Concrete models and lower bounds

In this lecture, we will examine some simple, concrete models of computation, each with a precise definition of what counts as a step, and try to get tight upper and lower bounds for a number of problems. Specific models and problems examined in this lecture include:

- The number of comparisons needed to find the largest item in an array,
- The number of comparisons needed to find the second-largest item in an array,
- The number of comparisons needed to sort an array,
- The number of swaps needed to sort an array

Objectives of this lecture

In this lecture, we want to:

- Understand some concrete models of computation (e.g., the comparison model)
- Understand the definition of a *lower bound* in a specific model
- See some examples of how to prove lower bounds in specific models, particularly for sorting and selection problems

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 8.1, Lower bounds for sorting
- DPV, *Algorithms*, Chapter 2.3, Mergesort (Page 59)

1 Terminology: Upper Bounds and Lower Bounds

In this lecture, we will look at (worst-case) upper and lower bounds for a number of problems in several different concrete models. Each model will specify exactly what operations may be performed on the input, and how much they cost. Each model will have some operations that cost a certain amount (like performing a comparison, or swapping a pair of elements), some that are free, and some that are not allowed at all.

Definition: Upper bound

By an *upper bound* of U_n for some problem and some length n, we mean that *there exists an algorithm A that for every input x of length n costs at most U_n*.

A lower bound for some problem and some length n, is obtained by the negation of an upper bound for that n. It says that some upper bound is not possible (for that value of n). If we take the above statement (in italics) and negate it, we get the following. *for every algorithm A there exists an input x of length n such that A costs more than U_n on input x*. Rephrasing:

Definition: Lower bound

By a *lower bound* of L_n for some problem and some length n, we mean that *for any algorithm A there exists an input x of length n on which A costs at least L_n steps.*

These were definitions for a single value of n. Now a function $f : \mathbb{N} \to \mathbb{R}$ is an upper bound for a problem if $f(n)$ is an upper bound for this problem for every $n \in \mathbb{N}$. And a function $g(\cdot)$ is an lower bound for a problem if $g(n)$ is a lower bound for this problem for every n.

The reason for this terminology is that if we think of our goal as being to understand the "true complexity" of each problem, measured in terms of the best possible worst-case guarantee achievable by any algorithm, then an upper bound of $f(n)$ and lower bound of $g(n)$ means that the true complexity is somewhere between $g(n)$ and $f(n)$.

Finally, what is the *cost* of an algorithm? As we said before, that depends on the particular model of computation we're using. We will consider different models below, and show each has their own upper and lower bounds.

One natural model for examining problems like sorting and selection is the comparison model from last lecture, which we recall as follows.

Definition: Comparison Model

In the *comparison model*, we have an input consisting of n elements in some initial order. An algorithm may compare two elements (asking is $a_i < a_j$?) at a cost of 1. Moving the items, copying them, swapping them, etc., is *free*. No other operations on the items are allowed (using them as indices, adding them, hashing them, etc).

2 Selection in the comparison model

2.1 Finding the maximum of n elements

How many comparisons are necessary and sufficient to find the maximum of n elements, in the comparison model of computation?

Claim: Upper bound on select-max in the comparison model

$n-1$ comparisons are sufficient to find the maximum of n elements.

Proof. Just scan left to right, keeping track of the largest element so far. This makes at most $n-1$ comparisons. □

Now, let's try for a lower bound. One simple lower bound is that we have to look at all the elements (else the one not looked at may be larger than all the ones we look at). But looking at all n elements could be done using $n/2$ comparisons, so this is not tight. In fact, we can give a better lower bound:

Claim: Lower bound on select-max in the comparison model

$n-1$ comparisons are necessary for any deterministic algorithm in the worst-case to find the maximum of n elements.

Proof. The key claim is that every item that is not the maximum must lose at least one comparison (by lose, we mean it is compared to another element and is the lesser of the two). Why is this true? Suppose there were two elements a_i and a_j and neither lost a comparison. Suppose without loss of generality that $a_i > a_j$. If the algorithm outputs a_j it is incorrect. Otherwise, if it outputs a_i then we could construct another input that is the same except that a_j is now the maximum (we don't change the relative order of any other elements). On this new input, none of the results of any comparisons change since a_j never lost any comparisons in the first place, so the algorithm, being deterministic, must output the same answer. However, the algorithm is now incorrect. Therefore there must be $n-1$ elements that lose a comparison, and since only one element loses per comparison, a correct algorithm must perform $n-1$ comparisons. □

Since the upper and lower bounds are equal, the bound of $n-1$ is *tight*.

