Chapter 7

The Loop Invariant for Lower Bounds

Time Complexity: The time complexity of a computational problem P is the minimum time needed by an algorithm to solve it.

$$\exists A,\ \forall I,\ [A(I) = P(I) \text{ and } Time(A,I) \leq T_{upper}(|I|)]$$
$$\forall A,\ \exists I,\ [A(I) \neq P(I) \text{ or } Time(A,I) \geq T_{lower}(|I|)]$$

Asymptotic Notation: When we want to bound the running time of an algorithm while ignoring multiplicative constants, we use the following notation.

Greek Letter	Standard Notation	My Notation	Meaning
Theta	$f(n) = \Theta(g(n))$	$f(n) \in \Theta(g(n))$	$f(n) \approx c \cdot g(n)$
BigOh	$f(n) = \mathcal{O}(g(n))$	$f(n) \leq \mathcal{O}(g(n))$	$f(n) \leq c \cdot g(n)$
Omega	$f(n) = \Omega(g(n))$	$f(n) \geq \Omega(g(n))$	$f(n) \geq c \cdot g(n)$

See Chapter 25.

An Upper Bound is an Algorithm: An upper bound for P is obtained by constructing an algorithm A which outputs the correct answer, namely $A(I) = P(I)$, within the bounded time, namely $Time(A,I) \leq T_{upper}(|I|)$, on every input instance I.

An Lower Bound is an Algorithm: Amusingly enough, a lower bound, proving that there is no faster algorithm for the problem, is also obtained by constructing an algorithm, but it is an algorithm to a different problem. The input to this problem is an algorithm A claiming to solve problem P in the required time. The output, as proof that this is false, is an input instance I on which the given algorithm A either does not give the correct answer, namely $A(I) \neq P(I)$ or uses too much time, namely $Time(A,I) \geq T_{lower}(|I|)$.

Read the Appendix: To understand this better you may have to read two chapters in the Appendix: Chapter 22 on how to think of statements with existential and universal quantifiers as a game between two players and Chapter 23.2 on time complexity.

Circular Argument: Proving lower bounds can lead to the following circular argument. Given an arbitrary algorithm A, we must find an input instance I on which A gives the wrong answer. The problem is that you do not know on which input instance the algorithm will give the wrong answer until you know what the algorithm does. But you do not know what the algorithm does until you give it an input instance and run it. This paradox is avoided by stepping through the computation on A one time step at a time, at each step narrowing the search space for I. This makes your algorithm for solving the lower bound problem an iterative algorithm. As such, it needs a loop invariant.

The Loop Invariant Argument:

The Loop Invariant: The loop invariant will be a classic "narrowing the search space" type of loop invariant. It states that we have a set S of input instances on which the algorithm A's knowledge and actions for its first t time steps are identical.

Establishing the Loop Invariant: Initially the set S is some large set of instances that we want to focus on. The loop invariant is trivially established for $t = 0$ because initially the algorithm knows nothing and has done nothing.

Maintaining the Loop Invariant: The loop invariant is maintained as follows. Assume that it is true at time $t-1$. Though we do not know which input instance I from S will ultimately be given to algorithm A, we do know that what A learns during its first $t-1$ time steps is the same independent of this choice. A, knowing what it has learn during during these first $t-1$ steps, but unaware that it has not been given a specific input instance, will then state what action it will do during time step t. What A learns at time t from this action will depend on which instance $I \in S$ A is given. We then partition S based on what A learns and narrow S down to one such part. This maintains the loop invariant, i.e. that we have a set S of input instances on which the algorithm A's knowledge and actions for its first t time steps are identical.

The Measure Of Progress: The measure of progress for our lower bound algorithm is that S does not get too much smaller.

The Exit Condition: The exit condition is then $t = T_{lower}(|I|)$.

Ending: From the loop invariant and the exit condition, we obtain the post condition by finding two input instances I and I' in S for which the computational problem P requires different outputs, namely $P(I) \neq P(I')$. If on instance I, algorithm A either does not give the correct answer, namely $A(I) \neq P(I)$ or uses too much time, namely $Time(A, I) \geq T_{lower}(|I|)$, then the post condition is met. Otherwise, we turn our attention to instance I'. By the loop invariant and the exit condition, the computation of A is identical on the two instances I and I' for the first $T_{lower}(|I|)$ time steps because both I and I' are in S. Hence, their outputs must be identical, namely $A(I) = A(I')$. By our choice of instances, $P(I) \neq P(I')$. Because $A(I) = P(I)$, it follows that $A(I') \neq P(I')$. Again we have found an instance on which A does not give the correct answer and the post condition is met.

