CS3230 Design & Analysis of Algorithms **January 27, 2022**

Lecture 3: Iteration, Recursion & Divide-and-Conquer

Lecturer: Warut Suksompong

The goal of today's lecture is to discuss two basic and widely occurring patterns in the design of algorithms: iteration and recursion. We will study how to rigorously argue that algorithms fitting these patterns are correct and how to bound their asymptotic running time.

3.1 Iteration

Loops (e.g., **for**, **while**) are basic constructs in almost[2] all programming languages. An iterative algorithm processes its inputs one or more times in sequence, updating some variables or data structures as it progresses.

For any given algorithm, we have two concerns: correctness and running time. Bounding the running time of iterative algorithms is typically straightforward, only requiring some careful bookkeeping, if the number of iterations is fixed[3]. The more non-trivial aspect is often proving correctness, and this is what we focus on in this section.

To make the discussion more concrete, consider the well-known Insertion-Sort algorithm for sorting an array A of n numbers.

```
Insertion-Sort(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j-1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8      A[i+1] = key
```

How does one formally prove the correctness of such an algorithm? One useful strategy is to show the existence of a *loop invariant*. For a loop running for N iterations, a loop invariant is a sequence of assertions $S_1, S_2, \ldots, S_N$ such that each S_i holds at the start of the i-th iteration and such that S_N implies the desired correctness property.

In Insertion-Sort, there are two loops (lines 1–8 and lines 5–7). We first consider the inner loop. Fix $j = J$ in the outer loop. The claim makes precise the following inner loop invariant: before the iteration with $i = I$, $A[1..I]$ is yet untouched and $A[I+2..J]$ are all elements bigger than *key* that have been shifted one place right from their original position.

Claim 3.1.1 (Inner loop invariant)**.** *Let A' be the array A before line 5 in an outer loop iteration with $j = J$. At the start of an inner loop iteration with $i = I$, the following hold:*

1. $A[1..I] = A'[1..I]$.

[1] See also Chapters 2.1, 2.3, and 4 of CLRS.

[2] Some functional programming languages like *Scheme* don't have explicit support for loops, but they support 'tail recursion' which is basically the same thing.

[3] The Collatz algorithm from Lecture 1 is also iterative, but there, the number of iterations is not fixed (indeed, not known to be finite).

2. $A[I+2\,..\,J] = A'[I+1\,..\,J-1]$.

3. *All elements in* $A[I+2\,..\,J]$ *are greater than* $A'[J] = key$.

Proof. The proof is by induction. The claim is trivial for the base case $I = J-1$. Suppose the claim holds for I, and the inner loop continues to $I-1$, we want to argue for $I-1$. By the induction hypothesis, at the start of the run with $i = I$,

(1) $A[1\,..\,I] = A'[1\,..\,I]$

(2) $A[I+2\,..\,J] = A'[I+1\,..\,J-1]$

(3) $A[I+2\,..\,J] > A'[J] = key$

Suppose I continues to $I-1$. Condition 1 holds true because: $A[1\,..\,I] = A'[1\,..\,I]$ and (1) gives $A[1\,..\,I-1] = A'[1\,..\,I-1]$. Condition 2 holds true because: $A[I] = A'[I]$ (from condition 1) and $A[I+1] = A[I]$ (after an inner loop iteration), therefore $A[I+1] = A'[I]$, combined with (2) gives $A[I+1\,..\,J] = A'[I\,..\,J-1]$, which is equivalent to $A[(I-1)+2\,..\,J] = A'[(I-1)+1\,..\,J-1]$. Condition 3 holds true because: $A'[I] = A[I] > key$ and $A[I+1] = A'[I]$ (after an inner loop iteration), therefore $A[I+1] > key$, combined with (3) gives $A[I+1\,..\,J] > key$, which is equivalent to $A[(I-1)+2\,..\,J] > key$. □

Suppose the inner loop terminates when $i = I$. This can happen because of two reasons, either $I = 0$ or $A[I] \le key$. In the former case, Conditions 2 and 3 of the inner loop invariant gives $A[2\,..\,J] = A'[1\,..\,J-1]$ and $A[2\,..\,J] > key$. In the latter case, the inner loop invariant gives: $A[1\,..\,I] = A'[1\,..\,I]$, $A[I+2\,..\,J] = A'[I+1\,..\,J-1]$ and $A[I+2\,..\,J] > key$.