2.2 Finding the second-largest of n elements

How many comparisons are necessary (lower bound) and sufficient (upper bound) to find the second largest of n elements? Again, let us assume that all elements are distinct.

Claim: Lower bound on select-second-max in the comparison model

$n-1$ comparisons are needed in the worst-case to find the second-largest of n elements.

Proof. The same argument used in the lower bound for finding the maximum still holds. □

Let us now work on finding an upper bound. Here is a simple one to start with.

Claim: Upper bound #1 on select-second-max in the comparison model

$2n-3$ comparisons are sufficient to find the second-largest of n elements.

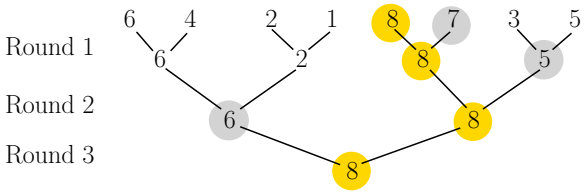
Proof. Just find the largest using $n-1$ comparisons, and then the largest of the remainder using $n-2$ comparisons, for a total of $2n-3$ comparisons. □

We now have a gap: $n-1$ versus $2n-3$. It is not a huge gap: both are $\Theta(n)$, but remember today's theme is tight bounds. So, which do you think is closer to the truth? It turns out, we can reduce the upper bound quite a bit:

Claim: Upper bound #2 on select-second-max in the comparison model

$n+\lg n-2$ comparisons are sufficient to find the second-largest of n elements.

Proof. As a first step, let's find the maximum element using $n-1$ comparisons, but in a tennis-tournament or playoff structure. That is, we group elements into pairs, finding the maximum in each pair, and recurse on the maxima. E.g.,



Now, given just what we know from comparisons so far, what can we say about possible locations for the second-highest number (i.e., the second-best player)? The answer is that the second-best must have been directly compared to the best, and lost.[1] This means there are only $\lg n$ possibilities for the second-highest number, and we can find the maximum of them making only $\lg(n)-1$ more comparisons. □

At this point, we have a lower bound of $n-1$ and an upper bound of $n+\lg(n)-2$, so they are nearly tight. It turns out that, in fact, the lower bound can be improved to exactly meet the upper bound, but the proof is rather complicated so we won't do it for now.

[1] Apparently the first person to have pointed this out was Charles Dodgson (better known as Lewis Carroll!), writing about the proper way to award prizes in lawn tennis tournaments.

2.3 An alternate technique: decision trees

Our lower bound arguments so far have been based on an *adversary* technique. We argued that if an algorithm makes too few comparisons, then we can concoct an input such that it will produce the wrong answer. There are many techniques that can be used to prove lower bounds. Another powerful one are *decision trees*.

A decision tree is a binary tree that represents the behavior of a specific algorithm based on the outcomes of each comparison it makes. Specifically, each internal node corresponds to a comparison such that the left subtree corresponds to the outcome of the comparison being true and the right subtree corresponds to it being false. At a leaf node the algorithm performs no more comparisons and thus is finished and produces an output.

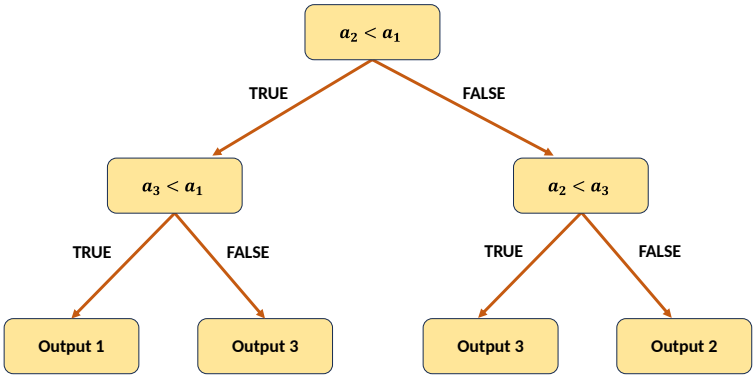
Remark: Decision trees are for particular algorithms

It is very important to remember that a decision tree encodes **a specific algorithm**. Different algorithms will have different decision trees. The decision tree does not however depend on the input to the algorithm, it encodes its behavior on any possible input. In some sense, you can think of the decision tree as a flow chart that tells you exactly what the algorithm does based on the results of the comparisons.

