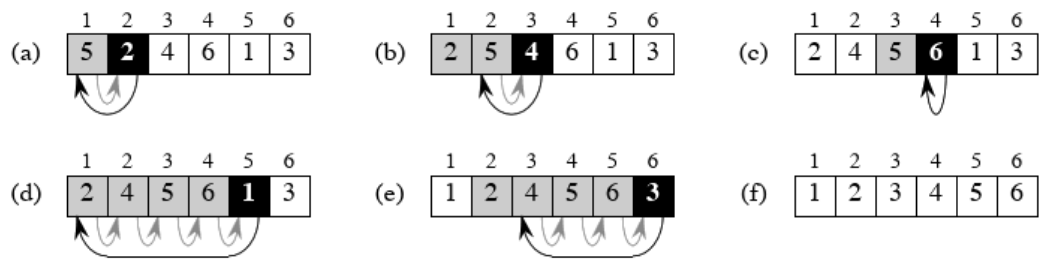
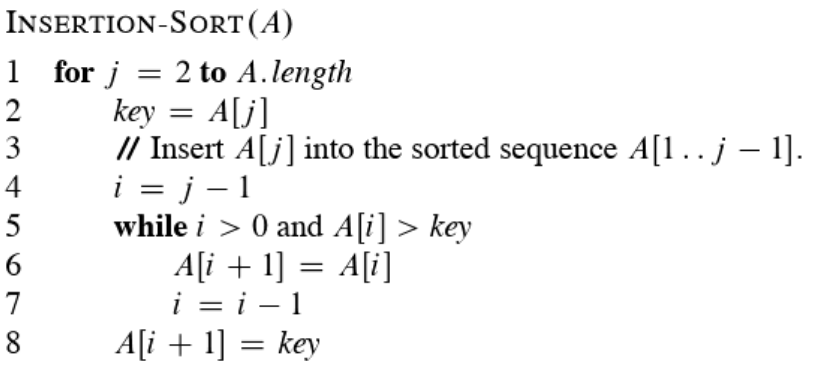
**CS 325 – Module 1 – Foundations and Growth Functions**

* **insertion sort** – takes time roughly equal to *c1 n2*to sort *n* times, where *c1*is a constant that does not depend on *n*. That is, it takes time roughly proportional to *n2*
* **merge sort** – takes time roughly *c2 n* lg *n*, where lg *n* stands for log2 *n* and c2 is another constant that also does not depend on *n*.
  + insertion sort typically has a smaller constant factor than merge sort, so that c1 < c2
* We shall see that the constant factors can have far less of an impact in the running time than the dependence on the input size *n*
  + We can write **insertion sort’s** running time as c1*n ∙ n* and **merge sort’s** running time as c2*n ∙* lg *n*.
  + In doing this, we see where **insertion sort** has a factor of *n* in its running time, **merge sort** has a factor of lg *n*, which is much smaller. (Ex: when *n* = 1000, lg *n* is approx. 10, and when *n* = 1,000,000, lg *n* is approx. 20)
  + although **insertion sort** usually runs faster than merge sort for small input sizes, once the input size *n* becomes large enough, **merge sort’s** advantage of lg *n* vs. *n* will more than compensate for the difference in constant factors. No matter how much smaller c1 is than c2, there will always be a crossover point beyond which **merge sort** is faster.
  + see “concrete example” on page 12
* **Writing insertion sort algorithm**:
  + **Input**: a sequence of *n* numbers (a1, a2, a3 … an)
  + **Output**: a permutation (reordering) of the input of the input sequence such that 
  + Although conceptually we are sorting a sequence, the input comes to us in the form of an array with *n* elements.
  + the pseudocode for insertion sort will take an array as a parameter, *A*[1…n] containing a sequence of length *n* that is to be sorted. (In the code, the number *n* of elements in *A* is denoted by *A.length*)
  + The algorithm sorts the input numbers ***in place***: it rearranges the numbers within the array *A* with at most a constant number of them stored outside the array at any time. The input array *A* contains the sorted output sequence when the insertion sort procedure is finished
  + this sorting method works well with a small number of elements and works similarly to how people may sort a hand of playing cards. You start with an empty left hand and the cards are face down on the table. You remove one card at a time and insert it into the correct position in your left hand. To find the correct position, we compare it with each of the cards already in the left hand, from right to left.





* + 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 that property as a ***loop invariant***
    - we use these ***loop invariants*** to understand why an algorithm is correct
  + We must show three things about **loop invariants**:
    - **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 the algo is correct
  + Lets pair these properties with our recently written insertion sort algo….
    - **Initialization:** we start by showing the loop invariant holds true before the first loop iteration when j = 2 (since j – 1 = 2 – 1 which = 1, this references the first array index position). 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 and shows the loop invariant to be true before the first loop iteration
    - **Maintenance:** Next, we must show 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] (the key) at which point it inserts the value of *A*[j]. 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**.
    - **Termination:** We now 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 is the entire array, we conclude that the entire array is sorted, hence the algo is correct.
* It is obvious that the time taken by and algo of a program grows with the size of the input
* the **running time** of an algo is the number of operations or “steps” executed. It is obvious that the less steps/operations, the better and more efficient the algo is for a particular task.
* **merge sort** algo closely follows the divide and conquer idea
  + **Divide:** divide the n-element sequence to be sorted into two subsequences of n/2 elements each
  + **Conquer:** Sort the two subsequences recursively using merge sort
  + **Combine:** merge the two sorted subsequences to produce the sorted answer
* Generally we want to see the upper bound/worst case run time to guarantee performance/an algo that does work, regardless if it’s the fastest or not
* O(n log n) grows slower than O(n2) (We want algos that grow slower)
* Inductive proofs have 3 elements to them…
  + Base Case
  + Inductive Hypothesis
  + Apply the axiom of induction
* **Asymptotic Notation**
  + **Big-O**: asymptotic “less than”
    - f(n) = O(g(n)) implies that **f(n) g(n)**
  + **Omega Ω:** asymptotic “less than”
    - f(n) = Ω (g(n)) implies that **f(n) ≥ g(n)**
  + **Theta θ:** asymptotic “equality”
    - f(n) = θ(g(n)) implies **f(n) = g(n)**