Claim 3.1.2 (Outer loop invariant)**.** *At the start of an outer loop iteration with* $j = J$, $A[1\,..\,J-1]$ *is the sorted list of elements originally in* $A[1\,..\,J-1]$.

Proof. The proof is by induction. The claim is trivial for the base case $J = 2$. Suppose the claim holds for J, and we want to argue for $J+1$. By the induction hypothesis, at the start of the run with $j = J$, $A[1\,..\,J-1]$ is a sorted arrangement of the original elements in these positions. Call this array A'. After the inner loop terminates in this run of the outer loop, by the paragraph after Claim 3.1.1, there can be two cases:

- $I = 0$. In this case, $key < A[2] \le \cdots \le A[J]$ where $A[2\,..\,J]$ is a right-shift of $A'[1\,..\,J-1]$.
- $A[I] \le key$. In this case, $A[1] \le \cdots \le A[I] \le key$ because these elements are the same as in A' and they were sorted, while $key < A[I+2] \le \cdots \le A[J]$ because these elements are a right-shift of $A'[I+1\,..\,J-1]$ which were sorted.

In either case, line 8 ($A[I+1] = key$) ensures that $A[1\,..\,J]$ is sorted and consists of the elements in $A'[1\,..\,J-1]$ and key. □

Applying Claim 3.1.2 for $J = A.length + 1$ proves the correctness of Insertion-Sort. Admittedly, it may seem that we are belaboring the analysis of Insertion-Sort for no reason as it's sort of obvious. Why we go into so much detail here is because we want to explain the basic concepts in a familiar context. In other situations, where the algorithms are more unfamiliar or complicated, loop invariants provide a precise way to capture the work done in each iteration.

Example 1. Consider the following algorithm to determine which bit appears at least $n/2$ times in an input array A of n bits.

```
MISRA-GRIES(A)
 1  id = ⊥
 2  count = 0
 3  for i = 1 to A.length
 4       if A[i] == id
 5            count = count + 1
 6       elseif id == ⊥
 7            id = A[i]
 8            count = 1
 9       else
10            count = count − 1
11            if count == 0
12                 id = ⊥
13  if id == ⊥
14       return "Tie"
15  else return id
```

MISRA-GRIES satisfies the following loop invariant.

Claim 3.1.3. *At the start of an iteration of the* ***for*** *loop with $i = I$:*

- *If there are an equal number of 0's and 1's in $A[1\,..\,I-1]$, then $id = \perp$ and $count = 0$.*
- *If $z \in \{0, 1\}$ is the majority element in $A[1\,..\,I-1]$, then $id = z$ and count equals the difference between the count of z and the count of $1 - z$ in $A[1\,..\,I-1]$.*

Proof. The proof is by induction. Suppose the claims hold true for I, we want to argue for $I + 1$.

Proof for the former claim for $I + 1$: If there are an equal number of 0's and 1's in $A[1\,..\,I]$, then the number of 0's and 1's are different in $A[1\,..\,I-1]$, and $A[I]$ is the minority element. Apply the latter claim for I, at the start of the loop for I, $id = 1 - A[I]$ and $count = 1$. After the loop at I, $count = count - 1 = 0$ and $id = \perp$.