It turns out that decision trees can be a useful tool for analyzing lower bounds. Keeping in mind that a decision tree always represents a particular algorithm, to prove a lower bound, we must argue some property about the structure of *any possible decision tree* for the problem (if we make an argument about a specific decision tree, that is just like arguing about a specific algorithm, which does not help us derive a lower bound for the problem).

Since we are interested in the worst-case number of comparisons, we should observe that the number of comparisons performed by the algorithm on a particular input is the *depth* of the leaf node corresponding to that output. Therefore the worst-case cost (number of comparisons) of the algorithm corresponds exactly to the *longest root-to-leaf path*, i.e., the height of the tree. Therefore, if we can successfully argue about the height of any possible decision tree for a problem, we have an argument for a lower bound!

Here is a decision tree for some arbitrary algorithm that solves the select-max problem.



You can follow it just like a flowchart to determine for any input what index the algorithm will output! We can also use it to argue about lower bounds.

Proof of select-max lower bound via decision trees. At the root node of any decision tree for the select-max problem there are n possible outputs (positions $1 \dots n$). For each comparison, exactly one element loses, and hence the set of possible outputs at each node is one fewer than at its parent node. Therefore all of the leaves of this decision tree have depth $n-1$ and hence $n-1$ comparisons are required to determine the maximum element. □

3 Sorting in the comparison model

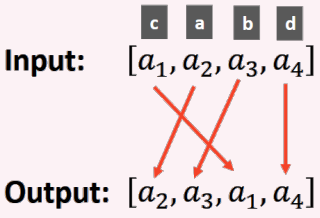
For the problem of *sorting* in the comparison model, the input is an array $a = [a_1, a_2, \dots, a_n]$, and the output is a permutation of the input $\pi(a) = [a_{\pi(1)}, a_{\pi(2)}, \dots, a_{\pi(n)}]$ in which the elements are in increasing order.

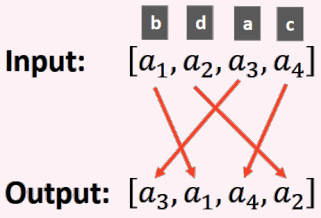
Remark: Correctly defining the "output" of an algorithm

A surprisingly subtle aspect of proving lower bounds, and the source of many buggy or incorrect lower bound proofs is the seemingly simple step of defining what the *output* of the algorithm is supposed to be.

Remember, importantly, that the comparison model has no concept of "values" of the input elements. The *only* thing that an algorithms knows about them are the results of the comparisons. Therefore when a comparison-model sorting algorithm produces an output, it doesn't know the values of the elements, it only knows what order it rearranged them into – so its output can only be described as a permutation of the input elements. Many of our lower bound proofs will be combinatoric in nature, where we will count the number of *required outputs that an algorithm could need to produce.*

For example, suppose we ask an algorithm to sort $[c, a, b, d]$ and $[b, d, a, c]$. Both of these will become $[a, b, c, d]$ when sorted, so does this mean they were the same output? **No!** The former is sorted by outputting $[a_2, a_3, a_1, a_4]$, and the latter is sorted by outputting $[a_3, a_1, a_4, a_2]$, so these are *not the same output.*





On the other hand, suppose we ask to sort both $[c, a, b, d]$ and $[m, d, e, z]$. These will sort to $[a, b, c, d]$ and $[d, e, m, z]$, which are *both* the permutation $[a_2, a_3, a_1, a_4]$. So these are in fact **the same output**, because their elements are in the same relative permuted order, and the actions taken by a deterministic sorting algorithm on them would therefore be

100% identical (the algorithm could not tell the difference between those two inputs.)

When thinking about comparison-model lower bound proofs, be sure to keep this important distinction in mind – values do not matter at all because the algorithm does not know them. It can only deduce/know information about relative order!

Theorem 1: Lower bound for sorting in the comparison model

Any deterministic comparison-based sorting algorithm must perform at least $\lg(n!)$ comparisons to sort n distinct elements in the worst case.[a]

[a] As is common in CS, we will use "lg" to mean "$\log_2$".

In other words, for any deterministic comparison-based sorting algorithm $\mathscr{A}$, for all $n \geq 2$ there exists an input I of size n such that $\mathscr{A}$ makes at least $\lg(n!) = \Omega(n \log n)$ comparisons to sort I.

