Lecture 3

- What is an algorithm?
- Termination and correctness.
- Non-existence result.
- Time- and space-efficiency, asymptotics
- An(other) example: analysis of binary search.

What is an "algorithm?"

Our informal characterization is that an algorithm is:

> *a step-by-step procedure to solve a problem in finite time.*

The term "algorithm," is from the name of the Persian mathematician, Muhammad Al-Khwarizmi. (He is credited also with being the source of the word "algebra.")

While we adopt the above informal characterization of what an algorithm is, we point out that that is not the only way that an algorithm may be characterized. In particular, the above characterization is based on a procedural, or imperative, perspective. This perspective underlies the encoding of an algorithm in a programming language such as C. One may instead adopt a functional perspective, e.g., via use of the language Haskell, or a logic-based perspective, e.g., via the use of the language Prolog.

Returning to our notion of an algorithm, we observe that in defining it above, we introduce two new terms which, in turn, need to be characterized/defined. These are "problem" and "step." This is a good segue to the following joke. Apparently the "fundamental theorem of software engineering" is: "We can solve any problem by introducing an extra level of indirection." In this context, the new terms are the extra level of indirection. "Oh, you don't know what an 'algorithm' is? All you need to know is what a 'problem' and 'step' are. Oh, you don't know what a 'problem' is? Well, all you need to know is what a 'function' is. Oh, you don't know what a 'function' is. Well, all you need to know is what a 'relation' is. Oh, you don't know what a 'relation' is? All you need to know is..."

We now do our best to clarify as precisely as I think is appropriate for this course, and more broadly in the context of algorithms, what we mean by "solve a problem," and "step."

"Problem" and "solve a problem" In this course, we assume that "solve a problem" means "compute a function." So a "problem" is a function.

Recall that a function, $f : A \to B$, associates a single member from the set B with each member of A. A is called the domain, and B the codomain, of f. The set $R = \{f(x) \mid x \in A\}$ is called the range of f. Of course, $R \subseteq B$.

E.g., if $\mathbb{Z}$ is the set of integers, i.e., $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3 \ldots\}$, then the following function f associates each pair $\langle a, b\rangle \in \mathbb{Z} \times \mathbb{Z}$ with the sum $a + b$.

$$f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, \text{ where } f : \langle a, b\rangle \mapsto a + b$$

We can certainly devise and express an algorithm that computes the above f.

Decision problem A special class of problems of interest to us are decision problems. A decision problem is associated with a function whose codomain is $\{0, 1\}$, or $\{\mathsf{true}, \mathsf{false}\}$.

E.g., consider the following function, which encodes the decision problem of whether a given integer is in a multiset of integers. A multiset, is a set that

allows multiple instances of the same item. The set $\mathcal{S}^{\mathbb{Z}}$ denotes the set of all multisets of integers.

$$f : \mathcal{S}^{\mathbb{Z}} \times \mathbb{Z} \to \{0, 1\}, \quad f : \langle S, i \rangle \mapsto \begin{cases} 0 & \text{if } i \notin S \\ 1 & \text{otherwise} \end{cases}$$

E.g., $f(\langle \{1, 1, 2\}, 5 \rangle) = 0, \quad f(\langle \{-2, 1, -2, -2, 0\}, 1 \rangle) = 1.$

Another way to express the above function is as an input-output pair. Given as input:

- A finite multiset of integers, S, and,
- an integer, i,

The output is 1 if $i \in S$, and 0 otherwise.

Optimization problem An optimization problem is a function that seeks the minimum, or maximum, or largest, or shortest... given some input. For example, given as input an array of integers, output the maximum of those integers. We will deal with discrete optimization problems such as those. An important observation we will make is that in many cases, an optimization problem is no harder, for some notion of "hard," than a decision problem that we can pose from the optimization problem. This, in turn, can help us more clearly identify the computational hardness of the optimization problem, and the best efficiency we can expect from any algorithm.

A "step" What a "step" is depends on our model of computation. We assume the existence of a computer, which is able to execute programs specified in a particular kind of pseudo-code we adopt. The language comprises types of instructions. We assume there are instructions for basic arithmetic operations, ability to define and instantiate variables, loop constructs (for, while) and conditions (if-then). A "step" is an instruction.

Following are (pseudo-code for) algorithms for the above two functions.

```
SUM(i, j)
1 r ← i + j
2 return r
```

```
ISIN(S, i)
1 foreach j ∈ S do
2     if i = j then return 1
3 return 0
```

The above kind of computer is a higher level version of a Turing machine. We do not discuss Turing machines in this course. We simply adopt our somewhat informal, high level model of computation.

Two basic properties of Algorithms

Given our above characterization of an algorithm, when can we say that a proposed algorithm is indeed an algorithm that solves a given problem? We adopt two basic properties: (i) Termination, and, (ii) Correctness.

Termination – this is the question as to whether an algorithm terminates, or halts, for every input. In considering this question, we usually exclude invalid inputs, under the assumption that such inputs can be easily checked for. Another assumption is that we typically consider finite inputs only.

For example, consider the function that maps a non-empty array of integers and an integer to a bit that indicates whether the latter is in the array. The following is a candidate algorithm for this problem.

```
ISINARRAY(A[1, ..., n], i)
1 foreach j from 1 to n do
2     if i = A[j] then return 1
3 return 0
```

In assessing whether ISINARRAY possesses the termination property, we first assume that n is a finite natural number, and therefore, A has finitely many entries only. We assume also that every entry of A is finite, as is the input i. Those are our finiteness assumptions. Our assumption regarding invalid inputs is that n is a natural number, and therefore ≥ 1, and therefore, A is non-empty. Now, we observe first that the **foreach** loop must terminate on any input because n is finite. Each of the lines (2) and (3) itself must

terminate, where the assumption that every entry of A and i are finite is part of the argument for Line (2). Thus, the algorithm possesses the termination property.

We may consider algorithms that do not meet the termination condition. Consider, for example, the following randomized algorithm to determine the index of the median of an input array of n distinct integers, where n is odd. As we discuss below, RANDMEDIAN does not possess the termination property, or "does not terminate," but is correct.

```
RandMedian(A[1, ..., n])
1 while true do
2     i ← uniformly random choice from 1, ..., n
3     c ← 0
4     foreach j from 1 to n do
5         if A[j] < A[i] then c ← c + 1
6     if c = (n−1)/2 then return i
```

Notwithstanding the fact that RANDMEDIAN does not terminate, it may still be considered a "good" algorithm to find the median. A reason is that the expected number of times we iterate in the **while** loop of Line (1) is n. The reason is that each trial in Line (2) is a Bernoulli trial, with a success probability of $1/n$. Thus, RANDMEDIAN does terminate in expectation under the assumption that n is a finite natural number.

Correctness – this is the property that whenever the algorithms halts with some output, that output is what we expect, i.e., as prescribed by the function the algorithm computes. For example, ISINARRAY above is intended to output some $b \in \{0, 1\}$. We deem it to be correct only if whenever it outputs 0, it is indeed the case that $i \notin A[1, \ldots, n]$, and whenever it outputs 1, it is indeed the case that $i \in A[1, \ldots, n]$. (We use the "$\in$" symbol to denote membership in an array, with the expected semantics.) Similarly, the index i that RANDMEDIAN outputs, for RANDMEDIAN to be deemed correct, must be such that $A[i]$ is the median of $A[1], \ldots, A[n]$, and indeed, it is.

Claim 19. ISINARRAY *is correct.*

Proof. We prove by case-analysis.

We first observe that isInArray outputs 0 or 1 only. We seek to prove, on input some $\langle A[1,\ldots,n], i\rangle$: (a) if isInArray outputs 1 then there exists $j \in \{1,\ldots,n\}$ such that $i = A[j]$, and, (b) if isInArray outputs 0, then for all $j \in \{1,\ldots,n\}$, $i \neq A[j]$. We assume in our proof that n is a natural number ≥ 1.

For Case (a), the only line in which we return 1 is Line (2). The return of 1 is preceded by the check "$i = A[j]$." Furthermore, the **foreach** statement in Line (1) guarantees that $j \in \{1,\ldots,n\}$ at any moment that we check the condition "$i = A[j]$" in Line (2). Thus, if we return 1, then we do so in Line (2). Which is true only if $i = A[j]$ at the moment that we return, which is in turn checked only if $j \in \{1,\ldots,n\}$ at that moment. Thus, we have proven Case (a).

For Case (b), we first observe that the only line in which we return 0 is in Line (3). And we reach Line (3) only if we exhaust the **foreach** loop, i.e., do not return in Line (2). If we indeed return in Line (3), this implies that we have exhausted the **foreach** loop, i.e., not returned in Line (2) for any $j \in \{1,\ldots,n\}$. This in turn implies that we have checked the condition "$i = A[j]$" for every $j \in \{1,\ldots,n\}$ because that condition is checked in every iteration of the **foreach** loop. Thus, if we return in Line (3), then for every $j \in \{1,\ldots,n\}$, $i \neq A[j]$, and we have proven Case (b). □

Note: the above proof really does make a mountain out of a molehill, i.e., belabours a rather straightforward proof. I do this for illustration/instructional purposes. You do not have to be so elaborate in your proofs. Just ensure that your logical reasoning is sound.

Claim 20. RandMedian *is correct.*

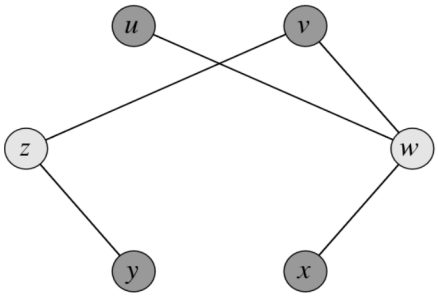
Proof. The only line in which we return is Line (6). And the i that we return is the index we chose in Line (2) in that iteration of the **while** loop. We observe that we return i only if $c = \frac{n-1}{2}$ at the moment we return i. Also, based on Lines (3)–(5), c is the number of entries in $A[1,\ldots,n]$ each of which is $< A[i]$. Thus, at the moment we return i, we know that exactly $(n-1)/2$ entries in $A[1,\ldots,n]$ are smaller than $A[i]$. As n is odd and each entry in

$A[1, \ldots, n]$ is distinct, this means that exactly $(n-1)/2$ entries in $A[1, \ldots, n]$ are larger than $A[i]$. Thus, $A[i]$ is indeed the median of $A[1, \ldots, n]$. □

As with termination, we may design algorithms that are incorrect by design. This is usually done to gain something along another axis, e.g., efficiency. Consider, for example, the problem of determining a minimum-sized vertex cover of an undirected graph.

Definition 2. *Given an undirected graph $G = \langle V, E \rangle$, a vertex cover for it, $C \subseteq V$ is one with the property that for every $\langle u, v \rangle \in E$, it is the case that $u \in C$, or $v \in C$, or both.*

For example, in the following graph, which is from CLRS, each of the sets of dark and light vertices is a vertex cover. That is, $\{u, v, y, x\}$ is a vertex cover, and so is $\{z, w\}$. Of course, any superset of a vertex cover is also a vertex cover.



