0.1 Insertion sort

We will start with insertion sort.

Why is insertion sort important? You may have heard it runs slower than other algorithms, such as merge sort. First of all, this is not true in all cases, and I'll get to that later. But the main reason I want to start with insertion sort is because I want to show you how algorithms *should* be analyzed, and it's easier to do that with simpler algorithms.

I do not want you to leave this lecture hall today without knowing

- How to prove that algorithms are correct. This is a broad topic, but today I want to discuss **loop invariants**.
- How to analyze the runtime of algorithms. You may have heard about **big-O notation**, but today I'm going to explain exactly what that means, formally. (It's more subtle than most people think.)

(Unless otherwise stated, you should provide a correctness and runtime proof for every algorithm we ask you to design on your homework.)

Insertion sort is an algorithm for sorting arrays. The following figure (from page 18 of CLRS) shows how insertion sort works. The whole time, you have a "sorted" part of the array (on the left) and an "unsorted" part of the array (on the right). In the beginning, the entire array is unsorted, and the sorted part of the array is just the first element in the array. On each iteration of the outer loop, you extend the sorted part by one element, and move that element to the correct position in the sorted part of the array. Eventually all of the numbers end up in the sorted part and the array is sorted.

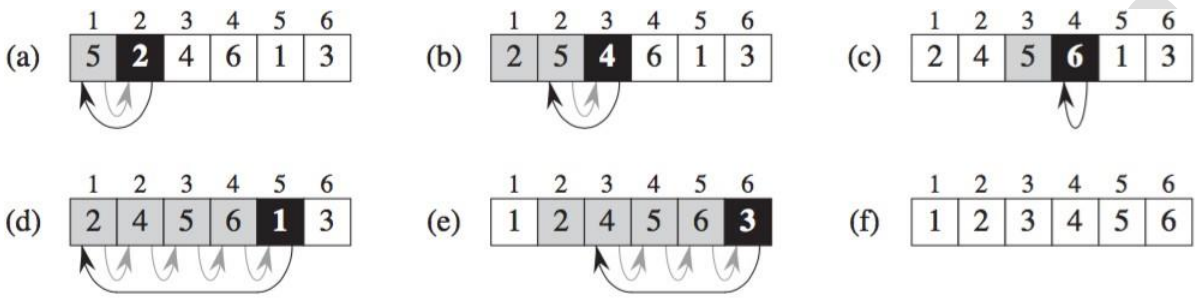

Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

0.2 Assumptions

In order to meaningfully analyze algorithms, we first need to formalize exactly how they work. When doing this we abstract away implementation details that are specific to a particular computer architecture, or a particular programming language. Instead, we make mathematical assumptions about the behavior of computers and base all our analyses on these assumptions. As a result, analyzing algorithms becomes more about proving theorems than actual programming.

For example, we assume that it takes constant time to access any element in an array. This is not necessarily true in practice, since computers have caches, and the time required to access an element on a real computer depends on which previous elements have been accessed. However, the constant time assumption is mathematically convenient for our purposes.

In this class, we will use the "RAM model" of computation unless otherwise specified. This model assumes that the following instructions take constant time: (this list of instructions is from page 23 of CLRS)

- Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling
- Data movement: load, store, copy

- Flow control: conditional and unconditional branch, subroutine call, return

0.3 Formal description of insertion sort

- Input: A sequence of n numbers $(a_1, a_2, \ldots, a_n)$.
- Output: A permutation of those numbers $(a'_1, \ldots, a'_n)$ where $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

```
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
```

0.4 Proof of correctness

In analyzing algorithms, it is important that they do what we say they do (i.e. given an input that meets the specifications, the algorithm produces the specified output). It's so important that often we find ourselves writing proofs that the algorithms are correct. Many of the standard proof techniques apply here, such as proof by contradiction and proof by induction.

To prove insertion sort is correct, we will use "loop invariants." The loop invariant we'll use is

Lemma: At the start of each iteration of the for loop, the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order.

To use this, we need to prove three conditions:

Initialization: The loop invariant is satisfied at the beginning of the for loop.

Maintenance: If the loop invariant is true before the ith iteration, then the loop invariant will be true before the $i+1$st iteration.