To prove this theorem, we cannot assume the sorting algorithm is going to necessarily choose a pivot as in Quicksort, or split the input as in Mergesort — we need to somehow analyze *any possible* (comparison-based) algorithm that might exist. This is a difficult task, and its not at all obvious how to even begin to do something like this! We now present the proof, which uses a very nice *information-theoretic* argument. (This proof is deceptively short: it's worth thinking through each line and each assertion.)

Proof of Theorem 1. First remember that we are dealing with *deterministic algorithms* here. Since the algorithm is deterministic, the first comparison it makes is always the same. Depending on the result of that comparison, the algorithm could take different actions, however, critically, **the result of all the previous comparisons always determines which comparison will be made next.** Therefore for any given input to the algorithm, we could write down the sequence of results of the comparisons (e.g., True, False, True, True, False, ...) and this sequence would entirely describe the behavior and hence the output of the algorithm on that input.

Now, in the comparison model, since values are unimportant and only order matters, there are $n!$ different possible input sequences that the algorithm needs to be capable of sorting correctly, one for each possible permutation of the elements. Furthermore, since the elements are distinct, there is only a single correct sorted order, and therefore **every input permutation has a unique output permutation that correctly sorts it**. So, for a comparison-based sorting algorithm to be correct, it needs to be able to produce $n!$ different possible output permutations, because if there is an output it can not produce, then there is an input which it can not sort.

If the algorithm makes ℓ comparisons whose results are encoded by a sequence of binary outcomes (True or False) $b_1, b_2, \ldots, b_\ell$, then since each comparison has only two possible outcomes, the algorithm can only produce 2^ℓ different outputs. Since we argued that in order to be correct the algorithm must be capable of producing $n!$ different outputs, we need

$$2^\ell \geq n! \qquad \Longrightarrow \qquad \ell \geq \lg n!,$$

which proves the theorem. □

Key Idea: Information-theoretic lower bound

The above is often called an "information theoretic" argument because we are in essence saying that we need at least $\lg(M) = \lg(n!)$ bits of information about the input before we can correctly decide which of M outputs we need to produce. This technique generalizes: If we have some problem with M different outputs the algorithm needs to be able to produce, then in the comparison model we have a worst-case lower bound of $\lg M$.

What does $\lg(n!)$ look like? We have:

$$\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ldots + \lg(1) < n\lg(n) = O(n \log n),$$

and

$$\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ldots + \lg(1) > (n/2)\lg(n/2) = \Omega(n \log n).$$

So, $\lg(n!) = \Theta(n \log n)$. However, since today's theme is tight bounds, let's be a little more precise. We can in particular use the fact that $n! \in [(n/e)^n, n^n]$ to get:

$$\begin{array}{rcccl} n\lg n - n\lg e & < & \lg(n!) & < & n\lg n \\ n\lg n - 1.443n & < & \lg(n!) & < & n\lg n. \end{array}$$

Since $1.433n$ is a low-order term, sometimes people will write this fact this as

$$\lg(n!) = (n\lg n)(1 - o(1)),$$

meaning that the ratio between $\lg(n!)$ and $n\lg n$ goes to 1 as n goes to infinity.

How Tight is this Bound? Assume n is a power of 2, can you think of an algorithm that makes at most $n\lg n$ comparisons, and so is tight in the leading term? In fact, there are several algorithms, including:

- *Binary insertion sort.* If we perform insertion-sort, using binary search to insert each new element, then the number of comparisons made is at most $\sum_{k=2}^{n} \lceil \lg k \rceil \leq n\lg n$. Note that insertion-sort spends a lot in moving items in the array to make room for each new element, and so is not especially efficient if we count movement cost as well, but it does well in terms of comparisons.
- *Mergesort.* Merging two lists of $n/2$ elements each requires at most $n-1$ comparisons. So, we get $(n-1) + 2(n/2-1) + 4(n/4-1) + \ldots + n/2(2-1) = n\lg n - (n-1) < n\lg n$.

3.1 An Adversary Argument

A slightly different lower bound argument comes from showing that if an algorithm makes "too few" comparisons, then an adversary can fool it into giving the incorrect answer. Here is a little example. We want to show that any deterministic sorting algorithm on 3 elements must perform at least 3 comparisons in the worst case. (This result follows from the information theoretic lower bound of $\lceil \lg 3! \rceil = 3$, but let's give a different proof.)