Consider the following algorithm, ApproxVertexCover, for the minimum-sized vertex cover problem. The input is $G = \langle V, E \rangle$, an undirected graph. The output is $C \subseteq V$, a candidate minimum-sized vertex cover.

```
ApproxVertexCover(G = ⟨V, E⟩)
1 C ← ∅
2 while E ≠ ∅ do
3     Pick some ⟨u, v⟩ ∈ E
4     C ← C ∪ {u, v}
5     Remove all edges incident on either u or v from E
6 return C
```

First, in the following claim, we establish that the set C output by ApproxVertexCover is indeed a vertex cover for the input G.

Claim 21. *If* ApproxVertexCover *terminates, then it outputs a vertex cover for its input* $G = \langle V, E \rangle$.

Proof. The only line in which ApproxVertexCover returns is Line (6), and it returns the set C. Suppose $\langle x, y \rangle$ is an edge in the input undirected graph G. Then, in some iteration of the **while** loop, $\langle x, y \rangle$ was removed from E in Line (5). We observe that in that line, we remove an edge only if at least one of the vertices on which it is incident is added to C. Thus, at least one of x or y is in C on output, which implies that C, on output, is a vertex cover of the input G. □

But the bigger question is: does the algorithm output a minimum-sized vertex cover? And the answer is, 'no, not necessarily.' It is easy to produce an input G for which it does not. Consider, for example, $V = \{u_0, u_1, \ldots, u_{2n-1}\}$, for some positive integer n, and $E = \{\langle u_{2i}, u_{2i+1} \rangle \mid i \in [0, n-1]\}$. Then, the algorithm outputs $C = V$, i.e., a vertex cover of size $2n$. But we know that a minimum-sized vertex cover is of size n. E.g., $\{u_0, u_2, u_4, \ldots, u_{2n-2}\}$ is a minimum-sized vertex cover. Note that the above is a class of inputs, with infinitely many instances of input in it, and this class has inputs of whatever size we want.

So then, what good is this algorithm? It turns out that the vertex cover problem is one that is highly unlikely to lend itself to an efficient algorithm. (We have not characterized "efficient" or "highly unlikely" yet, but we will.) The above algorithm, ApproxVertexCover, however, is simple, and highly efficient. All we have to do is scan the input graph once. Furthermore, even though the algorithm is sub-optimal, we can upper-bound how bad it can get, for every input.

Claim 22. *Suppose* c^* *is the minimum size for a vertex cover of an undirected graph* G*. Then, the output* C *produced by* ApproxVertexCover *on input* G *is guaranteed to be such that:* $|C| \leq 2c^*$.

Proof. Consider every edge, $\langle u, v \rangle$, that we pick in Line (3). At least one of u or v must be in every vertex cover. Thus, the size of the output from

ApproxVertexCover can be no more than twice a minimum-sized vertex cover. □

(Non-)existence of algorithms

Recall that we characterized an algorithm as a procedure that computes a function. We now pose the following basic question: given a function, is an algorithm guaranteed to exist for it? The answer to this question is, 'no.' That is, there exist functions that no algorithm can compute. We now give a proof for this, by construction. We propose a function and then establish that no algorithm can exist that computes the function. Following is the function, which we call *halt*.

Inputs:

- The encoding of an algorithm, call it A which takes as input a string, and,
- A string, call it x.

Output:

- true, if A is guaranteed to eventually halt (terminate) on input x.
- false, otherwise.

For example, if A is isInArray from above, and x is a string that encodes some pair $\langle A[1, \ldots, n], i\rangle$, the correct output from an algorithm, call it B, on input $\langle \text{isInArray}, x\rangle$ is true. On the other hand, B, on input RandMedian and any string x that encodes an array of size three or more odd number of distinct integers, would output false, but on input $\langle \text{RandMedian}, y\rangle$, where y encodes an array of size 1, would output true.

We now establish, via contradiction, that no such algorithm can exist for the function *halt*. For the purpose of contradiction, assume that such an algorithm, call it B, exists. That is, B is a two-input algorithm: the first

input is a string that is the encoding of some algorithm A, and the second input is a string x that is intended to be an input to A. B outputs true if A is guaranteed to halt, when run with input x, and false otherwise. Now consider the following two algorithms, C and D.

```
C(x, y)
1 if B(x, y) = false then return
2 else go into infinite loop
```

```
D(z)
1 C(z, z)
```

That is, C is a two-input algorithm. It interprets the first input string, x, as the encoding of an algorithm, and invokes $B(x, y)$. If $B(x, y)$ outputs false, then $C(x, y)$ halts. Otherwise, it goes in to an infinite loop. All D does on input a string x is invoke $C(x, x)$.

Now consider what happens when we invoke $B(D, D)$. That is, when we invoke B with both its inputs an encoding of the algorithm D.

- $B(D, D) = \mathsf{true} \implies D(D)$ halts $\implies C(D, D)$ halts $\implies B(D, D) = \mathsf{false}$.
- $B(D, D) = \mathsf{false} \implies D(D)$ does not halt $\implies C(D, D)$ does not halt $\implies B(D, D) = \mathsf{true}$.

Thus, we have a contradiction, and no such algorithm, B can exist.

A main point of the above result is that, given a function, we should not naively set out to seek an algorithm for it. Because none may exist. As we will see in this course, a similar issue arises also with seeking efficient algorithms. Given a function, even if an algorithm exists for it, it is sometimes possible to show that seeking an efficient one would be naive.

Algorithm efficiency

Correctness and termination (and existence) are of course rather basic and essential properties. Once we address those, efficiency may be of concern. In this course, we will discuss two kinds of efficiency: time-efficiency, and

space-efficiency. The former deals with how long an algorithm takes on an input. The latter deals with how much additional space, beyond that needed to encode the input, the algorithm allocates.

Time-efficiency We discuss by adopting an example, specifically, the algorithm isInArray above. For time-efficiency of algorithms, we customarily consider three cases: worst-, average-, and best-case for an algorithm. Suppose we start with trying to characterize the worst-case time-efficiency of isInArray on input some $A[1, \ldots, n], i$, where $n, i, A[j] \in \mathbb{N}$ for every $j \in \{1, \ldots, n\}$.

We do it the following way. First we assume that the underlying computer uses some base for its arithmetic, for example, base-2, i.e., binary. We normalize the time the algorithm takes to run to the time for a "basic operation" on a bit – assignment of a bit, addition of one bit to another, subtraction of one bit from another. . . any operation that operates on a bit. We further assume that each such "basic operation" takes some constant-time, say 1 time-unit. Now, we add up all the time taken across all the lines of execution.

We now apply this mindset to characterizing the running-time of isInArray. We have two parameters to consider: n, the number of entries in the array, which is an unbounded, finite natural number, the size of each entry of $A[\cdot]$, and the size of i. Note that we emphasize the *size* here of representing, or *encoding* things. The reason is that when we perform an operation, such as assignment, e.g., $j \leftarrow k$, given our assumption that assignment of 1 bit takes time of 1, it is reasonable to say that if k is $|k|$ bits long, then that assignment takes time $|k|$. Similarly, when we compare, e.g., $i = A[j]$, again representing the number of bits to encode i, or the size of the encoding of i, as $|i|$ and of $A[j]$ as $|A[j]|$, the worst-case time is something like $1 + \min\{|i|, |A[j]|\}$, because a natural algorithm is to compare the two values bit-by-bit, and if all the bits of the two that has fewer bits are the same, we have an additional step to return true or false.

Assume, as a simplification, that every entry in $|A[\cdot]|$, and $|i|$, is each m bits, i.e., the same size. In the worst-case, the **foreach** loop runs n times. And in the final iteration, we either return in Line (2), or the **if** condition fails again, and we return in Line (3) – both those situations elicit the worst-case

running-time. We need to account also for the assignment of each value to j in each iteration. We observe that each time we assign a value to j, that value, and therefore the size we assign, is different. Suppose we make the simplifying assumption that every assignment is for the maximum size j could possibly have. As we assume binary encoding, this is then $1 + \lfloor \log_2 n \rfloor$.

Thus, our worst-case running-time is $n[(m+1) + (1 + \lfloor \log_2 n \rfloor)] + 1$, where the final "$+1$" is to **return** one bit.

In the best-case, we return in Line (2) in the first iteration of the **foreach** loop. And perhaps it is more meaningful to make the assumption that the assignment to j of the value 1 is just an assignment of 1 bit. Thus, our best-case running-time is $1 + (m+1) + 1$.

For the average-case, we need to clearly articulate some underlying assumptions. Suppose, as a simplification, we consider only those inputs in which every entry in $A[\cdot]$ is unique, that every permutation of the entries in $A[1, \ldots, n]$ is equally likely, and that $i \in A[1, \ldots, n]$. The only variable is the number of iterations of the **foreach** loop we execute. We ask what that is, on average, i.e., what the expected case is.

That is, suppose X is a random variable whose value is the number of iterations of the **foreach** loop on some input constrained by the above assumptions. Let $Y_k = I\{i = A[k]\}$; that is, $Y_k = 1$ if we return after exactly k iterations, and $Y_k = 0$ otherwise. Now:

$$
\begin{aligned}
X &= \sum_{k=1}^{n} k \times Y_k \\
\implies E[X] &= \sum_{k=1}^{n} k \times E[Y_k] \\
&= \sum_{k=1}^{n} k \times \Pr\{Y_k = 1\} \\
&= \frac{1}{n} \sum_{k=1}^{n} k \\
&= \frac{n+1}{2}
\end{aligned}
$$

Thus, we expect to return after $\frac{n+1}{2}$ iterations. We now follow the same mind-set as we did for the worst-case to get the final expression for the average-, or expected-case running time.

For RandMedian, the key question in identifying the average-case running time is the number of iterations of the **while** loop, for which the analysis is the same as that we used to identify termination in expectation. As each trial in Line (2) is a Bernoulli trial, we expect to succeed after n trials. Thus, we expect n iterations of the **while** loop before we return in Line (6) with the correct value for the index of the median.

Space-efficiency As we mention before, space-efficiency is a characterization of the space additional to the input an algorithm allocates. For isInArray, this is whatever space is needed for j. Which, as we discuss before, is $1 + \lfloor \log_2 n \rfloor$. In RandMedian, we need to allocate space for i, c and j. Thus, its space-efficiency can be characterized as $3(1 + \lfloor \log_2 n \rfloor)$.