Example 7.1 Sorting: We have seen a number of algorithms that can sort N numbers using $\mathcal{O}(N \log N)$ comparisons between the elements, such as Merge, Quick, and Heap Sort. We will prove that no algorithm can sort faster.

Information Theoretic: The lower bounds technique described above does not consider the amount of work that must get done to solve the problem, but the amount of *information* that must be transferred from the input to the output. The problem with these lower bounds is that they are not bigger than linear with respect to the bit size of the input and the output.

$n = \Theta(N \log N)$: At first it may appear that this is a super linear lower bound. However, N is the number of elements in the list. Assuming that the N numbers to be sorted are distinct, each needs $\Theta(\log N)$ bits to be represented for a total of $n = \Theta(N \log N)$ bits. Hence, the lower bound does not in fact say that more than $\Theta(n)$ bit operations when n is the total number of bits in the input instance.

Definition of Binary Operation: Before we can prove that no algorithm exists that quickly sorts, we need to first be very clear about what an algorithm is and what its running time is. This is referred to as a *model of computation.* For this sorting lower bound, we will be very generous. We will allow the algorithm to perform any binary operation. This operation can use any information about the input or about what has already been computed by the algorithm, but the result of the operation is restricted to a Yes/No answer. For example, as is done in merge sort, it could ask whether the i^{th} element is less than the j^{th} element. For a stranger example, it could ask with one operation whether number of odd elements is odd.

Definition of the Sorting Problem P: The standard sorting problem, given N elements, is to output the same N elements in sorted order. To make our life easier, we will define sorting slightly differently. Our sorting problem P given N elements is to output where each element goes in the sorted order. For example, if the input is $I = \langle 19, 5, 81\rangle$, the output would be $\langle 2, 1, 3\rangle$ because the first element in I is second in the sorted order $\langle 5, 19, 81\rangle$, the second is the first, and the third element is the third. Similarly, the output for $I' = \langle 19, 81, 5\rangle$ would be $\langle 2, 3, 1\rangle$. What makes our lives easier in this version of the sorting problem is that the instances $I = \langle 19, 5, 81\rangle$ and $I' = \langle 19, 81, 5\rangle$ have different outputs, while in the standard problem definition, they would both have the output $\langle 5, 19, 81\rangle$. This change is reasonable because in any sorting algorithm needs to learn the order that the elements should be in.

The Initial Set of Instances: Because of the way we modified the sorting problem, the elements being sorted does not matter, only their initial order. Hence, we might as well assume that we are sorting the numbers 1 to N. Let the initial set S of input instance being considered consist of every permutation of these numbers.

Funny Outputs: The fact that it is hard to sort a permutation of the numbers 1 to N is a little counter intuitive. By the standard definition of sorting, the sorted output will always be $\langle 1, 2, 3, \ldots, N\rangle$. But for our new definition of sorting the algorithm must find the order that the numbers appear in the input. What is even more amusing is that according this definition, the output is identical to the input instance. For example, if the input is $I = \langle 2, 1, 3\rangle$, the output would be $\langle 2, 1, 3\rangle$ because the first element in I is second in the sorted order $\langle 1, 2, 3\rangle$, the second is the first, and the third element is the third.

$P(I) \neq P(I')$: Recall that our search ends by finding two input instances I and I' in S for which the computational problem P requires different outputs, namely $P(I) \neq P(I')$. The fact that the output for $I = \langle 2, 1, 3\rangle$ is $\langle 2, 1, 3\rangle$, effectively proves that for each instance $I \in S$, the output of the sorting problem P will be different.

The Measure of Progress: Our measure of progress, as we search for in instance I on which the algorithm A does not work, will be the number $|S|$ of instances still being considered. Initially, because S consists of all permutations of N elements, $|S| = N!$. We will prove that each iteration, S does not decrease by more than a factor of 2. Hence, after t iterations, $|S| \geq \frac{N!}{2^t}$. Be setting $T_{lower}(|I|)$ to be $\log_2(N!) - 1$, we know that in the end we have at least two input instances remaining to be our I and I'.

Math: In $N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$, $\frac{N}{2}$ of the factors are at least $\frac{N}{2}$ and all N of the factors are at most N. Hence $N!$ is in the range $[\frac{N}{2}^{\frac{N}{2}}, N^N]$. Hence $\log N!$ is in the range $[\frac{N}{2} \log \frac{N}{2}, N \log N]$.