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

Note that this is basically mathematical induction (the initialization is the base case, and the maintenance is the inductive step).

In the case of insertion sort, we have

Initialization: Before the first iteration (which is when $j = 2$), the subarray $[1..j-1]$ is just the first element of the array, $A[1]$. This subarray is sorted, and consists of the elements that were originally in $A[1..1]$.

Maintenance: Suppose $A[1..j-1]$ is sorted. 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.

Termination: The condition causing the for loop to terminate is that $j > n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. By the initialization and maintenance steps, we have shown that the subarray $A[1..n+1-1] = A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order.

0.5 Runtime analysis

0.6.1 Running time

The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed. We assume that 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.

Let t_j represent the number of times the while loop test is executed on the jth iteration of the for loop. Then the total runtime is (see figure)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

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	$n-1$

0.6.2 Best, worst, and average case analysis

So if I asked you how long an algorithm takes to run, you'd probably say, "it depends on the input you give it." In this class we mostly consider the "worst-case running time", which is the longest running time for any input of size n. (There is also "best-case" and "average-case".)

For insertion sort, what are the worst and best case inputs? The best case input is a sorted array, because you never have to move elements, and the worst case input is a backwards sorted array, like $[5, 4, 3, 2, 1]$.

In the best case, $t_j = 1$ for all j, and the running time $T(n)$ is a linear function of n. In the worst case, we must compare each element $A[j]$ with each element in the entire sorted subarray $A[1..j-1]$, so $t_j = j$ for all j. Substituting in $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$, we get that $T(n)$ is quadratic in n.

Average-case analysis presupposes a probability distribution on inputs, but if we assume all permutations of a given input array are equally likely, then the average case has quadratic runtime. (We won't prove this here.)

0.6 Asymptotic analysis and big-O notation

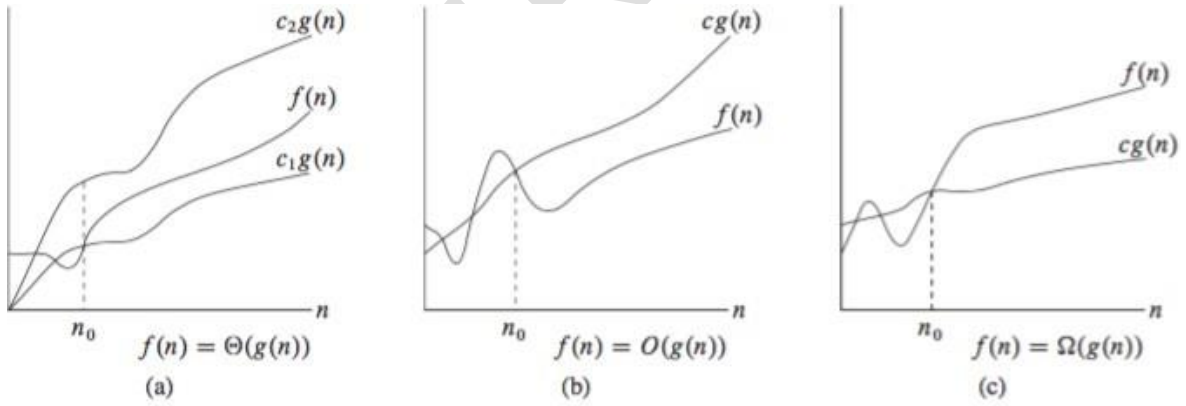

Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

You may have heard runtime described in terms of big-O notation, where linear algorithms

run in time $O(n)$, quadratic algorithms run in time $O(n^2)$, etc. Formally, big-O notation is a property of functions, and not necessarily algorithms. However, since runtimes of algorithms are expressed as functions of the size of the input, it is common to say an algorithm "is $O(n^2)$" when its runtime is in the set of functions known as $O(n^2)$.

Definition: For a given function $g(n)$, $O(g(n))$ is the set of functions $f(n)$ where there exist $c, n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

Colloquially, we say a function $f(n)$ "is $O(g(n))$" if $f(n) \in O(g(n))$. Also, an algorithm runs in $O(g(n))$ if its runtime is in $O(g(n))$.