Sub-routines and recursion

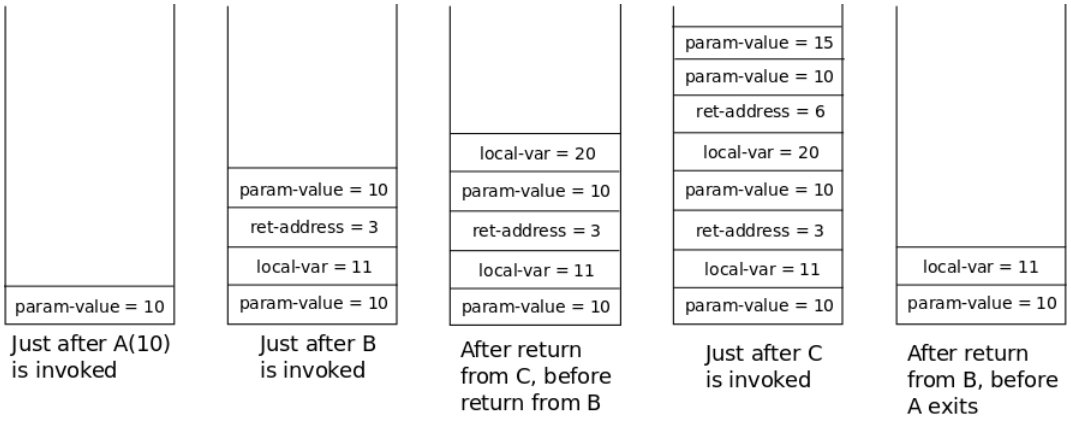
A powerful mechanism that we can use in an algorithm is a sub-routine call, a special case of which is a recursive call, i.e., a sub-routine call to the same sub-routine or algorithm from which we make the call. The use of a sub-routine, and in particular, a recursive call, can subtly impact the time- and space-efficiency of an algorithm. Specifically, we need to take in to consideration what impact the use of such calls has on time- and space-efficiency, if any. For this, we need to adopt, as part of our model of computation, the manner in which such calls are effected when the algorithm runs.

Consider the following example of an algorithm A that invokes another algorithm B as a sub-routine, which in turn invokes an algorithm C as a sub-routine.

```
A(x)                    B(y)                    C(p, q)
1 i ← x + 1             4 i ← y + 10            7 print p, q
2 B(x)                  5 C(y, i − 5)
3 print i               6 print i
```

We assume that the variable i is local to A, and to B. What happens in modern computers is that when a program runs as a process, a call-stack is allocated to it. The call-stack is a stack data structure, which is used to keep track of sub-routine invocations, and local variable values, and to transmit parameter-values.

In our above example, assume that our execution starts at Line (1) in A, for some input (parameter-value) x. Thus, when we are done executing Line (3) in A, the process exits. Just before we execute Line (2), the invocation to B, in A, the call-stack contains two things: the parameter-value, x, and the value of the local variable, i. When we invoke B in Line (2), two things need to happen: (i) we need to transmit the parameter-value to B, and, (ii) we need to remember that the return-address, once we are done executing B, is Line (3) in A. Our computer does this by pushing a tuple of those values on to the call-stack. Similarly, when we invoke C from B, we push its parameter-values, and the return address, which is Line (6) in B, on to the call-stack. The following picture indicates what the contents of the call-stack are at a few different points after we invoke $A(10)$.



Things can get more interesting when the sub-routine calls are recursive. Consider, for example, the following version of insertion sort, an algorithm to sort an array of items from a totally ordered set, e.g., integers, which we call RecursiveInsertionSort.

```
RecursiveInsertionSort(A, n)
 1 if n = 1 then return
 2 RecursiveInsertionSort(A, n − 1)
 3 foreach j from 1 to n − 1 do
 4     if A[n] < A[j] then
 5         tmp ← A[n]
 6         foreach k from n downto j + 1 do
 7             A[k] ← A[k − 1]
 8         A[j] ← tmp
```

The mindset that underlies recursion is *recurrence* – to evaluate $f(x)$ for some value x, if we already have evaluated $f(y)$ for a different value y, then we leverage that to evaluate $f(x)$. For example, the function $f : \mathbb{N} \to \mathbb{Z}_0^+$ where $f : x \mapsto \frac{x(x-1)}{2}$ can be characterized as a recurrence as:

$$f : x \mapsto \begin{cases} 0 & \text{if } x = 1 \\ (x-1) + f(x-1) & \text{otherwise} \end{cases}$$

In RecursiveInsertionSort, if the input array has only one entry, we have nothing to do. This is expressed in Line (1). Otherwise, in Line (2), we make a recursive call to sort all but the last entry of the array. We then "insert" the last entry in the correct location in Lines (3)–(8).

The space-efficiency of RecursiveInsertionSort is worse than that of IterativeInsertionSort below on account of the recursive calls. The underlying logic behind both algorithms is the same. In IterativeInsertionSort, the algorithm maintains the invariant that immediately before the i^{th} iteration of the outer **foreach** loop, $A[1, \ldots, i-1]$ are in sorted order. The time-efficiency of the two algorithms can also be shown to be the same.

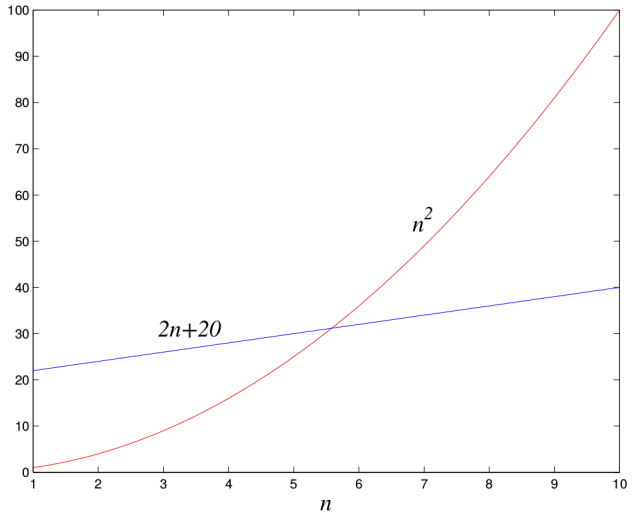
```
IterativeInsertionSort(A, n)
1 foreach i from 2 to n do
2     foreach j from 1 to i − 1 do
3         if A[i] < A[j] then
4             tmp ← A[i]
5             foreach k from i downto j + 1 do
6                 A[k] ← A[k − 1]
7             A[j] ← tmp
```

However, their space-efficiency is different. For IterativeInsertionSort, the call-stack has depth 1, for one tuple that corresponds to the single invocation to the algorithm. For RecursiveInsertionSort, the corresponding worst-case call-depth is n.

Asymptotics

Thus far, we have considered time-efficiency by normalizing to a "basic operation" on each bit, and then counting the total number of such operations. Similarly, for space-efficiency, we counted each bit as a single unit of space. We now abstract the manner in which we characterize time- and space-efficiency. First we motivate this abstraction.

Which running time is better?



Suppose we have two algorithms for a problem, one, call it Algorithm (A), which runs in time $2n + 20$ on input-size n, and another, call it Algorithm (B), which runs in time n^2. A plot of these two functions is shown above in a picture from Dasgupta et al., "Algorithms." Of course, we consider input size $n \in \mathbb{Z}_0^+$ only. And we observe that for $n \leq 5$, Algorithm (B) performs better; otherwise Algorithm (A) does.

Thus, if we ask which of algorithm (A) or (B) is better, the answer is "it depends" on the size of the input. Note that this is different from the case that the functions that capture time- or space-efficiency are more "similar" to one another; for example, it is easier to compare the functions n^2 and $2n^2$ because we can, without qualification, say that an algorithm whose running-time is the former performs better than an algorithm whose running-time is the latter. Thus, one of our motivations is that we seek a meaningful way to compare such "different looking" functions.

Another motivation is "bang for buck," or the relative pay-off we get from investing in a better computer. We explain this motivation after we introduce the mindset behind what I call asymptotics to compare functions.

The mindset behind asympototics is to ask how well an algorithm behaves for large n. That is, a kind of scalability question: how well does the algorithm

scale with large input sizes? Under such a mindset, it is easy to observe that Algorithm (A) is more time-efficient than Algorithm (B). This is exactly the mindset that asymptotics, i.e., "in the limit," captures.

We now explain what this has to do with pay-off or "bang for buck" as I say above. We ask: for which of algorithms (A) and (B) do we get a better pay-off if we were to invest in a better computer? Suppose we fix our time-budget at t time-units. We first ask, for each of algorithms (A) and (B), what size inputs we can handle in time t.

For Algorithm (A), we have $2n_a+20 \leq t \iff n_a \leq \frac{t-20}{2}$. And for Algorithm (B), we have $n_b^2 \leq t \iff n_b \leq \sqrt{t}$. Now suppose we buy resources (e.g., CPU, RAM) for our computer that causes it to be able to execute twice as many instructions in the same time as before. This is equivalent to saying that the same time-budget as before is now $2t$. For Algorithm (A), we are now able to handle, in the same time-budget as before, an input of size $n'_a \approx 2n_a$. For Algorithm (B), we are able to handle a new input size in the same time budget of $n'_b \approx 1.4n_b$, where $1.4 \approx \sqrt{2}$. Thus, the pay off with Algorithm (A) for our investment in the computer is better, and therefore we should deem Algorithm (A) to be the better algorithm.

This is exactly the mindset that a comparison of functions based on asymptotics captures, and is behind the $O(\cdot), \Omega(\cdot), \Theta(\cdot), o(\cdot)$ and $\omega(\cdot)$ notation to compare functions to one another. We now define these. Before we do that, we adopt some assumptions. We deal only with functions of the form $f\colon \mathbb{N} \to \mathbb{R}^+$. That is only those functions whose domain is positive integers, and co-domain is positive reals. The reason is that an input size for us is always a positive integer, and time- and space-efficiency is quantified as a positive real. Furthermore, we consider only functions that are non-decreasing; that is, $a, b \in \mathbb{N}$ with $a \geq b \implies f(a) \geq f(b)$. Again, this is because we do not expect that time- and space-efficiency improves with increasing input size.

Thus, henceforth in this context, when we say "function," we mean a function that satisfies the above assumptions.

Definition 3 $(O(\cdot))$**.** *We say that* $f(n) = O(g(n))$ *if there exists a constant* $c \in \mathbb{R}_0^+$ *such that* $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = c$.

For example, $\lim_{n\to\infty} \frac{50n\log_2 n}{2n^2} = 0$, and therefore $50n\log_2 n = O(2n^2)$. But, $\lim_{n\to\infty} \frac{2n^2}{50n\log_2 n} = \infty$, and therefore $2n^2 \neq O(50n\log_2 n)$. Similarly, $2n+2 = O(n^2)$.

As another example, $\lim_{n\to\infty} \frac{12n^2-5}{n^2+2} = 12$, and therefore $12n^2-5 = O(n^2+2)$. And $\lim_{n\to\infty} \frac{n^2+2}{12n^2-5} = \frac{1}{12}$, and therefore $n^2+2 = O(12n^2-5)$.

The mindset, when we say $f(n) = O(g(n))$ is that $g(n)$ is a kind of upper-bound for $f(n)$. The following alternative definition for $O(\cdot)$, which can be seen as the discrete version of Definition 3.

Definition 4 ($O(\cdot)$, alternative)**.** $O(g(n)) = \{f(n) \mid$ *there exist constants* $n_0 \in \mathbb{N}, c \in \mathbb{R}^+$ *such that for all* $n \geq n_0$, $0 \leq f(n) \leq c \cdot g(n)\}$.

The above alternative definition characterizes $O(\cdot)$ as a set. Thus, it would be appropriate to say, for example, $50n\log_2 n \in O(2n^2)$. But we abuse notion and write $50n\log_2 n = O(2n^2)$.

