QUICK NOTES FOR DSA

TOWER OF HANOI

Definition: *A **recurrence relation** is an equation that recursively defines a sequence. That is, each term in the sequence is defined as a function of the preceding terms in some way.*

Algorithms that repeatedly break something down until a base case is reached and build a solution back
up are known as **divide and conquer** algorithms.

For the groups of students problem stated above, we can define a function, $G(n)$, that calculates the number of groups of two that can be formed from n total students. We could express this function as follows:

$$G(n)=\begin{cases}0 & \text{, if } n=1\\ (n-1)+G(n-1) & \text{, otherwise}\end{cases}$$

We read this as follows: the number of groups of two that can be formed from n students is:

- 0, if the number of students, n, is 1; or
- n minus 1 plus the number of groups of two that can be formed from n minus 1 students, otherwise.

Definition: ***Recursion** is the process of breaking down a problem into smaller and smaller versions of itself until a base or trivial case is reached. Recursion must have two parts: (1) a base or trivial case that provides an immediate answer to some specified input; and (2) a recursive step that breaks the problem down into a smaller version of itself.*

Let's take a look at the first example again: $2^2 = 2 * 2$. Technically, $2^2 = 2 * 2^1$. And $2^1 = 2 * 2^0$! So the base case can eventually be reached. We can now formally define a recurrence relation for some function `Pow(x, y)` as follows:

$$Pow(x,y)=\begin{cases}1 & \text{, if } y=0\\ x*Pow(x,y-1) & \text{, otherwise}\end{cases}$$

We can design a recursive algorithm in Python as follows:

```
def hanoi(n, src, dst, spr):
    if (n == 1):
        print "{} -> {}".format(src, dst)
    else:
        hanoi(n - 1, src, spr, dst)
        hanoi(1, src, dst, spr)
        hanoi(n - 1, spr, dst, src)
```

The numbers that we wish to sort are also known as the ***keys***.

We start with ***insertion sort***, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of

```
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
```


```
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
```


Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. The index j indicates the "current card" being inserted into the hand. At the beginning of each iteration of the **for** loop, which is indexed by j, the subarray consisting of elements $A[1 . . j - 1]$ constitutes the currently sorted hand, and the remaining subarray $A[j + 1 . . n]$ corresponds to the pile of cards still on the table. In fact, elements $A[1 . . j - 1]$ are the elements *originally* in positions 1 through $j - 1$, but now in sorted order. We state these properties of $A[1 . . j - 1]$ formally as a ***loop invariant***:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$.[1] The subarray $A[1 \,..\, j-1]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4–7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1 \,..\, j]$ then consists of the elements originally in $A[1 \,..\, j]$, but in sorted order. Incrementing j for the next iteration of the **for** loop then preserves the loop invariant.

A more formal treatment of the second property would require us to state and show a loop invariant for the **while** loop of lines 5–7. At this point, however, we prefer not to get bogged down in such formalism, and so we rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine what happens when the loop terminates. The condition causing the **for** loop to terminate is that $j > A.length = n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $A[1 \,..\, n]$ consists of the elements originally in $A[1 \,..\, n]$, but in sorted order. Observing that the subarray $A[1 \,..\, n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

[1] When the loop is a **for** loop, the moment at which we check the loop invariant just prior to the first iteration is immediately after the initial assignment to the loop-counter variable and just before the first test in the loop header. In the case of INSERTION-SORT, this time is after assigning 2 to the variable j but before the first test of whether $j \le A.length$.

Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs. For most of this book, we shall assume a generic one-processor, ***random-access machine (RAM)*** model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and would yield little insight into algorithm design and analysis. Yet we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then we could sort in just one instruction. Such a RAM would be unrealistic, since real computers do not have such instructions. Our guide, therefore, is how real computers are designed. The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time.

The data types in the RAM model are integer and floating point (for storing real numbers). Although we typically do not concern ourselves with precision in this book, in some applications precision is crucial. We also assume a limit on the size of each word of data. For example, when working with inputs of size n, we typically assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. We require $c \geq 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily. (If the word size could grow arbitrarily, we could store huge amounts of data in one word and operate on it all in constant time—clearly an unrealistic scenario.)

The best notion for ***input size*** depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the *number of items in the input*—for example, the array size n for sorting. For many other problems, such as multiplying two integers, the best measure of input size is the *total number of bits* needed to represent the input in ordinary binary notation. Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph. We shall indicate which input size measure is being used with each problem we study.