One thing you may notice from this definition is that big-O time complexity is not nearly as important as people say it is. Some people spend a lot of effort trying to find algorithms that are $O(n^2)$ instead of $O(n^3)$, because they say those algorithms are faster. However, the algorithm with better asymptotic complexity might actually be slower in most practical circumstances.

This is for two reasons.

The first reason is c. Maybe your $O(n^2)$ algorithm runs in $1000000n^2$ for all $n \geq 1$, whereas your $O(n^3)$ algorithm runs in $2n^3$ for $n \geq 1$. Then the $O(n^2)$ algorithm would only start beating the $O(n^3)$ algorithm at large values of n.

The second reason is n_0. The big-O guarantee says nothing about the behavior of the function when $n < n_0$. Your algorithm could potentially take an extremely long time on small inputs and a relatively small time on large inputs, and the time complexity would be relatively small.

For instance, right now I'm running some code to get the followers of users on Twitter. If the user has up to 100000 followers, I call the Twitter API to grab the list of followers, and the amount of time it takes scales linearly with the number of followers. However, if the user has more than 100000 followers, I give up and just return an empty list. This algorithm technically runs in $O(1)$ time (as a function of the number of followers of the user), because big-O notation only describes the behavior of an algorithm on large inputs.

Example: Any algorithm that runs in time $T(n) = n^2 + 2n + 5$ runs in $O(n^2)$. This is because $2n + 5 \leq n^2$ for all $n \geq 4$, so when $n \geq 4$, we can wrote $n^2 + 2n + 5 \leq 2n^2$, which means the equations are satisfied for $c = 2$ and $n_0 = 4$.

Example: Any algorithm that runs in time $T(n) = n^2$ runs in $O(n^2 + 2n + 5)$. This is because $n^2 < n^2 + 2n + 5$ for any nonnegative n, so $c = 1$ and $n_0 = 1$ satisfy the constraints of the definition.

Example: The function $f(n) = \log n$ is $O(n)$. To show this we can show that for $n > 2$, $\log n < n$. First of all, we have $\log 2 = 1 < 2$, so $n - \log n > 0$ for $n = 2$. Second of all, we can take derivatives to find that $n - \log n$ increases for $n \geq 2$ (because the derivative of it with respect to n is $1 - \frac{1}{n}$, which is always greater than 0 for $n \geq 2$). So $n > \log n$ when $n \geq 2$.

(Note that technically you can't take the derivative of f if f is only defined over the integers,

and not over the real numbers. But we can prove the same result by differentiating the extension of f to the real numbers.)

Example: Any function that is $O(n)$ is also $O(n^2)$. Let $f \in O(n)$. Then there exists c, $n_0 > 0$ such that $f(n) \leq cn$ for $n \geq n_0$. But for $n \geq 1$, we have $cn \leq cn^2$. So $f(n) \leq cn^2$ for $n \geq \max(n_0, 1)$.

Definition: For a given function $g(n)$, $\Omega(g(n))$ is the set of functions $f(n)$ where there exist c, $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.

Example: No function that is $O(n)$ can be $\Omega(n^2)$. Suppose $f \in O(n)$. Then there exists $c, n_0 > 0$ such that $f(n) \leq cn$ for all $n \geq n_0$. However, in order for f to be $\Omega(n^2)$, we would need to find c', n_0' such that $f(n) \geq c'n^2$ for all $n \geq n_0'$. This can't work, because for any given value of c' and n_0', you can find an $N > n_0, n_0'$ where $cn < c'n^2$, so $f(n)$ cannot be simultaneously $\leq cN$ and $\geq cN^2$.

Note that a function can be both $O(g(n))$ and $\Omega(g(n))$. In this case, you may want to use different values of c and n_0 when you are proving the O case and the Ω case. If a function is both $O(g(n))$ and $\Omega(g(n))$, we say it is $\Theta(g(n))$.

(We can alternatively define $\Theta(g(n))$ as the set of functions $f(n)$ where there exists $c_1, c_2, n_0 > 0$ such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$.)

Note that if $f(n) \in O(g(n))$, then $g(n) \in \Omega(f(n))$. So in our example, $f(n) = n^2 + 2n + 5$ is actually both $O(n^2)$ and $\Omega(n^2)$, making it $\Theta(n^2)$.