If the algorithm does fewer than two comparisons, some element has not been looked at, and the algorithm must be incorrect. So after the first comparison, the three elements are w the winner of the first query, l the loser, and z the other guy. If the second query is between w and z, the adversary replies $w > z$; if it is between l and z, the adversary replies $l < z$. Note that in either case, the algorithm must perform a third query to be able to sort correctly.

3.2 Extra example: Sorting with duplicates

The analysis of sorting with n distinct elements was surprisingly simple because we were able to characterize all of the possible inputs as all $n!$ permutations which all required a distinct output, and therefore argue that any correct algorithm therefore must be able to produce $n!$ distinct outputs. Most of the time it will not be this simple and we will need to take some extra steps. Here's a problem to demonstrate:

Problem: Sorting with D distinct elements

Suppose you have an array of n elements $a_1, \ldots, a_n$ and a parameter D such that you are guaranteed that there are at most D distinct elements in the array (where $1 \leq D \leq n$.)

When $D = n$, it is the original sorting problem from before, which has a lower bound of $\Theta(n \log n)$, so this generalizes the previous problem by allowing duplicates in a constrained way. For $D = 1$, the array would consist of copies of a single element, which could be sorted in zero comparisons since it would be already sorted. For $D = 2$, we could sort the array in linear cost by scanning over the array and grouping the elements of the first value and second value. So it appears that the problem is cheaper as D gets smaller which makes sense; the fewer distinct elements, the more possible sorted orders there are so fewer outputs are required.

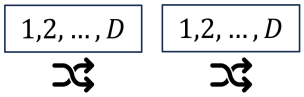
What makes this problem tricky is that it is very unclear how to count exactly how many required outputs there are. It is no longer true that each input element requires a distinct output; for example, both the arrays $[a, a, b]$ and $[a, b, b]$ are sorted by the identity permutation (they are already sorted). So it is **not** the case that we can just count the number of possible inputs and assume that it is equal to the number of required outputs, since a single output could sort multiple inputs. There are also multiple valid outputs for a single input since duplicates can be interchanged without violating sorted order.

Finding a hard subset of inputs A powerful technique that we will use to overcome this issue is to focus on a *specific subset of inputs* to the problem. If we find some family I of inputs and prove a lower bound on the cost to solve any input from I, then of course that lower bound

also applies to solving the whole problem (all possible inputs). What properties do we want this family to have? Ideally two things:

- **It needs to require a lot of outputs**: The information-theoretic lower bound uses the number of required outputs, so we need a subset that requires a lot, otherwise we will get a very weak (low) lower bound.
- **It needs to be simple enough to count**: Since we are required to count the number of required outputs, our family of inputs should be simple enough that we can actually count that number! If our construction is too complicated, it will be too hard, so we usually try to construct something that is easy to describe and count with combinatorics tools that we have (factorials, binomials, powers, etc).

Constructing a family with the distinctness property The vanilla sorting problem was nice because every input required a distinct output, which made counting the number of outputs equivalent to counting the number of inputs. A common technique is therefore to try to construct our family of inputs so that it too has this property. We don't *always* have to do this, we could instead construct a family of inputs and then try to reason about *how many* outputs sort each input, and then divide the total number of outputs by that. For now, we will use the first technique. An important fact to keep in mind is that a permutation of distinct elements always has a unique inverse, i.e., it requires a distinct permutation to sort it. So, how can we generalize that idea to arrays with duplicates? What if we just glue two permutations together?



In other words, we take the elements $1,\ldots,D$ twice, then we randomly shuffle the first half and the second half independently. This gives us an array consisting of two copies each of $1,\ldots,D$, but with the extra property that there is only one 1 in the first half and one in the second half, and so on for each element.

An important fact about this construction is that given two different arrays generated by this process, **the same output can not sort both of them**. This is because if two elements on one side were in different positions, then the output permutation would sort those elements into the wrong order because they are unique!

Constructing our family of inputs Generalizing the above idea, given n and D, we can construct a family of inputs by taking n/D independently shuffled permutations of $1\ldots D$ and concatenating them together (if D does not divide n, the last group might stop early and not contain all of $1\ldots D$, that's fine). By the reasoning above, every input in this family **requires a distinct output**, i.e., no one output can correctly sort two of these inputs. So, the number of requires outputs to sort everything in this set is equal to the number of inputs in this family!