Just as $O(\cdot)$ characterizes an upper-bound, we can characterize a notion of a lower-bound, and a tight- (upper- and lower-) bound. $\Omega(\cdot)$ is the former, and $\Theta(\cdot)$ is the latter.

Definition 5 ($\Omega(\cdot)$)**.** *We say that* $f(n) = \Omega(g(n))$ *if there exists a constant* $c \in \mathbb{R}_0^+$ *such that* $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$.

Definition 6 ($\Omega(\cdot)$, alternative)**.** $\Omega(g(n)) = \{f(n) \mid$ *there exist constants* $n_0 \in \mathbb{N}, c \in \mathbb{R}^+$ *such that for all* $n \geq n_0$, $0 \leq c \cdot g(n) \leq f(n)\}$.

Claim 23. $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$.

Claim 23 is easy to prove from either set of definitions.

Definition 7 ($\Theta(\cdot)$)**.** $f(n) = \Theta(g(n))$ *if and only if* $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$.

For example, $12n^2 - 5 = \Theta(2n^2 + 1)$. But $12n^2 - 5 \neq \Theta(50n \log_2 n)$. The following figure from CLRS illustrates the notions quite nicely.

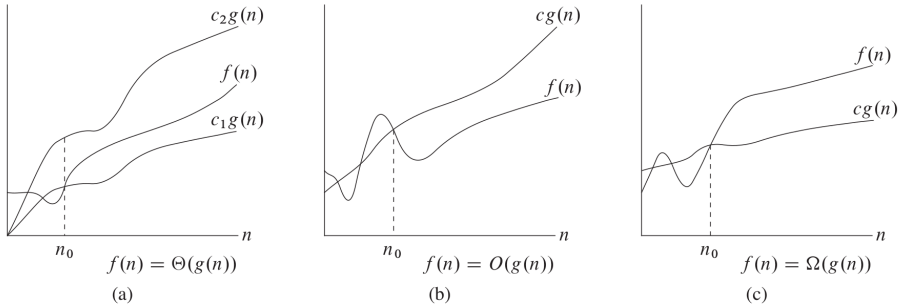


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ 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 $cg(n)$.

Before we move on, we introduce two more notions, $o(\cdot)$, and $\omega(\cdot)$. These are upper- and lower-bounds, respectively, that are not tight.

Definition 8 ($o(\cdot)$)**.** $f(n) = o(g(n))$ *if* $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$.

$o(\cdot)$ is the special case of $O(\cdot)$ in which the limit is exactly 0. The point, when we say $f(n) = o(g(n))$ is to say that $g(n)$ is an asymptotic upper-bound for $f(n)$, but that is <u>not</u> tight. That is, $g(n)$ is "strictly greater than" $f(n)$. For example, $12n^2 - 5 = O(2n^2 + 1)$, but $12n^2 - 5 \neq o(2n^2 + 1)$. And $50n \log_2 n = O(n^2)$, and $5n \log_2 n = o(n^2)$.

Definition 9 ($o(\cdot)$, alternative)**.** $o(g(n)) = \{f(n) \mid$ *for every constant* $c \in \mathbb{R}^+$, *there exists* $n_0 \in \mathbb{N}$ *such that for all* $n \geq n_0$, $0 \leq f(n) < c \cdot g(n)\}$.

Similarly, we have the notion of $\omega(\cdot)$.

Definition 10 ($\omega(\cdot)$)**.** $f(n) = \omega(g(n))$ *if* $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

Definition 11 ($\omega(\cdot)$, alternative)**.** $\omega(g(n)) = \{f(n) \mid$ *for every constant* $c \in \mathbb{R}^+$*, there exists* $n_0 \in \mathbb{N}$ *such that for all* $n \geq n_0$*,* $0 \leq c \cdot g(n) < f(n)\}$.

Claim 24. $f(n) = o(g(n)) \iff g(n) = \omega(f(n))$.

Classes of functions

Our interest will mostly be in four classes of functions, or a mixture of these.

- Exponential. E.g., $f(n) = 2^n$.
- Polynomial. E.g., $g(n) = 2n^2 + 1$.
- Logarithmic. E.g., $h(n) = 1 + \log_2 n$.
- Constant. E.g., $i(n) = 15$.

An example of what we call a "mixture" of these functions is $j(n) = n \log_2 n$. We now relate functions across these classes, and within the classes.

Across classes

Claim 25. *For constants* $a > 1, d \geq 0$*,* $n^d = o(a^n)$.

E.g., $n^{100} = o(2^n)$. Which means also $2^n = \omega(n^{100})$.

Claim 26. *For constants* $a > 1, d > 0$*,* $n^d = \omega(\log_a n)$.

E.g., $\sqrt[30]{n} = \omega(\log_2 n)$. Which means also: $\log_2 n = o(\sqrt[30]{n})$.

Claim 27. *For constants* $a > 1, c > 0$*,* $c = o(\log_a n)$.

E.g., $10^{100} = o(\log_2 n)$.

Within classes

Claim 28. *Let $p(n), q(n)$ be polynomial functions (that satisfy our assumptions) of the same degree. Then, $p(n) = \Theta(q(n))$.*

E.g., $n^2 - 5 = \Theta(2n^2 + 1)$.

Claim 29. *Let $p(n)$ be a polynomial of some degree, d_1, and $q(n)$ be a polynomial of degree d_2 where $d_1 > d_2 > 0$. Then, $p(n) = \omega(q(n))$.*

E.g., $10^{-100} \cdot n^2 = \omega(10^6 \cdot n + 10^{100})$.

Claim 30. *For constants $a > b > 0$, $a^n = \omega(b^n)$. Also, $a^n = o(n^n)$.*

E.g., $2^n = o(3^n)$.

Claim 31. *For constants $a > 1, b > 1$, $\log_a n = \Theta(\log_b n)$.*

E.g., $\log_2 n = \Theta(\log_{16} n)$.

Claim 32. *For constants $a, b > 0$, $a = \Theta(b)$.*

E.g., $15 = \Theta(0.5)$.

Consequence a consequence of the above claims, and the fact that $\Theta(\cdot)$ is an equivalence relation, is that we can choose a succint representative of several functions.

For example, if $f(n)$ is constant in n, we represent this as $f(n) = \Theta(1)$. If $g(n)$ is a polynomial of degree d (and positive, non-decreasing for all positive n), we write this as $\Theta(n^d)$. If $h(n)$ is logarithmic with some base $b > 1$, we write this as $h(n) = \Theta(\lg n)$.

Examples We now consider examples of applying these ideas to analysing the efficiency of algorithms. We begin with insertion sort.

Insertion sort

Consider the following version of insertion sort, from Cormen, et al., "Introduction to Algorithms."

```
InsertionSort_clrs(A[1, ..., n])
1 foreach j from 2 to n do
2     key ← A[j]
3     i ← j − 1
4     while i > 0 and A[i] > key do
5         A[i + 1] ← A[i]
6         i ← i − 1
7     A[i + 1] ← key
```

The mindset behind this version is the same as IterativeInsertionSort Lecture 2. In the **while** loop of $\text{InsertionSort}_{\text{clrs}}$ Lines (4)–(6), we identify a slot for the current entry, which is temporarily stored in *key*, in Line (2). As we determine the slot, we shift entries to the right in Line (5). In the following claim, we prove correctness, albeit for a class of inputs only — those in which every entry in the input array is distinct.

Claim 33. $\text{InsertionSort}_{\text{CLRS}}$ *is correct if the input array A comprises all distinct entries.*

Proof. We first observe that if $n = 1$, i.e., if the input array A has one entry only, the algorithm does nothing. This is correct because a single-entry array is already in sorted order. We now consider an input array of at least two entries.

We prove the following invariant that the **foreach** loop of Line (1) maintains:

> Immediately after we execute the loop, i.e., Lines (2)–(7), for some value of j, the entries $A[1, \ldots, j]$:
>
> (I) all existed in $A[1, \ldots, j]$ on input, and,
>
> (II) are in sorted order.

It is important that we prove both (I) and (II) above. If we omit either, we would not have a meaningful notion of correctness.

We prove by induction on the number of executions of the **foreach** loop of Line (1). For the base case, we have one execution only, and this must be for $j = 2$. Thus, Line (2) is $key \leftarrow A[2]$ and Line (3) is $i \leftarrow 1$. The **while** loop of Line (4) executes at most once because we decrement i by 1 in Line (6). Thus, the **while** loop of Line (4) executes (once) if and only if $A[i] > key$, i.e., $A[1] > A[2]$ at the moment we check that condition. If indeed this is true, then, in Line (5), we assign $A[2] \leftarrow A[1]$, and then, immediately after when we exit the **while** loop, we assign $A[1] \leftarrow key$, because $i + 1$ at that moment is 1. Thus: if indeed $A[1] > A[2]$ on input, we will have swapped $A[1] \leftrightarrow A[2]$, and otherwise, we will have done nothing. In either case, the array is left in sorted (increasing) order, and the set of entries $\{A[1], A[2]\}$ is exactly what was in $A[1]$ and $A[2]$ on input, i.e., we satisfy properties (I) and (II) above.

To prove the step, we first observe that by the induction assumption, immediately after we execute the **foreach** loop with $j = n - 1$, $A[1, \ldots, n - 1]$ satisfies properties (I) and (II) above. For the loop iteration with $j = n$, Line (2) is $key \leftarrow A[n]$, and, Line (3) is $i \leftarrow n - 1$.

We now prove by induction on the number of iterations, call it k, of the **while** loop of Line (4) that immediately after k iterations of the **while** loop: (i) if $i < n - 1$, then $A[i + 2], \ldots, A[n]$ contains exactly the items $A[i + 1], \ldots, A[n - 1]$ in sequence from just before we executed Line (4) for the first time, (ii) if $i < n - 1$, then $key < A[i + 2]$, and, (iii) if $i \geq 1$, then $key > A[i]$. For the base case, $k = 0$, i.e., we do not enter the **while** loop at all. As $n \geq 2$, $i = n - 1 \geq 1 > 0$, and thus, the only reason for us not to enter the while loop is $A[i] \not> key$, i.e., $A[n - 1] < A[n]$, and all of (i)–(iii) are true.

For the step, consider some $k > 0$. Thus, we enter the **while** loop at least once. Consider the very first iteration of the **while** loop. It must be the case that $A[n-1] > key$, i.e., $A[n-1] > A[n]$. Then, in Line (5), we copy $A[n-1]$ to $A[n]$ and decrement i by 1 in Line (6). Thus, at the end of this iteration of the **while** loop, properties (i)–(iii) are true for $i = n - 2$. We then exploit

the induction assumption because the subsequent number of iterations of the **while** loop are strictly fewer.

The final part of the proof regards Line (7). We observe that all Line (7) des is instantiate $A[i+1]$ to what was $A[n]$ on input. As properties (i)—(iii) above are true, executing Line (7) leaves $A[1,\ldots,n]$ satisfying properties (I) and (II) above. □

Now that we have addressed correctness, we ask: what is the time-efficiency of $\textsc{InsertionSort}_{\text{clrs}}$, in $O(\cdot)$ notation?

