6$ a brief, coarse estimation, as it depends on the computing model. Therefore, it doesn't make much sense to keep the coefficients 4 and the constants 5. Besides, we care how the running time grows as m and n grows. Hence, it suffices to write $T(m,n) = \Theta(m+n)$, meaning it grows as fast as function $m+n$. We will introduce asymptotic notations to facilitate this way of analyzing running time in next lecture.