Proof for the latter claim for $I + 1$: Assume the numbers of 0's and 1's are different in $A[1\,..\,I]$. There are two cases. Consider the first case where there are an equal number of 0's and 1's in $A[1\,..\,I-1]$, after the loop at I, $id = A[I]$ which is the majority element now and $count = 1$ which is the difference. The second case is when at the start of the loop at I, id is the majority element. If $A[I] = id$, $count = count + 1$, and id is still the majority element, and the difference now increases by 1. If $A[I] \neq id$, $count = count - 1$ and the difference now decreases by 1, id is still the major element unless $count = 0$ which contradicts the assumption. □

The idea behind MISRA-GRIES is used for space-efficient frequency estimation algorithms in the *streaming* setting. We will talk about streaming briefly in Week 6.

Example 2. Given two non-negative integers a and b, their *greatest common divisor (gcd)*, denoted $\gcd(a, b)$ is the largest of the common divisors of a and b. The following algorithm to compute it is one of the oldest[4] algorithms in common use, more than 2000 years old.[5]

```
EUCLID(a, b)
1  while b > 0
2       c = a
3       a = b
4       b = c (mod b)
5  return a
```

[4] Knuth writes in *The Art of Computer Programming*, volume 2: "[The Euclidean algorithm] is the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day."

[5] Recall that for $u \geq 0, n > 0$, the notation $u \pmod n$ denotes the remainder of the quotient u/n, meaning $u \pmod n = u - n\lfloor u/n \rfloor$.

The running time of this algorithm is an interesting problem which we won't discuss here. (It is clear though that EUCLID terminates in finite time because b keeps strictly decreasing.) To establish a loop invariant, the following observation is critical.

Claim 3.1.4. *If* $a \geq 0, b > 0$, *then* $\gcd(a, b) = \gcd(b, a \mod b)$.

Proof. Suppose for some integers $q \geq 0$ and $0 < r < b$, we can write $a = qb + r$, so that $r = a \pmod b$. Let $x = \gcd(a, b)$ and $y = \gcd(b, r)$. We will be done if we can show that x divides y, and also y divides x.

First, let's show that x divides y. Since x divides a as well as b, it must also divide $r = a - qb$. As x is a common divisor of both b and r, it must divide $\gcd(b, r)$.

Similarly, let's show that y divides x. Since y divides b as well as r, it must also divide $a = qb + r$. As y is a common divisor of both a and b, it must divide $\gcd(a, b)$. □

Thus, if a^* and b^* were the original inputs to EUCLID, then an invariant which holds at the start of every iteration of the **while** loop is that: $\gcd(a, b) = \gcd(a^*, b^*)$. This follows immediately from Claim 3.1.4. At termination, $b = 0$. Since $\gcd(a, 0)$ is trivially a, it follows that EUCLID returns $\gcd(a^*, b^*)$ at termination.

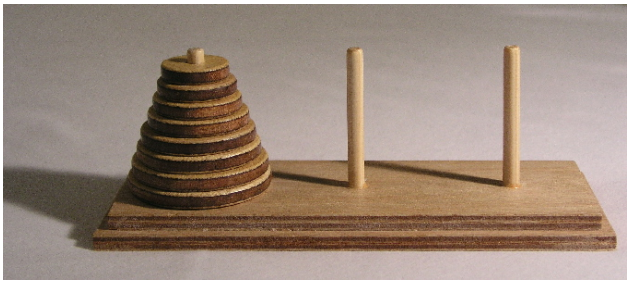
3.2 Divide-and-Conquer

Divide-and-conquer is an incredibly general and powerful algorithm design paradigm. The idea is to design algorithms that have a *recursive* structure, meaning they call themselves one or more times on other inputs derived from the original input. More specifically, as CLRS describes on page 30, the divide-and-conquer strategy involves the following:

- *Divide* the problem into sub-problems that are smaller instances of the original problem.
- *Conquer* the sub-problems by solving them recursively.
- *Combine* the solutions to the sub-problems into the solution for the original problem.