The ***running time*** of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the ith line takes time c_i, where c_i is a constant. This viewpoint is in keeping with the RAM model, and it also reflects how the pseudocode would be implemented on most actual computers.[5]

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.[6] To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost* and *times* columns, obtaining

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \,. \end{aligned}$$

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. We place on the bottom of each pile a ***sentinel*** card, which contains a special value that we use to simplify our code. Here, we use ∞ as the sentinel value, so that whenever a card with ∞ is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly $r - p + 1$ cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

```
MERGE(A, p, q, r)
 1  n_1 = q - p + 1
 2  n_2 = r - q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i - 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

```
BUBBLESORT(A)
1  for i = 1 to A.length - 1
2      for j = A.length downto i + 1
3          if A[j] < A[j - 1]
4              exchange A[j] with A[j - 1]
```

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\ , \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\ . \end{cases}$$

FOR MERGE-SORT

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

Next, we add the costs across each level of the tree. The top level has total cost cn, the next level down has total cost $c(n/2) + c(n/2) = cn$, the level after that has total cost $c(n/4)+c(n/4)+c(n/4)+c(n/4) = cn$, and so on. In general, the level i below the top has 2^i nodes, each contributing a cost of $c(n/2^i)$, so that the ith level below the top has total cost $2^i\, c(n/2^i) = cn$. The bottom level has n nodes, each contributing a cost of c, for a total cost of cn.

The total number of levels of the recursion tree in Figure 2.5 is $\lg n + 1$, where n is the number of leaves, corresponding to the input size. An informal inductive argument justifies this claim. The base case occurs when $n = 1$, in which case the tree has only one level. Since $\lg 1 = 0$, we have that $\lg n + 1$ gives the correct number of levels. Now assume as an inductive hypothesis that the number of levels of a recursion tree with 2^i leaves is $\lg 2^i + 1 = i + 1$ (since for any value of i, we have that $\lg 2^i = i$). Because we are assuming that the input size is a power of 2, the next input size to consider is 2^{i+1}. A tree with $n = 2^{i+1}$ leaves has one more level than a tree with 2^i leaves, and so the total number of levels is $(i + 1) + 1 = \lg 2^{i+1} + 1$.

To compute the total cost represented by the recurrence (2.2), we simply add up the costs of all the levels. The recursion tree has $\lg n + 1$ levels, each costing cn, for a total cost of $cn(\lg n + 1) = cn \lg n + cn$. Ignoring the low-order term and the constant c gives the desired result of $\Theta(n \lg n)$.

$f(n) = \Theta(g(n))$. For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

The definition of $\Theta(g(n))$ requires that every member $f(n) \in \Theta(g(n))$ be ***asymptotically nonnegative***, that is, that $f(n)$ be nonnegative whenever n is sufficiently large. (An ***asymptotically positive*** function is one that is positive for all sufficiently large n.) Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty. We shall therefore assume that every function used within Θ-notation is asymptotically nonnegative. This assumption holds for the other asymptotic notations defined in this chapter as well.

We interpret such equations using the following rule: *No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid.* Thus, our example means that for *any* function $f(n) \in \Theta(n)$, there is *some* function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$ for all n. In other words, the right-hand side of an equation provides a coarser level of detail than the left-hand side.

We can chain together a number of such relationships, as in

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) \ . \end{aligned}$$

Comparing functions

Many of the relational properties of real numbers apply to asymptotic comparisons as well. For the following, assume that $f(n)$ and $g(n)$ are asymptotically positive.

Transitivity:

$$\begin{aligned}
&f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)) ,\\
&f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)) ,\\
&f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)) ,\\
&f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \quad \text{imply} \quad f(n) = o(h(n)) ,\\
&f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad \text{imply} \quad f(n) = \omega(h(n)) .
\end{aligned}$$

Reflexivity:

$$\begin{aligned}
f(n) &= \Theta(f(n)) ,\\
f(n) &= O(f(n)) ,\\
f(n) &= \Omega(f(n)) .
\end{aligned}$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

Transpose symmetry:

$$\begin{aligned}
f(n) &= O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)) ,\\
f(n) &= o(g(n)) \text{ if and only if } g(n) = \omega(f(n)) .
\end{aligned}$$