Maintaining the Loop Invariant: Assume that the loop invariant is true at time $t-1$ and that S is the set of input instances on which the algorithm A's knowledge and actions for its first $t-1$ time steps are identical. Given this, the action A will perform during time step t is fixed. The model of computation dictates that the result of A's actions is restricted to a Yes/No answer. We then partition S into two sets based on whether this answer on this instance I is Yes or No. We simply narrow S down to the part of S that is larger of the two. Restricting the algorithm to learning only this one answer maintains the loop invariant. Clearly, the larger of two parts has size of at least a half.

The Lower Bound: This completes the lower bound that any algorithm requires at least $\Omega(N \log N)$ binary operations to correctly solve the Sorting problem.

Example 7.2 Binary Search Returning Index: Consider the problem of searching a sorted list of N elements where the output states the index of the key in the list. Binary search solves the problem with $\log_2(N)$ comparisons. We will now prove a matching lower bound.

The Initial Set of Instances: To follow the same technique that we did for sorting, we need a set of legal input instances each of which has a unique output. Now, however, there are now only N possible outputs. Let the initial set of instances be $S = \{I_j \mid j \in [1, N]\}$, where I_j is the input instance searching for the key five within the list that has the first $j-1$ elements zero, the j^{th} element five, and the last $n-j$ elements ten.

The Measure of Progress: Initially, $|S| = N$. As before, S does not decrease by more than a factor of 2 at each iteration. Hence, after t iterations, $|S| \geq \frac{N}{2^t}$. By setting $T_{lower}(|I|)$ to be $\log_2(N) - 1$, we know that in the end we have at least two input instances remaining to be our I and I'.

The Lower Bound: The rest of the lower bound is the same, proving that any algorithm requires at least $\Omega(\log N)$ binary operations to correctly solve the Search a Sorted List problem.

You Have To Look At The Data Lower Bounds: The following lower bounds do not really belong in the iterative algorithms chapter because in these cases we do not find the instances I and I' iteratively. However, the basic idea is the same. These lower bounds say that at least $N' \leq N$ operations are required on an input of size N, because you must look at at least N' of the input values.

Example 7.3 Parity: The easiest example is for the computational problem *parity*. The input consists of n bits and the output simply states whether the number of ones is even or odd.

The Information Theoretic Approach Does Not Work: The information theoretic approach given above allows the model of computation to charge only one time step for any Yes/No operation about the input instance, because it counts only the bits of information learned. However, this does not work for the parity problem. If any Yes/No operation about the input instance is allowed, then the algorithm can simply ask for the parity. This solves the problem in one time step.

Reading the Input: Suppose, on the other hand, the model of computation charges one time step for reading a single bit of the input. (We could even give any additional operations for free.) Clearly, an algorithm cannot know the parity of the input until it has read all of the bits. This proves the lower bound that any algorithm solving the problem requires at least n time. We will see, however, that there is a bug in this argument.

Example 7.4 Multiplexer: The *multiplexer* computational problem has two inputs, an n bit string x and a $\log_2(n)$ bit index i which has the range 1 to n. The output is simply the i^{th} bit of x. As we did for parity, we might give a lower bound of n for this problem as follows. If the algorithm does not read a particular bit of x then it will give the wrong answer when this bit is the required output. This argument is clearly wrong, because the following is correct algorithm that has running time $\log_2(n) + 1$. It reads i and then learns the answer by reading the i^{th} bit of x.

Dynamic Algorithms: Proving a lower bound based on how many bits need to be read is made a little harder because an algorithm is allowed to change which bits it reads based on what it has read before. Given any single instance, the algorithm might read only $n - 1$ of the bits, but which bit is not read depends on the input instance.

Fix One Instance and Flipping a Bit: Before we can know what the algorithm A does, we must give it a specific input instance I. We must choose one. Then we determine the set $J \subseteq [1, n]$ of bits of this instance that are critical, meaning for each $j \in J$ if you flip just the bit j^{th} bit of I but leave the rest of the instance alone, then the answer to the computation problem on this instance changes. We then give a lower bound of $n' = |J|$ on the time required to solve the problem as follows. We run the algorithm on I and see which bits of this instance it reads. If it reads $n' = |J|$ bits then we are done. Otherwise, there is some bit $j \in J$ that the algorithm does not read on this instance. Because the algorithm does not read it, we can flip this bit of the instance without affecting the answer the algorithm gives. We are not allowed to change any of the bits that the algorithm does read because not only may this change the answer that it gives, it might also change which bits it reads. We have made sure, however, that the flipping of this single bit changes the answer to the computation question. Hence, on one of the two input instances, the algorithm must give the wrong answer.