Recursion feels like a sprinkle of magic (Erickson talks about the "recursion fairy" in his book), because it sometimes feels like it allows us to solve a problem without *really* solving it. When it works, it's because the problem admits a decomposition into sub-problems so that the solutions to the sub-problems can be efficiently combined to get the overall solution. Instead of talking in generalities, it's more instructive to look at examples where we can put these ideas to use.

Example 3. (Tower-of-Hanoi) A famous example showcasing the power of recursion is the famous "Tower of Hanoi" problem. The puzzle, shown below, consists of 3 pegs where initially, the first peg has disks stacked in order of decreasing size and the goal is to move the whole stack to the third peg while obeying two rules: (1) One disk may be moved at at time, from the top of one stack to the top of another stack, and (2) No disk may be placed on top of a disk smaller than it.



The computational problem is to find a sequence of move instructions when there are n disks on the first peg. Note that the second peg can used to temporarily store the disks as they are being moved from the first to the third.

The recursive solution is extremely elegant: Move the top $n-1$ disks from the first to the second peg using the third for temporary storage. Now, move the biggest disk directly to the currently empty third peg. Finally, move the $n-1$ disks from the second peg to the third using the first peg for temporary storage.

```
HANOI(n, src, dst, tmp)
1  if n ≥ 1
2      HANOI(n − 1, src, tmp, dst)
3      Move disk n from src to dst
4      HANOI(n − 1, tmp, dst, src)
```

Although correctness is sort of obvious, we can prove it formally via induction. The base case is $n = 0$. Here, there is nothing to do. Now, suppose HANOI correctly moves $n-1$ disks. Because the top $n-1$ disks are all smaller than the n-th disk, HANOI describes a correct sequence of moves for moving n disks.

What about the total number of moves? Let $H(n)$ be the number of moves HANOI makes for n disks. $H(1) = 1$, and for $n > 1$, it follows the recurrence: $H(n) = 2 \cdot H(n-1)+1$. It is easy to see that $H(n) = 2^n - 1$.

The recursive algorithm HANOI does not explicitly give the sequence of move instructions. It is complicated to unroll the recurrence and get an iterative algorithm. If you're interested, check the Wikipedia page on the topic.

Example 4. (Merge Sort) Merge Sort is a divide-and-conquer algorithm used to sort an input array. The idea is that in order to sort an array $A[l\,..\,r]$, we split it into two equal halves $A[l\,..\,m]$ and $A[m+1\,..\,r]$, call Merge Sort to sort both of them, and merge them to get a sorted array of $A[l\,..\,r]$.

Since both of the halves are sorted, it is possible to merge them in $O(n)$ time, where n is the size of the array. We keep two indices $i = l$ and $j = m+1$ to point to the current value in $A[l\,..\,m]$ and $A[m+1\,..\,r]$ respectively. We start building the new sorted array $A'[]$ by comparing $A[i]$ with $A[j]$. Without loss of generality, let $A[i] < A[j]$. Then $A[i] < A[j\,..\,r]$, and $A[i] < A[i+1\,..\,m]$, and so, we know that $A[i]$ is the smallest element among all the elements that haven't been put into $A'[]$, and we put $A[i]$ into $A'[]$ as the next element. We continue until all elements are put in $A'[]$, the final sorted array.

```
MERGE(A[], B[])
 1  i = 1, j = 1
 2  A.append(∞), B.append(∞)
 3  C[]
 4  while i < A.length or j < B.length
 5      if j = B.length
 6          C[i + j − 1] = A[i], i = i + 1
 7      elseif i = A.length
 8          C[i + j − 1] = B[j], j = j + 1
 9      elseif A[i] < B[j]
10          C[i + j − 1] = A[i], i = i + 1
11      else C[i + j − 1] = B[j], j = j + 1
12  return C[]
```

```
MERGE-SORT(A[], l, r)
1  if l = r return A[]
2  m = ⌊(l + r)/2⌋
3  leftArray = MERGE-SORT(A[], l, m)
4  rightArray = MERGE-SORT(A[], m + 1, r)
5  return MERGE(leftArray, rightArray)
```

Correctness of MERGE-SORT follows from strong induction (specifically, that MERGE-SORT correctly sorts *leftArray* and *rightArray*) and from the correctness of MERGE. We omit a proof of the latter. See CLRS (Chapter 2) for a detailed proof using loop invariants.