It remains to just count how many inputs are in this family. In each contiguous chunk containing $1\ldots D$, there are $D!$ possible orders, and there are n/D chunks. So, the total number of

inputs in this family is

$$(D!)^{\frac{n}{D}}.$$

Obtaining the lower bound Applying the information-theoretic lower bound, any algorithm in the comparison model for solving this problem therefore requires

$$\log_2\left((D!)^{\frac{n}{D}}\right) = \frac{n}{D}\log_2(D!) = \frac{n}{D}\Theta(D\log D) = \Theta(n\log D)$$

comparisons! This intuitively makes sense, since if $D = n$ we get $\Theta(n\log n)$ which we should, since that is the problem from earlier of just sorting n distinct elements, and if D is smaller, the cost goes down. For example, if $D = 1$, then $\log_2(D) = 0$ which is correct since it takes no comparisons to sort an input consisting of entirely duplicates (it is already sorted).

As an exercise, try to come up with an algorithm that solves this problem in $\Theta(n\log D)$ comparisons, which proves that this bound is asymptotically tight.

4 Sorting in the exchange model

Consider a shelf containing n unordered books to be arranged alphabetically. In each step, we can swap any pair of books we like. How many swaps do we need to sort all the books? Formally, we are considering the problem of *sorting* in the *exchange model*.

Definition: The Exchange Model

In the **exchange model**, an input consists of an array of n items, and the only operation allowed on the items is to swap a pair of them at a cost of 1 step. All other work is free: in particular, the items can be examined and compared to each other at no cost.

Question: how many exchanges are necessary (lower bound) and sufficient (upper bound) in the exchange model to sort an array of n items in the worst case?

Claim: Upper bound on sorting in the exchange model

$n-1$ exchanges is sufficient.

Proof. For this we just need to give an algorithm. For instance, consider the algorithm that in step 1 puts the smallest item in location 1, swapping it with whatever was originally there. Then in step 2 it swaps the second-smallest item with whatever is currently in location 2, and so on (if in step k, the kth-smallest item is already in the correct position then we just do a no-op). No step ever undoes any of the previous work, so after $n-1$ steps, the first $n-1$ items are in the correct position. This means the nth item must be in the correct position too. □

But are $n-1$ exchanges necessary in the worst-case? If n is even, and no book is in its correct location, then $n/2$ exchanges are clearly necessary to "touch" all books. But can we show a better lower bound than that?

Claim: Lower bound on sorting in the exchange model

In fact, $n-1$ exchanges are necessary, in the worst case.

Proof. Here is how we can see it. Create a graph in which a directed edge (i, j) means that that the book in location i must end up at location j. An example is given in Figure 1.

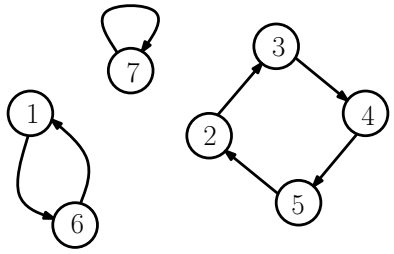


Figure 1: Graph for input `[f c d e b a g]`

This is a special kind of directed graph: it is a permutation — a set of cycles. In particular, every book points to *some* location, perhaps its own location, and every location is pointed to by exactly one book. Now consider the following points:

1. What is the effect of exchanging any two elements (books) that are in the same cycle?

   Answer: Suppose the graph had edges (i_1, j_1) and (i_2, j_2) and we swap the elements in locations i_1 and i_2. Then this causes those two edges to be replaced by edges (i_2, j_1) and (i_1, j_2) because now it is the element in location i_2 that needs to go to j_1 and the element in i_1 that needs to go to j_2. This means that if i_1 and i_2 were in the same cycle, that cycle now becomes two disjoint cycles.

2. What is the effect of exchanging any two elements that are in different cycles?

   Answer: If we swap elements i_1 and i_2 that are in different cycles, then the same argument as above shows that this merges those two cycles into one cycle.

3. How many cycles are in the final sorted array?

   Answer: The final sorted array has n cycles.

Putting the above 3 points together, suppose we begin with an array consisting of a single cycle, such as $[n, 1, 2, 3, 4, \ldots, n-1]$. Each operation at best increases the number of cycles by 1 and in the end we need to have n cycles. So, this input requires $n-1$ operations. □