Before we answer this question, we adopt another piece of mindset. What we do in answering such questions is first identify and argue that it suffices to identify a "hot operation" or "hot spot" in the algorithm that meaningfully characterizes its running-time. For example, for $\textsc{InsertionSort}_{\text{clrs}}$, we argue that the number of executions of Line (5) meaningfully characterizes the time-efficiency or running-time of the algorithm.

Is counting only the number of executions of Line (5) a simplification? Yes, it is. In this case, as an example, this mindset causes us to gloss over the time it takes, for example, to execute Line (5). And that time has to do with the size of $A[i]$, which is a component of the size of the input. That is, the number of executions of Line (5) has nothing to do with the size of $A[i]$, for any i. Rather, it is a function of n, the number of entries in the array, only.

Thus, while we may argue that counting the number of executions of Line (5) is a meaningful characterization of the time-efficiency of $\textsc{InsertionSort}_{\text{clrs}}$, we need to be careful enough to understand what we lose with such a characterization.

We now characterize the number of executions of Line (5). We observe that the outer **foreach** loop runs for $n-1$ iterations. The inner **while** loop runs at most as many times as it takes i to become 0, starting at $j-1$; we observe that i is decremented by 1 in each iteration of the **while** loop in Line (6). Thus, for a particular value of j, Line (5) is executed at most $j-1$ times. Thus, the maximum number of times Line (5) is executed is:

$$\begin{aligned}\sum_{j=2}^{n}\sum_{i=1}^{j-1} 1 &= \sum_{j=2}^{n}(j-1) \\ &= \sum_{j=2}^{n} j - \sum_{j=2}^{n} 1 \\ &= \left(\frac{n(n+1)}{2} - 1\right) - (n-1) \\ &= \frac{n^2}{2} - \frac{n}{2}\end{aligned}$$

Thus, the running time of InsertionSort$_{\text{clrs}}$ can be characterized as, or "is," $O(n^2)$. Note that our analysis above actually identifies that the running time of InsertionSort$_{\text{clrs}}$ is $O(n^2)$ in the worst-case. Indeed, if an algorithm's running-time is $O(f(n))$ in the worst-case, then the running-time of the algorithm, without qualification as to whether it is the worst-, best- or average-case, and therefore in every case, is $O(f(n))$.

Similarly, if the best-case running time of an algorithm is $\Omega(g(n))$, then the running-time of the algorithm is $\Omega(g(n))$.

Before we identify the best-case running-time of InsertionSort$_{\text{clrs}}$, we can ask for a better identification of the worst-case running-time of InsertionSort$_{\text{clrs}}$. Specifically, what is the worst-case running-time of InsertionSort$_{\text{clrs}}$ in $\Theta(\cdot)$ notation?

Whenever possible, we should endeavour to express the running-time, whether it be best-, worst- or average-case, in $\Theta(\cdot)$ notation. Why? Because that provides the highest quality information. For example, it is correct to say that the worst-case running time of InsertionSort$_{\text{clrs}}$ is $O(n^3)$ or even $O(2^n)$. However, it is <u>not correct</u> to say that it is $\Theta(n^3)$, because it is not, as we establish below.

It turns out that the worst-case running-time is $\Theta(n^2)$. To prove this, suppose $T(n)$ is the worst-case running-time of InsertionSort$_{\text{clrs}}$. We have already shown that $T(n) = O(n^2)$. It remains to be shown that $T(n) = \Omega(n^2)$. That is, in the worst-case, the number of times Line (5) runs is lower-bounded asymptotically by n^2. We can prove this by construction, that is, by produc-

ing an input array of n entries for which we are guaranteed that Line (5) is guaranteed to run $j-1$ times, and no fewer, for each value of j.

We observe that this is exactly the case if $A[1,\ldots,n]$ comprises distinct entries that are sorted in reverse. Thus, $T(n) = \Omega(n^2)$.

What about in the best-case? We observe that there exists an input for which the condition "$A[i] > key$" never evaluates to `true`. And this is when $A[\cdot]$, on input, is already sorted. Therefore, to characterize the best-case, counting the number of executions of Line (5) is no longer meaningful. Rather, counting the number of executions of Line (2), we claim, is meaningful. As the **foreach** loop runs unconditionally, Line (2) is executed $\Theta(n)$ times, and therefore $\Theta(n)$ is a meaningful characterization of the best-case running-time of $\textsc{InsertionSort}_{\text{clrs}}$.

How about in the average-, or expected-case? As before, we make some assumptions. Assume that we restrict ourselves to arrays of distinct entries only. And we assume that given n distinct items, every permutation of them is equally likely. Then, if such a permutation is stored in the array $A[1,\ldots,n]$, given two distinct indices i, j, it is equally likely that $A[i] < A[j]$, and $A[i] > A[j]$.

To intuit the number of times Line (5) is executed, adopt a set of indicator random variables $X_{j,i}$, for all $j = 2,\ldots,n$, $i = 1,\ldots,j-1$.

$$X_{j,i} = \begin{cases} 1 & \text{if } A[i] > A[j] \\ 0 & \text{otherwise} \end{cases}$$

For example, if the input array $A = \langle 12, -3, 15, 7, 8\rangle$, and $j = 4$, then $X_{4,1} = 1, X_{4,2} = 0, X_{4,3} = 1$. Now define another set of random variables, m_j, for each $j = 2,\ldots,n$, which is the number of executions of Line (5) for that value of j in the outermost loop. And let m be the random variable that is the total number of executions of Line (5) for this run of the algorithm. Then:

$$
\begin{aligned}
m_j &= \sum_{i=1}^{j-1} X_{j,i} \\
m &= \sum_{j=2}^{n} m_j = \sum_{j=2}^{n} \sum_{i=1}^{j-1} X_{j,i} \\
E[m] &= E\left[\sum_{j=2}^{n} \sum_{i=1}^{j-1} X_{j,i}\right] = \sum_{j=2}^{n} \sum_{i=1}^{j-1} E[X_{j,i}] \\
&= \sum_{j=2}^{n} \sum_{i=1}^{j-1} \Pr\{X_{j,i} = 1\} = \sum_{j=2}^{n} \sum_{i=1}^{j-1} \frac{1}{2} \\
&= \frac{n(n-1)}{4} = \Theta(n^2)
\end{aligned}
$$

Thus, the expected case running time $\textsc{InsertionSort}_{\text{clrs}}$ is asymptotically no different than the worst-case running time, i.e., $\Theta(n^2)$.

We now ask about RecursiveInsertionSort. As RecursiveInsertionSort employs recursion, a meaningful way to start the characterization of its running-time is as a recurrence. Suppose $T(n)$ is the worst-case number of times Line (7) of RecursiveInsertionSort runs across all recursive invocations starting with the first. We keep in mind that the **if** condition of Line (4) is `true` for at most one value of j. For the worst-case, we need to minimize j, as that maximizes the number of times the **foreach** loop of Line (6) iterates. And this is for $j = 1$. Thus:

$$
T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n-1) + \Theta(n) & \text{otherwise} \end{cases}
$$

Where "$\Theta(1)$" stands for "some $f(n)$ where $f(n) = \Theta(1)$" and similarly for $\Theta(n)$. We can solve this recurrence to get a closed-form solution. One way to do this is to first adopt concrete functions for $\Theta(n)$, and then inductively do a kind of string replacement.

Suppose we adopt the function n in place of $\Theta(n)$, and 1 for $\Theta(1)$. Then:

$$\begin{aligned}
T(n) &\longrightarrow T(n-1) + n \\
&\longrightarrow T(n-2) + n - 1 + n \\
&\longrightarrow T(n-3) + n - 2 + n - 1 + n \\
&\quad\vdots \\
&\longrightarrow T(n-(n-1)) + 2 + 3 + \ldots + (n-2) + (n-1) + n \\
&\longrightarrow 1 + 2 + 3 + \ldots + (n-2) + (n-1) + n \\
&= \Theta(n^2)
\end{aligned}$$

Thus, the worst-case running time of RecursiveInsertionSort is no different from that of the other two versions.

Algorithm efficiency and input size

A note of caution regarding time- and space-efficiency of algorithms. We always characterize it as a function of the size of the input. As an example, consider the following algorithm to multiply two natural numbers.

```
MultByAdding(x, y)
1 result ← 0
2 foreach i from 1 to y do
3     result ← result + x
4 return result
```

What is a meaningful characterization of the time-efficiency of this algorithm? The **foreach** loop runs y times. And in each iteration, we perform an addition. What is the running-time of each addition? We can encode the natural number x using $\Theta(\lg x)$ bits. Thus, adding x to any other natural number takes time $\Omega(\lg x)$. Thus, our total running-time is $\Omega(y \lg x)$.

We point out that this function, $y \lg x$, is exponential in the size of the input $\langle x, y \rangle$. The reason is that each of x, y can be encoded with size $\lg x, \lg y$ respectively. And $y = \Theta(2^{\lg y})$. Thus, as a function of the size of the inputs, this is what we would call an exponential-time, and not polynomial-time algorithm.

"Polynomial-time" and "exponential-time" We say that an algorithm is polynomial-time if there exists a constant d such that the algorithm's time-efficiency can be meaningfully characterized as $O(n^d)$, where n is the size of the input. We say that an algorithm is exponential-time if there exists a constant $d > 1$ such that the algorithm's time-efficiency can be meaningfully characterized as $O(d^n)$, where n is the size of the input. Note that sometimes, when we say "polynomial-time" or "exponential-time," we mean "no better than polynomial-/exponential-time," i.e., the running-time of the algorithm is $\Omega(n^d)$ or $\Omega(d^n)$ for some constant $d > 1$, respectively. It is usually clear from context which one we mean.

When our input is an integer, then we need to be particularly cautious, because an integer i can be encoded with $\Theta(\lg i)$ bits only, where the base of the log depends on the encoding adopted by the underlying computer. If the underlying computer adopts unary, i.e., base-1, encoding, then we observe that $\log_1 i = \Theta(i)$. However, if the computer adopts base-x encoding, for $x \in \mathbb{R}, x > 1$, then $\log_x i = o(i)$, and therefore $\log_x i \neq \Theta(i)$.

However, if our input is a set, or an array, or a list, etc., then, given such a data structure of n items, we have no way of encoding it with size any less than $\Theta(n)$. The reason is that we need to encode each item distinctly.