As for the time complexity, it is easy to check that MERGE on two arrays of size n and m runs in $\Theta(m+n)$ time. Since *leftArray* and *rightArray* have sizes of $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ respectively, the time complexity $T(n)$ for MERGE-SORT on arrays of size n is:

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

We analyze this recurrence in Section 3.3.

Example 5. (Powering) Binary exponentiation is a fast way to find power n of a number A. The idea is we start with base case of $A^0 = 1$. For $n > 0$, if n is odd, $A^n = (A^{(n-1)/2})^2 \cdot A$, and if n is even, $A^n = (A^{n/2})^2$.

RECURSE-POWER(A, n)
1 **if** $n = 0$ **return** 1
2 **if** n is odd **return** RECURSE-POWER$(A, (n-1)/2)^2 \cdot A$
3 **else return** RECURSE-POWER$(A, n/2)^2$

Correctness is by strong induction. If $T(n)$ denotes the worst-case running time of RECURSE-POWER:

$$T(n) = T(\lfloor n/2 \rfloor) + \Theta(1).$$

Example 6. (Karatsuba Multiplication) The standard primary-school algorithm to multiply two n-digit numbers takes time $O(n^2)$ (taking additions and multiplications of pairs of single digits in $O(1)$ time). Can we do better? Kolmogorov, a larger-than-life Russian scientist who invented modern probability, statistics, and information theory, believed not, and organized a seminar at Moscow University in 1960 with the agenda being to prove this claim. A participant was 23-year old Anatolii Karatsuba. Here's what happened in Karatsuba's own words (taken from Erickson's book, *Algorithms*):

After the seminar I told Kolmogorov about the new algorithm and about the disproof of the n^2 conjecture. Kolmogorov was very agitated because this contradicted his very plausible conjecture. At the next meeting of the seminar, Kolmogorov himself told the participants about my method, and at that point the seminar was terminated.

In some sense, Karatsuba's algorithm for integer multiplication started the whole field of algorithm design and analysis!

Karatsuba Multiplication is an algorithm to reduce the number of single-digit multiplications when multiplying two n-digit numbers from $O(n^2)$ to $O(n^{\log_2 3})$. Let's say we want to multiply two n-digit numbers x and y. Let $m = \lceil n/2 \rceil$. First we split x into the form of $x_1 10^m + x_0$, and y into the form of $y_1 10^m + y_0$ where x_1, x_0, y_1, y_0 have less than or equal to m digits. Then, $xy = (x_1 10^m + x_0) \cdot (y_1 10^m + y_0) = x_1 y_1 10^{2m} + (x_1 y_0 + x_0 y_1) 10^m + x_0 y_0$. It usually requires 4 multiplications of (m or $m+1$)-digit numbers to calculate $x_1 y_1$, $(x_1 y_0 + x_0 y_1)$, and $x_0 y_0$. However, $(x_1 y_0 + x_0 y_1) = -(x_1 - x_0) \cdot (y_1 - y_0) + x_1 y_1 + x_0 y_0$. Now, we only need 3 multiplications of m-digit numbers to calculate xy.