Examples:

Example 7.3' Parity: We now give a formal lower bound of n for the parity problem as follows. Let I be the all zero instance. Let $J = [1, n]$ be the set of all bits of the input. For each $j \in J$, changing the j^{th} bit of I changes the answer from even parity to odd parity. Hence, if the algorithm does not read the j^{th} bit when given instance I, it either gives the wrong answer on instance I or on the instance with this bit flipped.

Example 7.4' Multiplexer: Above we gave a $\log_2(n) + 1$ time algorithm for solving the multiplexer problem. Now we give a matching lower bound. Let I be the instance with $x = 100000$, namely one in its first bit and zero in the rest and with $j = 1$. Let J consist of the $\log_2(n)$ bits of j and the first bit of x. The output of the multiplexer on I is one because this is the j^{th} bit of x. But if you flipping any bit of j, then a different bit of x is indexed and the answer changes to zero. Also if you flip the first bit of x from being a one, then the answer also changes. Hence, if the algorithm does not read one of these bits when given instance I, it either gives the wrong answer on instance I or on the instance with this bit flipped.

Binary Search Returning Yes/No: In Example 7.2 we proved a $\log_2(N)$ lower bound for searching a sorted list of N elements. We are to do the same again. The difference now is that if the key is in the list, the problem returns only the output Yes and if not then No.

The Approach:

The Information Theoretic Approach Does Not Work: Again the information theoretic approach does not work because if any Yes/No operation about the input instance is allowed, then the algorithm can simply ask whether the key is in the list, solving the problem in one time step.

The Set J of Bits to Flip Approach Does Not Work: The initial instance I needs to consist of the key being searched for and some sorted list. Given this, there are not very many elements J that can be changed in order to change the output of the searching problem.

Some Combination: Instead, we will use a combination of the two lower bound approaches.

The Initial Set of Instances: As we did when we proved the lower bound for this problem in Example 7.2, we consider the input instance I_j which is to search for the key five within the list with the first $j-1$ elements zero, the j^{th} element five, and the last $n-j$ elements ten. Unlike before, however, these instances all have the same output Yes. As in the set J of bits to flip approach, let I'_j be the same instance except the j^{th} element is changed to from five to six so that I_j and I'_j have opposite answers. Considering these, let the initial set of instances be $S = \{I_j \mid j \in [1,n]\} \cup \{I'_j \mid j \in [1,n]\}$.

The Standard Loop Invariant: As before, the loop invariant states that we have a set S of input instances on which the algorithm A's knowledge and actions for its first t time steps are identical.

Another Loop Invariant: We have additional loop invariants stating that the current structure of S is $S = \{I_j \mid j \in [j_1,j_2]\} \cup \{I'_j \mid j \in [j_1,j_2]\}$, where $[j_1,j_2]$ is a subinterval of the sorted list of size. More over, $|[j_1,j_2]| \geq \frac{N}{2^t}$.

Maintaining the Loop Invariant: We maintain the loop invariant as follows. Assume that the loop invariant is true for time $t-1$. Let m be the index of the element read at time t by the algorithm on all inputs instances in S.

$m \notin [j_1, j_2]$: If the algorithm reads an element m before our subrange $[j_1,j_2]$, then for all instances in S, this m^{th} element is zero. Similarly, if m is after, then this element is definitely ten. In either case, the algorithm learns nothing that has not already been fixed. The loop invariant is maintained trivially without changing anything.

$m \in [j_1, j_{mid}]$: Let $[j_1,j_{mid}]$ and $[j_{mid}+1,j_2]$ split our subrange in half. If m is in the first half $[j_1,j_{mid}]$, then we set our new subinterval to be the second half $[j_{mid}+1,j_2]$. This narrows our set of instances down to $S = \{I_j \mid j \in [j_{mid}+1,j_2]\} \cup \{I'_j \mid j \in [j_{mid}+1,j_2]\}$. For all instances in this new S, the m^{th} element is zero. The algorithm reads and learns the value zero and proceeds. The loop invariant is maintained.