We now return to our problem of multiplying two natural numbers. Does there exist a polynomial-time algorithm for it? The answer is yes. And an algorithm is the grade-school algorithm for multiplication. As an example, the following shows how we would multiply two binary numbers, 11010 and 1001, using that algorithm.

$$
\begin{array}{cccccccc}
 & & & 1 & 1 & 0 & 1 & 0 \\
 & & & & 1 & 0 & 0 & 1 \\
\hline
 & & & 1 & 1 & 0 & 1 & 0 \\
 & & 0 & 0 & 0 & 0 & 0 & \\
 & 0 & 0 & 0 & 0 & 0 & & \\
1 & 1 & 0 & 1 & 0 & & & \\
\hline
1 & 1 & 1 & 0 & 1 & 0 & 1 & 0
\end{array}
$$

As a sanity check, $(11010)_2 = (26)_{10}, (1001)_2 = (9)_{10}$ and $26 \times 9 = (234)_{10} =$

$128 + 64 + 32 + 8 + 2 = (11101010)_2$. The algorithm is as follows, assuming the inputs x, y are encoded in binary.

```
MultGradeSchool(x, y)
1 result ← 0
2 while y > 0 do
3     if least significant bit of y = 1 then
4         result ← result + x
5     x ← x × 2
6     y ← ⌊y/2⌋
7 return result
```

Note that Line (5) is bit-shift to the left, and Line (6) is a bit-shift to the right, i.e., each runs in time $\Theta(1)$. The **while** loop runs for as many bits as there are in y, ignoring leading 0's, i.e., $O(\lg y)$ times. And Line (4) runs in time $O(\lg(xy))$. Thus, the total running-time is $O(\lg y \times (\lg x + \lg y))$. So if we represent the size our input as n, the algorithm runs in time $O(n^2)$. Therefore, this is a polynomial-time algorithm. More specifically, at worst a quadratic-time algorithm.

We can think of the above algorithm as multiplication by "repeated doubling." More specifically, it realizes the following recurrence.

$$xy = \begin{cases} 2x\lfloor y/2 \rfloor & \text{if } y \text{ is even} \\ x + 2x\lfloor y/2 \rfloor & \text{otherwise} \end{cases}$$

Similarly, we can conceive an algorithm for division by "repeated halving," and an algorithm for exponentiation by "repeated squaring." For example, a recurrence for a^b, for $a, b \in \mathbb{N}$ is as follows. This recurrence can then be realized in a straightforward way as an algorithm.

$$a^b = \begin{cases} \left(a^{\lfloor b/2 \rfloor}\right)^2 & \text{if } b \text{ is even} \\ a \cdot \left(a^{\lfloor b/2 \rfloor}\right)^2 & \text{otherwise} \end{cases}$$

Another example: array with repetitions As another example, consider the following situation. We want to store a multiset of natural numbers.

We emphasize multiset; i.e., the same number may occur more than once. We want to compare the following two ways of storing the multiset as an array.

One is to store each occurrence of a number in each slot of the array. The other is to store a pair $\langle \text{value}, \text{number of occurrences} \rangle$ for each distinct number. As an example, suppose our multiset is $\{2, 2, 1, 3, 2, 4, 2, 4\}$. Then Approach (A) is to store this as the array: $\langle 2, 2, 1, 3, 2, 4, 2, 4 \rangle$. Approach (B) would be to store, instead: $\langle \langle 2, 4 \rangle, \langle 1, 1 \rangle, \langle 3, 1 \rangle, \langle 4, 2 \rangle \rangle$. That is, the value 2 occurs 4 times, the value 1 occurs once, and so on.

Is one approach necessarily better than the other?

Here is one way to answer this question. Consider the worst-case for (B) when compared to (A), and ask how bad that case would have been if we had adopted (A) instead, and vice versa. Assume that the size of the multiset is n, and each number is encoded with the same size, m, bits.

The worst-case for (B) when compared to (A) is when every item is distinct, i.e., the multiset is a set. Then, the size with Approach (A) is nm, and with Approach (B) is $nm + n$. The reason is that with each distinct value, we store the value 1, which costs us 1 bit.

The worst-case for (A) when compared to (B) is when every item is the same. Then, under approach (A), we again consume space of nm. Under approach (B), we consume space $m + \lg n$. The reason is that we store one pair, $\langle x, y \rangle$, where x is the single value of size m, and y is the number of times it occurs, which is n, which can be encoded with size $\lg n$.

Thus, the worst-case for (B) when compared to (A) is not so bad: if under Approach (A) we consume space s, under Approach (B) for that same situation, we consume space $O(s)$, because $s = nm$, and $nm + n \leq 2nm = O(nm)$.

However, the worst-case for (A) when compared to (B) is bad. If under Approach (B) we consume space s, under Approach (A), in the worst-case, we consume space $\Theta(2^s)$. This happens when m is constant in n, and in this case, (A) consumes space n, and (B) consumes space $\lg n$ only.

Thus, Approach (B) can be said to be superior to Approach (A).

$\Theta(\cdot), O(\cdot)$ and $\Omega(\cdot)$, and worst-, best- and average-case Another important point regards the somewhat subtle relationship between the notions of $\Theta(\cdot), O(\cdot)$ and $\Omega(\cdot)$, and the notions of "worst-case," "best-case" and "average-case" efficiency. The two are different, but related, and it would certainly be meaningful to use both in the same sentence. E.g., "The time-efficiency of my algorithm is $\Theta(n \log n)$ in the worst-case." Or, "The space-efficiency of my algorithm is $O(n)$ in the best-case." We now discuss this further, to further clarify the relationship between those two sets of notions.

Suppose I owe you \$50. Now someone asks me whether \$30 is a lower-bound on the amount I owe you. Then the answer is 'yes.' This is because the statement, "the amount I owe is lower-bounded by \$30" is exactly the same as "the amount I owe is no less than \$30." Similarly, the statement, "the amount I owe is upper-bounded by \$70" is true. It is the same as, "the amount I owe is no more than \$70."

However, neither \$30 nor \$70 is a tight-bound. A tight-bound is a bound that is simultaneously an upper- and a lower-bound. In this example, a tight-bound on the amount I owe is \$50.

Now suppose, as a different example, we know that I owe you an amount that is between \$40 and \$50, inclusive. Then, \$30 remains a lower-bound on the amount I owe and \$70 remains an upper-bound. Indeed, an amount a is a lower-bound on what I owe if and only if $a \leq 40$. In other words, it is *not true* that, for example, \$60 is a lower-bound on what I owe. Similarly, an amount b is an upper-bound on what I owe if and only if $b \geq 50$.

In the above example that the amount is between \$40 and \$50, no tight-bound exists, unless we further qualify the statement.

The notions of worst- and best-case are such qualifications. Suppose the more I owe you, the worse it is for me. Returning to the knowledge that I owe you between \$40 and \$50, we can say that in the worst-case, I owe you \$50, and in the best-case, I owe you \$40. And these are, respectively, tight-bounds on the worst- and best-case.

The above examples are similar to what we consider in the context of al-

gorithms, except that with algorithms, we typically consider asymptotic bounds, rather than exact bounds like we do in the examples above. And the notations $\Theta(\cdot), O(\cdot)$ and $\Omega(\cdot)$ are used to indicate asymptotic tight-, upper- and lower-bounds, respectively.

For example, consider the following version of linear-search for an item i in an array $A[1, \ldots, n]$.

```
LinSearch(A[1,...,n], i)
1 ret ← false
2 foreach j from 1 to n do
3     if A[j] = i then
4         ret ← true
5 return ret
```

Suppose we measure the time-efficiency, or running-time, of the above algorithm by counting the number of assignments and comparisons. Then, in the worst-case, we have an assignment in Line (1), n assignments in Line (2), n comparisons in Line (3) and an assignment in Line (4). Thus, our total running-time, which is a function of the number of array-entries n, is $2n + 2$. Thus, under that measure of running-time, it is correct to say that in the worst-case, the running-time of LinSearch is $\Theta(n)$. In the worst-case, the running-time is both $O(n)$ and $\Omega(n)$.

Because the running-time in the worst-case is $O(n)$, we can say, without qualification, that the running-time of the algorithm is $O(n)$. That is, an upper-bound in the worst-case is an upper-bound for every case.

What about in the best-case? The best-case, under the measure we have adopted is elicited by an input in which $i \notin A[1, \ldots, n]$, and Line (4) is never executed. Thus, our running-time is $2n + 1$, which is different from $2n + 2$, but still $\Theta(n)$. Thus, in the best-case, the running-time is both $O(n)$ and $\Omega(n)$.

Because the running-time in the best-case is $\Omega(n)$, we can say, without qualification, that the running-time of the algorithm is $\Omega(n)$. That is, a lower-

bound in the best-case is an lower-bound for every case.

Thus, for LinSearch, under the measure for running-time we have adopted, we can drop all qualifications, and simply say that the algorithm's running-time is $\Theta(n)$.

Suppose we change our measure of time-efficiency to the number of executions of Line (4) only, and furthermore, consider only those inputs in which $i \in A[1, \ldots, n]$, i.e., inputs that result in a return value of true only. Then, as we do not preclude the case that every entry in $A[1, \ldots, n]$ is i, in the worst-case, we have n executions of Line (4). And of course, $n = \Theta(n)$. Indeed, even if only some constant multiple of n entries in A are i, e.g., $n/10$ of the entries only, we have the same asymptotic tight-bound on the running-time of $\Theta(n)$.

In the best-case, we have one execution of Line (4) only, and $1 = \Theta(1)$. Indeed, even if the number of instances of i is some constant in n, the best-case has the same asymptotic tight-bound of $\Theta(1)$.

As the asymptotic tight-bounds on the running-time are not the same across cases, specifically, the worst- and best-cases, we no longer can specify a tight-bound on the running-time of the algorithm without qualification. However, we can say that the algorithm runs in time $\Omega(1)$ and $O(n)$; we can say that without the qualifications of best- and worst-time. If we are willing to add the qualification of worst- or best-case, then we indeed have tight bounds. The algorithm runs in time $\Theta(n)$ in the worst-case, and $\Theta(1)$ in the best-case.

As a final clarification, when we say, "the algorithm's time-/space-efficiency in the worst-case is $\Theta(n^2)$," what we mean is: the algorithm's time-/space-efficiency in the worst-case is some function $f(n)$, where $f(n) = \Theta(n^2)$.

An Analysis of Binary Search

We conclude this lecture with a complete example, specifically, an analysis of the termination, correctness and efficiency properties of binary search on a sorted array. We adopt the following version of binary search.

Inputs:

- The array A of integers, sorted non-decreasing, of size n. We assume the array is indexed $0, 1, \ldots, n-1$.
- lo, hi – two indices with the property $0 \leq lo \leq hi \leq n-1$.
- i – the item we are searching for.

Output: the intent is that BinSearch(A, lo, hi, i) outputs true if i is in $A[lo, \ldots, hi]$, and false otherwise. Thus, we can invoke it as BinSearch$(A, 0, n-1, i)$ to check whether i is in A, or constrain lo and hi to check within only a portion of the array.

```
BinSearch(A, lo, hi, i)
1 while lo ≤ hi do
2     mid ← ⌊(lo+hi)/2⌋
3     if A[mid] = i then return true
4     if A[mid] < i then lo ← mid + 1
5     else hi ← mid − 1
6 return false
```

As an example of the working of the algorithm, suppose we invoke BinSearch$(A, 1, 6, 134)$, where $A = \langle 1, 8, 8, 14, 123, 645, 646, 1023 \rangle$. The algorithm progresses as follows.

Iteration #	*lo*	*hi*	*mid*	*condition*	*change*
1	1	6	3	$A[3] = 14 < 134$	$lo \leftarrow 4$
2	4	6	5	$A[5] = 645 > 134$	$hi \leftarrow 4$
3	4	4	4	$A[4] = 123 < 134$	$lo \leftarrow 5$

The algorithm then returns false in Line (6).