KARATSUBA(x, y, n)
1 **if** $x < 10$ and $y < 10$ **return** $x \cdot y$
2 $m = \lceil n/2 \rceil$
3 $x_1 = \lfloor x/10^m \rfloor$, $x_0 = x \pmod{10^m}$
4 $y_1 = \lfloor y/10^m \rfloor$, $y_0 = y \pmod{10^m}$
5 $z_0 =$ KARATSUBA(x_0, y_0, m)
6 $z_1 =$ KARATSUBA(x_1, y_1, m)
7 $z_2 =$ KARATSUBA$(x_1 - x_0, y_1 - y_0, m)$
8 **return** $z_1 \cdot 10^{2m} + (-z_2 + z_1 + z_0) 10^m + z_0$

Note how z_1 is reused in the code above. This is eventually what leads to the runtime savings. Denoting by $T(n)$ the worst-case runtime of KARATSUBA, we get:

$$T(n) = 3T(\lceil n/2 \rceil) + \Theta(n)$$

3.3 Solving Recurrences

As we saw in the previous section, recurrences naturally arise when analyzing recursive algorithms. Very commonly, these recurrences are of the following form:

$$T(n) = aT(n/b) + \Theta(f(n)) \tag{3.1}$$

along with a base case (e.g., $T(n) = \Theta(1)$ for $n \leq n_0$). This corresponds to a divide-and-conquer algorithm which divides a problem on inputs of size n into a sub-problems on inputs of size n/b, with time overhead $\Theta(f(n))$ for dividing into sub-problems and combining solutions of the sub-problems into a solution of the original problem. Note that we are being sloppy in even just writing (3.1) because n/b may not be an integer while the sub-problem sizes are obviously integers. Also, we may need to solve more complicated recurrences where the sub-problems are not all of exactly the same size. We will re-visit these concerns later below.

There are three standard ways to approach recurrences of the form (3.1): (i) recursion trees, (ii) master method, and (iii) substitution method. These are described well in Sections 4.4, 4.5, and 4.3 respectively of CLRS (3rd edition), so there is no need for me to repeat here. These sections are **required reading** for the contents of this lecture.

Example 7. (MERGE-SORT analysis) First consider the simplified recurrence:

$$T_{\text{simp}}(n) = 2T_{\text{simp}}(n/2) + O(n)$$

Then, using the Master theorem with $a = 2, b = 2$, and $f(n) = n$, we get that $T_{\text{simp}}(n) = O(n \log n)$.

The actual recurrence is more complicated:

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

We will now argue why we can reduce to the simplified case. First of all, note that since we want an upper bound, we can write:

$$T(n) \leq 2T(\lceil n/2 \rceil) + O(n) \leq 2T(n/2 + 1) + O(n)$$

since T is a non-decreasing function. (For the last inequality, we are extending T to a function over the reals instead of just over the integers.) Let $S(n) = T(n + 2)$. Note that we have the recurrence:

$$S(n) = T(n + 2) \leq 2T((n + 2)/2 + 1) + O(n) = 2S(n/2) + O(n)$$

Thus, S does satisfy the simplified recurrence, and hence, $T(n+2) = O(n \log n)$ implying $T(n) = O(n \log n)$.

Remark. In the future, we won't bother with this kind of detailed analysis of floors and ceilings in recurrences. It turns out that one can ignore them without loss of generality for the purposes of recovering asymptotic behavior. For details, see Section 4.6.2 of CLRS.

Example 8. (Powering analysis) For the RECURSE-POWER algorithm, the recurrence was $T(n) = T(n/2) + \Theta(1)$, again ignoring floors as remarked above. By the Master theorem, $T(n) = \Theta(\log n)$.

Example 9. (Karatsuba analysis) As argued in Example 6, the recurrence to analyze here is: $T(n) \leq 3T(n/2) + O(n)$, again ignoring floors and ceilings. The recursion tree method or the Master method yields the bound: $T(n) = O(n^{\log_2 3})$.

Remark. Although we won't have occasion to use it in this module, there exists a counterpart of the Master theorem which holds when there are multiple sub-problems of different sizes. Suppose we have recurrences of the form:

$$T(n) = \sum_{i=1}^{k} a_i T(n/b_i) + f(n)$$

with a base case $T(n) = O(1)$ if $n \leq n_0$. The Akra-Bazzi theorem described in the "Chapter notes" section of Chapter 4 in CLRS handles such recurrences.