$m \in [j_{mid}+1, j_2]$: If m is in the second half $[j_{mid}+1,j_2]$, then we set our new subinterval to be the first half and for all instances in the new S, the m^{th} element is ten.

Ending: The exit condition is then $t = T_{lower}(|I|) = \log_2(N)$. From the loop invariant, when we exit our subinterval $[j_1, j_2]$ is size at least $\frac{N}{2^t} = 1$. Let $j = j_1 = j_2$. Our set S still contains the two instances I_j and I'_j. By the definition of these instances, the first requires the answer $P(I_j) =$ Yes and the second $P(I'_j) =$ No. From the loop invariant the computation of A is identical on these instances for the first $T_{lower}(|I|)$ time steps and hence, their outputs must be identical, namely $A(I_j) = A(I'_j)$. Hence, it must give a wrong answer on at least one of them.

The Lower Bound: The rest of the lower bound is the same proving that any algorithm requires at least $\Omega(\log N)$ binary operations to correctly solve the Search a Sorted List problem.

Current State of the Art in Proving Lower Bounds: Lower bounds are *very* hard to prove because you must consider *every* algorithm, no matter how strange or complex. After all, there are examples of algorithms that start out doing very strange things and then in the end magically produce the required output.

Information Theoretic: The technique used here to prove lower bounds does not consider the amount of work that must get done to solve the problem, but the amount of *information* that must be transferred from the input to the output. The problem with these lower bounds is that they are not bigger than linear with respect to the bit size of the input and the output.

Restricted Model: A common method of proving lower bounds is to consider only algorithms that have a restricted structure. My PhD thesis proved lower bounds on the time/space needed to solve $s - t$-connectivity of a graph in a model that only allows pebbles to slide along edges and jump between each other.

General Model: The theory community is just now managing to prove the first non-linear lower bounds on a general model of computation. This is quite exciting for those of us in the field.

Exercise 7.0.3 *How would the lower bound change if a single operation, instead of being only a yes/no question, could be a question with at most r different answers? Here r is some fixed parameter.*

Exercise 7.0.4 *(See solution in Section V) Recall the magic sevens card trick introduced in Section 4.2. Someone selects one of n cards and the magician must determine what it is by asking questions. Each round the magician rearranges the cards into rows and ask which of the r rows the card is in. Give an information theoretic argument to prove a lower bound on the number of rounds t that are needed.*

Exercise 7.0.5 *(See solution in Section V) Suppose that you have n objects that are completely identical except that one is slightly heavier. The problem P is to find the heavier object. You have a scale. A single operation consists of placing any disjoint sets of the objects the two sides of the scale. If one side is heavier then the scale tips over. Give matching upper and lower bounds for this problem.*

Exercise 7.0.6 *Communication Complexity: Consider the following problem: Alice has some object from the M objects $\{I_1, \ldots, I_M\}$ and she must communicate which object she has to Bob by send a string of bits. The string sent will be an* identifier *for the object. The goal is to assign each object a unique identifier so that the longest one has as few bits as possible.*

Exercise 7.0.7 *State and prove a lower bound when instead of bits Alice can send Bob letters from some fixed alphabet* Σ*.*

Exercise 7.0.8 *(See solution in Section V) The* AND *computational problem given* n *bits determines whether at least one of the bits is a one. This is the same as the* game show *problem mentioned in Chapter 21 which requires finding which of the* n *doors contains a prize. The way this differs from the parity problem is that the algorithm can stop as soon as it finds a prize. Give a tight lower bound for this problem. In the lower bound for the parity problem, which initial instances I work? Which ones work for the AND problem?*

Exercise 7.0.9 *Search a Sorted Matrix Problem: The input consists of a real number* x *and a matrix* $A[1..n, 1..n]$ *of* n^2 *real numbers such that each row* $A[i, 1..n]$ *is sorted and each column* $A[1..n, j]$ *is sorted. The goal is to find the maximum array entry* $A[i, j]$ *that is less than or equal to* x*, or report that all elements of* A *are larger than* x*. Give a lower bound on the number of matrix entries that need to be accessed. Exercise 4.4.1 asks for a matching upper bound.*

Exercise 7.0.10 *Consider the problem of determining the smallest element in a max heap. The smallest elements of a max heap must be one of the* $\lceil n/2 \rceil$ *leaves. (Otherwise, there must be a non-leaf that is smaller than one of its descendants, which means the tree is not a max heap.) Thus it is sufficient to search all leaves. Prove a lower bound that searching all the leaves is necessary?*