Termination of binary search

The termination property is: for every legal input, $\langle A, lo, hi, i \rangle$, BinSearch is guaranteed to halt. By "legal" input, we mean one that meets the constraints we impose on the invoker of BinSearch. That is, A must be a finite

array of finite sorted, non-decreasing integers, of $n \geq 1$ finitely many entries, and lo and hi must be integers that satisfy $0 \leq lo \leq hi \leq n-1$, and i must be a finite integer.

Claim 34. BinSearch *possesses the termination property.*

Proof. We first observe that if BinSearch is invoked with $lo > hi$, then we return immediately in Line (6) as the **while** condition evaluates to false. For the case that BinSearch is invoked with $lo \leq hi$, if we return in Line (3) in that iteration, we are done, as the algorithm has terminated. Now suppose that when we enter an iteration of the **while** loop, we do so with lo, hi values of lo_1, hi_1, respectively. Suppose also that in that iteration, we do not return in Line (3). Thus, we are guaranteed to check the **while** condition again. This second time, suppose that the lo, hi values are lo_2, hi_2 respectively.

We claim that $hi_2 - lo_2 < hi_1 - lo_1$.

To prove this, let $m = \left\lfloor \frac{lo_1 + hi_1}{2} \right\rfloor$. We observe that we need to consider two cases. (1) $lo_2 = m + 1, hi_2 = hi_1$, and, (2) $lo_2 = lo_1, hi_2 = m - 1$.

In case (1):

$$\begin{aligned} hi_2 - lo_2 &= hi_1 - m - 1 \\ &= hi_1 - \left\lfloor \frac{lo_1 + hi_1}{2} \right\rfloor - 1 \\ &< hi_1 - \left(\frac{lo_1 + hi_1}{2} - 1 \right) - 1 \qquad \because \forall x \in \mathbb{R}, x - \lfloor x \rfloor < 1 \\ &= \frac{hi_1 - lo_1}{2} \\ &\leq hi_1 - lo_1 \end{aligned}$$

Similarly, in case (2):

$$\begin{aligned} hi_2 - lo_2 &= m - 1 - lo_1 \\ &= \left\lfloor \frac{lo_1 + hi_1}{2} \right\rfloor - 1 - lo_1 \\ &< \left(\frac{lo_1 + hi_1}{2} + 1 \right) - 1 - lo_1 \\ &= \frac{hi_1 - lo_1}{2} \\ &\leq hi_1 - lo_1 \end{aligned}$$

□

Thus, we have proven that $hi_2 - lo_2 < hi_1 - lo_1$. Thus, if we start out with $lo \leq hi$, then we either return in Line (3) in some iteration, or, if we do not, eventually $hi - lo < 0 \implies lo > hi$, and we exit the **while** loop and terminate.

Correctness of binary search

For BinSearch, correctness can be articulated as follows:

> We say that BinSearch is correct if, for any legal input $\langle A, lo, hi, i \rangle$, both the following are true:
>
> - BinSearch(A, lo, hi, i) returns true implies that i is in $A[lo, \ldots, hi]$, and,
> - BinSearch(A, lo, hi, i) returns false implies that i is not in $A[lo, \ldots, hi]$.

We characterize what we mean by "legal" input in our discussion above on Termination.

We build up to the proof that BinSearch is indeed correct by first proving some properties that BinSearch possesses that are then useful in the proof for correctness.

Claim 35. *Suppose we successfully enter the while loop with some values for lo and hi. Then, in that iteration of the loop, $lo \leq mid \leq hi$.*

Proof. From Line (2) of the algorithm, $mid = \left\lfloor \frac{lo+hi}{2} \right\rfloor$. We carry out the following case-analysis.

Case 1: $lo + hi$ is even. In this case:

$$\begin{aligned} & mid = \frac{lo + hi}{2} \\ \implies & mid \geq \frac{lo + lo}{2} && \because lo \leq hi \\ \implies & mid \geq lo \end{aligned}$$

Similarly, we have:

$$\begin{aligned} & mid = \frac{lo + hi}{2} \\ \implies & mid \leq \frac{hi + hi}{2} && \because lo \leq hi \\ \implies & mid \leq hi \end{aligned}$$

Case 2: $lo + hi$ is odd. In this case:

$$\begin{aligned} & mid = \frac{lo + hi - 1}{2} \\ \implies & mid \geq \frac{lo + lo - 1}{2} && \because lo \leq hi \\ \implies & mid \geq lo - 1/2 \\ \implies & mid \geq lo && \because \text{both } lo \text{ and } mid \text{ are integers} \end{aligned}$$

Similarly, we have:

$$\begin{aligned} & mid = \frac{lo + hi - 1}{2} \\ \implies & mid \leq \frac{hi + hi - 1}{2} && \because lo \leq hi \\ \implies & mid \leq hi - 1/2 \leq hi && \because hi - 1/2 \leq hi \end{aligned}$$

□

Claim 36. *Suppose the values of lo and hi on input are designated $lo^{(in)}$ and $hi^{(in)}$, respectively. Then, the following are all true at any point in the run of the algorithm.*

1. $hi \leq hi^{(in)}$.
2. $lo \geq lo^{(in)}$.
3. $lo > hi^{(in)} \implies lo > hi$.
4. $hi < lo^{(in)} \implies lo > hi$.

Proof. By induction on the number of iterations of the while loop, call it k, on an invocation of BinSearch with input $\left\langle A, lo^{(\text{in})}, hi^{(\text{out})}, i \right\rangle$, where we assume that $1 \leq lo^{(\text{in})} \leq hi^{(\text{in})} \leq n$, with the valid indices in A being $1, \ldots, n$.

Base case: $k = 1$. In this case, the values of lo and hi in the iteration of the while loop are exactly $lo^{(\text{in})}$ and $hi^{(\text{in})}$ respectively. We know, from Claim 35, that in that iteration, $lo^{(\text{in})} \leq mid \leq hi^{(\text{in})}$. Let $lo^{(\text{out})}$ and $hi^{(\text{out})}$ be the values of lo and hi after any changes to those values in this only iteration. Thus, the only possible values for lo are $lo^{(\text{in})}$ and $lo^{(\text{out})}$, and the only possible values for hi are $hi^{(\text{in})}$ and $hi^{(\text{out})}$. Then, we have the following cases:

- We return in Line (3). Then, $hi^{(\text{out})} = hi^{(\text{in})}, lo^{(\text{out})} = lo^{(\text{in})}$, and all the assertions (1)–(4) are true.
- We update lo in Line (4). Then, $hi^{(\text{out})} = hi^{(\text{in})}, lo^{(\text{out})} > lo^{(\text{in})}$, because, from Claim 35, $mid \geq lo^{(\text{in})}$. Thus, the assertions (1) and (2) are true. As this is the only iteration that happens, we know that $lo^{(\text{out})} > hi^{(\text{out})}$. But $hi^{(\text{out})} = hi^{(\text{in})}$, and therefore (3) is true. And (4) is true because the premise, $hi^{(\text{out})} < lo^{(\text{in})}$ is false.
- We update hi in Line (5). Then, $hi^{(\text{out})} < hi^{(\text{in})}, lo^{(\text{out})} = lo^{(\text{in})}$, because, from Claim 35, $mid \leq hi^{(\text{in})}$. We follow a similar reasoning as the above case.

Induction assumption: for $k = p - 1$, for some integer $p \geq 1$, it is the case that if a run of BinSearch undergoes k iterations of the while loop, then, all four assertions (1)–(4) are true.

To prove: for $k = p$, for some integer $p \geq 1$, it is the case that if a run of BinSearch undergoes k iterations of the while loop, then, all four assertions (1)–(4) are true.

Proof: consider the very first iteration. On entry into the while loop for this very first iteration, $lo = lo^{(\text{in})}$ and $hi = hi^{(\text{in})}$. Let the values of lo and hi be $lo^{(\text{out})}$ and $hi^{(\text{out})}$, respectively. Then, exactly as we prove for the base case, these possible values of lo and hi satisfy all the assertions (1)–(4). Once we are done with this iteration of the while loop, we have only $p - 1$ more iterations, which falls under the induction assumption. □

Claim 37 (Corollary to Claims 35 and 36)**.** *At any moment that we have computed a value for mid in a run of* BinSearch *on input* $\left\langle A, lo^{(in)}, hi^{(in)}, i \right\rangle$*, it is the case that* $lo^{(in)} \leq mid \leq hi^{(in)}$*.*

Proof. Any time immediately after we have computed a value for mid, and before subsequent updates to lo or hi, we know the following:

$lo^{(\text{in})} \leq lo$	from Claim 36, (2)
$lo \leq mid$	from Claim 35
$mid \leq hi$	from Claim 35
$hi \leq hi^{(\text{in})}$	from Claim 36, (1)

Thus, $lo^{(\text{in})} \leq mid \leq hi^{(\text{in})}$. □

We now assert and prove one case of correctness.

Claim 38. *If* BinSearch *returns* true *on some input* $\left\langle A, lo^{(in)}, hi^{(in)}, i \right\rangle$*, then* i *is in* $A[lo^{(in)}, \ldots, hi^{(in)}]$*.*

Proof. The only way an invocation of BinSearch returns true is in Line (3). And this happens only if $A[mid] = i$. And by Claim 37, we are guaranteed that $lo^{(\text{in})} \leq mid \leq hi^{(\text{in})}$. □

Before we assert and prove the other case of correctness, we establish another property of the algorithm.

Claim 39. *Suppose, at the time we successfully enter the while loop in some run of the algorithm, the values of lo and hi are designated* $lo^{(in)}$ *and* $hi^{(in)}$*, respectively. Suppose we do not return* true *in Line (3) for this iteration, and when we are done with the iteration, the values of lo and hi are designated* $lo^{(out)}$ *and* $hi^{(out)}$*, respectively. Then, at least one of the following is true: (i)* $lo^{(in)} < lo^{(out)}$ *and* $hi^{(in)} = hi^{(out)}$*, or, (ii)* $hi^{(in)} > hi^{(out)}$ *and* $lo^{(in)} = lo^{(out)}$*.*

Proof. As we do not return true in Line (3), we know that we either (i) update *lo* and leave *hi* unchanged in Line (4), or, (ii) update *hi* and leave *lo* unchanged in Line (5). Suppose (i) happens. Then, $lo^{(\text{out})} = mid + 1$. But from Claim 37, $mid \geq lo^{(\text{in})}$. Therefore, $mid + 1 > lo^{(\text{in})}$, and $lo^{(\text{out})} > lo^{(\text{in})}$. And as we leave *hi* unchanged, $hi^{(\text{in})} = hi^{(\text{out})}$.