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$$\begin{aligned}
&f(n) = O(g(n)) \quad \text{is like} \quad a \le b ,\\
&f(n) = \Omega(g(n)) \quad \text{is like} \quad a \ge b ,\\
&f(n) = \Theta(g(n)) \quad \text{is like} \quad a = b ,\\
&f(n) = o(g(n)) \quad \text{is like} \quad a < b ,\\
&f(n) = \omega(g(n)) \quad \text{is like} \quad a > b .
\end{aligned}$$

We say that $f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$, and $f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$.

One property of real numbers, however, does not carry over to asymptotic notation:

Trichotomy: For any two real numbers a and b, exactly one of the following must hold: $a < b$, $a = b$, or $a > b$.

Although any two real numbers can be compared, not all functions are asymptotically comparable. That is, for two functions $f(n)$ and $g(n)$, it may be the case that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds. For example, we cannot compare the functions n and $n^{1+\sin n}$ using asymptotic notation, since the value of the exponent in $n^{1+\sin n}$ oscillates between 0 and 2, taking on all values in between.

A weak upper bound on the factorial function is $n! \le n^n$, since each of the n terms in the factorial product is at most n. ***Stirling's approximation***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) , \qquad (3.18)$$

might be asked in growth function comparison ig??

$$
\begin{aligned}
n! &= o(n^n)\,, \\
n! &= \omega(2^n)\,, \\
\lg(n!) &= \Theta(n \lg n)\,,
\end{aligned}
$$

The iterated logarithm function

We use the notation $\lg^* n$ (read "log star of n") to denote the iterated logarithm, defined as follows. Let $\lg^{(i)} n$ be as defined above, with $f(n) = \lg n$. Because the logarithm of a nonpositive number is undefined, $\lg^{(i)} n$ is defined only if $\lg^{(i-1)} n > 0$. Be sure to distinguish $\lg^{(i)} n$ (the logarithm function applied i times in succession, starting with argument n) from $\lg^i n$ (the logarithm of n raised to the ith power). Then we define the iterated logarithm function as

$$\lg^* n = \min \left\{ i \geq 0 : \lg^{(i)} n \leq 1 \right\} .$$

The iterated logarithm is a *very* slowly growing function:

$$
\begin{aligned}
\lg^* 2 &= 1\,, \\
\lg^* 4 &= 2\,, \\
\lg^* 16 &= 3\,, \\
\lg^* 65536 &= 4\,, \\
\lg^*(2^{65536}) &= 5\,.
\end{aligned}
$$

BINARY TREES

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

Theorem 12.1
If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

We can rewrite this procedure in an iterative fashion by "unrolling" the recursion into a **while** loop. On most computers, the iterative version is more efficient.

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

successor of a node x is the node with the smallest key greater than $x.key$.

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == NIL
10      T.root = z          // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

with the input item z. The procedure maintains the ***trailing pointer*** y as the parent of x. After initialization, the **while** loop in lines 3–7 causes these two pointers

Deletion

The overall strategy for deleting a node z from a binary search tree T has three basic cases but, as we shall see, one of the cases is a bit tricky.

- If z has no children, then we simply remove it by modifying its parent to replace z with NIL as its child.
- If z has just one child, then we elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.
- If z has two children, then we find z's successor y—which must be in z's right subtree—and have y take z's position in the tree. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree. This case is the tricky one because, as we shall see, it matters whether y is z's right child.

In order to move subtrees around within the binary search tree, we define a subroutine TRANSPLANT, which replaces one subtree as a child of its parent with another subtree. When TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child.

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Theorem 12.3
We can implement the dynamic-set operations INSERT and DELETE so that each one runs in $O(h)$ time on a binary search tree of height h. ■

STACKS/QUEUES

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

Figure 10.1 shows the effects of the modifying operations PUSH and POP. Each of the three stack operations takes $O(1)$ time.

The queue has a ***head*** and a ***tail***.

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3       Q.tail = 1
4  else Q.tail = Q.tail + 1

DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

Figure 10.2 shows the effects of the ENQUEUE and DEQUEUE operations. Each operation takes $O(1)$ time.

In a ***circular list***,
the *pre* pointer of the head of the list points to the tail, and the *next* pointer of the tail of the list points to the head.

A ***sentinel*** is a dummy object that allows us to simplify boundary conditions. For example, suppose that we provide with list L an object $L.nil$ that represents NIL

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

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 $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■