If (ii) happens, then $hi^{(\text{out})} = mid - 1$. But from Claim 37, $mid \leq hi^{(\text{in})}$. Therefore, $mid - 1 < hi^{(\text{in})}$, and $hi^{(\text{out})} < hi^{(\text{in})}$. And as we leave *lo* unchanged, $lo^{(\text{in})} = lo^{(\text{out})}$. □

Claim 40. *If* BinSearch *returns* false *on some input* $\left\langle A, lo^{(in)}, hi^{(in)}, i \right\rangle$*, then i is not in* $A[lo^{(in)}, \ldots, hi^{(in)}]$*.*

Proof. We prove by induction on $d = hi^{(\text{in})} - lo^{(\text{in})}$.

Base case: $d = 0$. In this case, $hi^{(\text{in})} = lo^{(\text{in})}$, and all we are checking via the call to BinSearch is whether $A[hi^{(\text{in})}] = i$. As $hi^{(\text{in})} = lo^{(\text{in})}$, we enter the while loop at least once. For the first iteration of the while loop, we have $mid = hi^{(\text{in})} = lo^{(\text{in})}$, and therefore $A[hi^{(\text{in})}] \neq i$, or otherwise, we would have returned true in Line (3) in this very first iteration. Thus, in this case, i is not in $A[lo^{(\text{in})}, \ldots, hi^{(\text{in})}]$, as desired.

Induction assumption: for all $d = 0, 1, \ldots, p-1$, for some integer $p \geq 1$, we assume that BinSearch returns false implies i is not in $A[lo^{(\text{in})}, \ldots, hi^{(\text{in})}]$.

To prove: for $d = p$, it is true that BinSearch returns false implies i is not in $A[lo^{(\text{in})}, \ldots, hi^{(\text{in})}]$.

Proof: $d = p \geq 0$ implies $lo^{(\text{in})} \leq hi^{(\text{in})}$, and we enter the while loop successfully at least once. We do not return true in this first iteration; let $lo^{(\text{out})}$ and

$hi^{(\text{out})}$ be the values of lo and hi, respectively, at the end of this iteration. We consider two cases: (a) this is the only iteration of the while loop for this input, and, (b) there is at least one more iteration of the while loop for this input.

- Case (a): we know that $lo^{(\text{out})} > hi^{(\text{out})}$. We consider the following two sub-cases, which, according to Claim 39, are exhaustive.
  - Sub-case (i): $lo^{(\text{out})} = lo^{(\text{in})}, hi^{(\text{out})} < hi^{(\text{in})}$. Thus, we have $hi^{(\text{out})} = mid - 1 < lo^{(\text{in})} \implies mid \leq lo^{(\text{in})} \implies mid = lo^{(\text{in})}$, because by Claims 36 and 37, $lo^{(\text{in})} \leq lo \leq mid \leq hi \leq hi^{(\text{in})}$, at any point of a run of the algorithm that we are within the while loop.

    Thus, because we did not return true in this iteration, and we adjusted the value of hi, we know that $i < A[lo^{(\text{in})}]$. Thus, i is not in $A[lo^{(\text{in})}, \ldots, hi^{(\text{in})}]$ because A is sorted in non-decreasing order.
  - Sub-case (ii): $lo^{(\text{out})} > lo^{(\text{in})}, hi^{(\text{out})} = hi^{(\text{in})}$. Thus, we have $lo^{(\text{out})} = mid + 1 > hi^{(\text{in})} \implies mid \geq hi^{(\text{in})} \implies mid = hi^{(\text{in})}$, because by Claims 36 and 37, $lo^{(\text{in})} \leq lo \leq mid \leq hi \leq hi^{(\text{in})}$, at any point of a run of the algorithm that we are within the while loop.

    Thus, because we did not return true in this iteration, and we adjusted the value of lo, we know that $i > A[hi^{(\text{in})}]$. Thus, i is not in $A[lo^{(\text{in})}, \ldots, hi^{(\text{in})}]$ because A is sorted in non-decreasing order.
- Case (b): we know that $lo^{(\text{out})} \leq hi^{(\text{out})}$. We consider the following two sub-cases, which, according to Claim 39, are exhaustive.
  - Sub-case (i): $lo^{(\text{out})} = lo^{(\text{in})}, hi^{(\text{out})} < hi^{(\text{in})}$. Thus, we have $hi^{(\text{out})} - lo^{(\text{out})} < hi^{(\text{in})} - lo^{(\text{in})}$, and all future runs of the algorithm are equivalent to an invocation BinSearch$(A, lo^{(\text{out})}, hi^{(\text{out})}, i)$, which falls under the induction assumption.
  - Sub-case (ii): $lo^{(\text{out})} > lo^{(\text{in})}, hi^{(\text{out})} = hi^{(\text{in})}$. Thus, we have $hi^{(\text{out})} - lo^{(\text{out})} < hi^{(\text{in})} - lo^{(\text{in})}$, and all future runs of the algorithm are equivalent to an invocation BinSearch$(A, lo^{(\text{out})}, hi^{(\text{out})}, i)$, which falls under the induction assumption.

□

Efficiency of binary search

BinarySearch can be meaningfully characterized as highly space-efficient. The only space it needs, in addition to that needed to encode the input, is those for lo, hi and mid. As all of these are indices into the array $A[1, \ldots, n]$, the space-efficiency of BinarySearch can be characterized as $\approx 3 \log n$.

As for time-efficiency, the running-time of BinarySearch can be characterized as the number of loop iterations $\times$ the time for each iteration. Any iteration in which we do not return in Line (3) undergoes: (i) the time to compare $lo \leq hi$, (ii) the time to compute and assign mid in Line (2), (iii) the comparison $A[mid] = i$ in Line (3), (iv) the comparison $A[mid] < i$ in Line (4), and (v) the assignment in Line (4) or (5). We can reasonably argue that for each such iteration, the time is the same; some constant $\times \log n$.

We focus now on the number of iterations of the while loop. We first make the following observation about two successive iterations of the loop.

Claim 41. *Suppose we have at least two consecutive iterations of the while loop, and $hi - lo + 1$ at entry in to the first iteration is k_1, and $hi - lo + 1$ at entry in to the second iteration is k_2. Then, $k_2 \leq \lfloor \frac{k_1}{2} \rfloor$.*

Proof. Let the lo, hi values at entry in to the first iteration be lo_1, hi_1 respectively, and lo_2, hi_2 at entry in to the second iteration. We consider the following four cases, which are exhaustive.

1. $lo_1 + hi_1$ is even, $lo_2 > lo_1, hi_2 = hi_1$. Then:

$$\begin{aligned} lo_2 &= \frac{lo_1 + hi_1}{2} + 1 \\ \implies k_2 = hi_2 - lo_2 + 1 \quad &= hi_1 - \left(\frac{lo_1 + hi_1}{2} + 1 \right) + 1 \quad \because hi_2 = hi_1 \\ &= \frac{(hi_1 - lo_1)}{2} \\ &= \frac{k_1 - 1}{2} \leq \left\lfloor \frac{k_1}{2} \right\rfloor \end{aligned}$$

2. $lo_1 + hi_1$ is even, $lo_2 = lo_1, hi_2 < hi_1$. Then:

$$\begin{aligned}
& hi_2 && = \frac{lo_1 + hi_1}{2} - 1 \\
\implies k_2 = hi_2 - lo_2 + 1 \quad & && = \frac{lo_1 + hi_1}{2} - lo_1 - 1 + 1 \quad \because lo_2 = lo_1 \\
& && = \frac{(hi_1 - lo_1)}{2} \\
& && = \frac{k_1 - 1}{2} \leq \left\lfloor \frac{k_1}{2} \right\rfloor
\end{aligned}$$

3. $lo_1 + hi_1$ is odd, $lo_2 > lo_1, hi_2 = hi_1$. Then:

$$\begin{aligned}
& lo_2 && = \frac{lo_1 + hi_1 - 1}{2} + 1 \\
\implies & k_2 && = hi_2 - lo_2 + 1 \\
& && = hi_1 - \left(\frac{lo_1 + hi_1 - 1}{2} + 1 \right) + 1 \qquad \because hi_2 = hi_1 \\
& && = \frac{(hi_1 - lo_1)}{2} + \frac{1}{2} \\
& && = \frac{k_1}{2} \leq \left\lfloor \frac{k_1}{2} \right\rfloor \qquad \because \text{ see below}
\end{aligned}$$

The last inference is true because $lo_1 + hi_1$ is odd $\implies 2 \times hi_1 - (lo_1 + hi_1)$ is odd $\implies hi_1 - lo_1 + 1 = k_1$ is even. Therefore, $\frac{k_1}{2} = \left\lfloor \frac{k_1}{2} \right\rfloor$.

4. $lo_1 + hi_1$ is odd, $lo_2 = lo_1, hi_2 < hi_1$. Then:

$$\begin{aligned}
& hi_2 && = \frac{lo_1 + hi_1 - 1}{2} - 1 \\
\implies & k_2 && = hi_2 - lo_2 + 1 \\
& && = \frac{lo_1 + hi_1 - 1}{2} - 1 - lo_1 + 1 \qquad \because lo_2 = lo_1 \\
& && = \frac{(hi_1 - lo_1)}{2} - \frac{1}{2} \\
& && = \frac{k_1}{2} - 1 \leq \left\lfloor \frac{k_1}{2} \right\rfloor
\end{aligned}$$

□

Claim 42. *Let $t(k)$ be the number of loop iterations that happen in the worst-case in an invocation to* BinSearch *with some input $\langle A, lo, hi, i\rangle$, where $k = hi - lo + 1$. Then:*

$$t(k) = \begin{cases} 0 & \text{if } k \leq 0 \\ t(\lfloor k/2 \rfloor) + 1 & \text{otherwise} \end{cases}$$

Proof. If $k \geq 2$, in the worst-case, we have two or more iterations. And therefore, $t(k) = t(\lfloor k/2 \rfloor) + 1$ from Claim 41. If $k = 0$, then $hi < lo$ on input, and we have no iterations. If $k = 1$, then $hi = lo$, and we are guaranteed, if we do not return true in the iteration, that the next time we check the condition of the while loop, $hi < lo$. □

Claim 43. *If $t(k)$ is characterized as the recurrence in Claim 42, then $t(k) = O(\lg k)$. More specifically, for all integers $k \geq 1$, $t(k) \leq 1 + \log_2 k$.*

Proof. Base case: $k = 1$. Then, by the recurrence, $t(1) = t(\lfloor 1/2 \rfloor) + 1 = t(0) + 1 = 1$. And $1 + \log_2 k = 1 + \log_2(1) = 1 + 0 = 1$.

Step: we consider two cases. In one, k is odd $\implies \lfloor (k+1)/2 \rfloor = (k+1)/2 < k+1$. From the recurrence and the induction assumption, we have $t(k+1) = t((k+1)/2) + 1 \leq 2 + \log_2((k+1)/2) = 2 + \log_2(k+1) - 1 = 1 + \log_2(k+1)$, as desired.

The other case is that k is even $\implies \lfloor (k+1)/2 \rfloor = k/2$. We assume that $k > 0$, because otherwise, $k + 1 = 1$, which is the base case above. From the recurrence and induction assumption, $t(k+1) = t(k/2) + 1 \leq 2 + \log_2(k/2) = 1 + \log_2(k) \leq 1 + \log_2(k+1)$, as desired. □