Number 11

DECISION TREES AND LOWER BOUNDS ON THE COMPLEXITY OF PROBLEMS

1. The complexity of problems

So far we have dealt with the analysis of (worst-case time) complexity of specific data structures and algorithms. In these notes we address a different, but related, question: the worst-case complexity of a *problem*, as opposed to algorithm. The relation between an algorithm and a problem is that an algorithm is a particular way of solving a problem. For example, binary search is a particular algorithm that solves the problem of searching a sorted list; sequential search is a another algorithm that solves the same problem. Heap sort is a particular algorithm that solves the problem of sorting and so is selection sort. The Euclidean algorithm is a solution to the problem of finding the greatest common divisor of two natural numbers. Dijkstra's algorithm solves the problem of finding the shortest path between two nodes in a graph. In each case, we notice that there may be many different algorithms that solve a given problem.

Before we clarify what we mean by the complexity of a problem, we review what we know about the complexity of an algorithm. (Unless otherwise specified, in these notes by "complexity" we mean "worst-case time complexity".) The complexity of an algorithm A is a function $C_A : \mathbf{N} \to \mathbf{N}$, where $C_A(n)$ is the maximum number of steps that A takes on inputs of size n. This means that

(a) for every input of size n, A requires no more than $C_A(n)$ steps; and

(b) for some input of size n, A requires at least $C_A(n)$ steps.

An *upper bound* on the complexity of A is any function $U : \mathbf{N} \to \mathbf{N}$ such that $U(n) \geq C_A(n)$, for all n. To prove that U is an upper bound on the time complexity of A it is necessary and sufficient to prove that

(c) for every input of size n, A requires at most $U(n)$ steps.

A *lower bound* on the complexity of A is any function $L : \mathbf{N} \to \mathbf{N}$ such that $L(n) \leq C_A(n)$, for all n. To prove that L is a lower bound on the time complexity of A it is necessary and sufficient to prove that

(d) for some input of size n, A requires at least $L(n)$ steps.

Of course if a function F is both an upper and a lower bound on the complexity of A, then this function is exactly the complexity of A, i.e. $F(n) = C_A(n)$, for all n.

Now we want to define the complexity of a problem. Before we do this we must make more precise what we mean by "problem". We can formalise this by defining a problem as a binary relation (in the mathematical sense) between inputs and outputs. More precisely, we have a set of inputs I and a set of outputs O; a problem Π is a binary relation between I and O, i.e. a subset of $I \times O$. If $(x, y) \in \Pi$, we say that *y is an output of Π on x*. We define a problem as a *relation* between inputs and outputs, rather than as a *function* from inputs to outputs because, in general, there may be several acceptable outputs for a given input. For instance, if the problem is to find a prime divisor of a given number then for input 6 there are two acceptable outputs, 2 and 3. For the same reason we say that y is *an* output, rather than *the* output, of Π on x.

The problems of interest in computer science are those that admit algorithmic solutions; i.e., those for which we can write a program (say, in Turing) which, given a representation of the input will compute a representation of a corresponding output. We say that an algorithm A *solves* a problem Π, if for every input x, A computes an output of Π on x.†

† You may, at this point, be inclined to assume that *any* problem can be solved algorithmically.

Now we can define the complexity of a problem Π: it is the complexity of the *best* algorithm that solves Π. That is, the *complexity of a problem* Π is a function $C_\Pi : \mathbf{N} \to \mathbf{N}$, such that

(a) there is an algorithm A that solves Π whose complexity is C_Π (i.e., $C_A(n) = C_\Pi(n)$, for all n); and

(b) the complexity of *any* algorithm B that solves Π is at least C_Π (i.e., $C_B(n) \geq C_\Pi(n)$, for all n).

An *upper bound* on the complexity of a problem Π is any function U, such that $U(n) \geq C_\Pi(n)$, for all n. To prove that U is an upper bound on C_Π, it is necessary and sufficient to prove that

(c) there is some algorithm that solves Π whose complexity is at most $U(n)$.

A *lower bound* on the complexity of a problem Π is any function L, such that $L(n) \leq C_\Pi(n)$, for all n. To prove that L is a lower bound on C_Π, it is necessary and sufficient to prove that

(d) any algorithm that solves Π has complexity at least $L(n)$ (or, equivalently, there is no algorithm that solves Π with complexity less than $L(n)$).

From this it follows that to prove a lower bound we must give an argument that covers all possible algorithms for solving the problem that anyone could ever construct. It is *not* enough to give an argument that covers all algorithms we happen to know! Obviously, this is not a trivial task.

When investigating the complexity of a problem Π, we try to find an upper bound U and a lower bound L on the complexity of the problem. In the happy case where $U = L$ this function is exactly the complexity of Π. This is a very pleasant situation indeed: The fact that we have established the upper bound means that we have an algorithm that can solve the problem in that time. The fact that we have established the lower bound means that no algorithm can solve the problem in less time. Thus, we have an algorithm which is optimal — at least from the point of view of worst-case time complexity, which is the usual measure of efficiency. Unfortunately, this happy case is a rather rare occurrence: usually we cannot establish matching upper and lower bounds for a problem. In that case we try to find upper and lower bounds that are as close to each other as we can.

As with the complexity of algorithms, it is sometimes difficult to derive exact bounds on the complexity of a problem and we are content with asymptotic bounds. We say that U_a is an *asymptotic upper bound* on the complexity of Π if there is an algorithm that solves the problem whose complexity is in $O(U_a)$. Analogously, L_a is an *asymptotic lower bound* on the complexity of Π if the complexity of *every* algorithm that solves the problem is in $\Omega(L_a)$. If $U_a = L_a = F$, we say that F is an *asymptotic tight bound* on the complexity of Π and we have that $C_\Pi \in \Theta(F)$.

As we just saw, to prove a lower bound on the complexity of a problem Π we must give an argument that covers all possible algorithms to solve Π. This begs the question: What do we mean by "all possible algorithms"? A reasonable, though somewhat informal, answer is: All algorithms that can be written in some programming language. (The choice of the specific programming language is not very important because with sufficient ingenuity and varying degrees of difficulty we can translate any program from one language to another without significant loss of efficiency.)

This, in fact, is not the case. It is a fundamental and very interesting result of mathematical logic and theoretical computer science that there are non-computable problems. That is, there exist relations Π such that there is no algorithm which, given any x, can compute some y so that $(x, y) \in \Pi$. This result, in a form that is directly relevant to computer science was proved by Alan Turing in 1936, although it is implicit in Gödel's famous "Incompleteness Theorem", published in 1931.

However, such a general answer can sometimes become a hindrance to the derivation of useful lower bounds.

To see this, take the following example. Consider the problem of searching a given sorted sequence of keys to determine if it contains a given key. If we know nothing further about the nature of the keys (except, of course, that they can be compared so that it makes sense to say that they can be sorted) then it is reasonable to assume that the only operations we can apply to solve the problem are comparisons. Under this assumption we can (and actually will) prove that $\geq \lceil \log_2 n \rceil$ comparisons are needed (in the worst-case) by any algorithm that solves the problem, where n is the number of keys in the sorted sequence.

However, such a lower bound cannot be proved for *all* algorithms that solve the problem. For example, suppose we know that the keys are integers in the range 1 to m for some m. Then if the sorted sequence of keys is given to us in the form of an integer array $A[1..m]$ such that $A[i] = t$ iff there are t keys with the value i in the sorted list, we can answer the question "is key K in the list?" in constant time: All we have to do is check whether $A[K]$ is non-zero! This is is *not* a comparison-based algorithm because it assumes many properties of the keys beyond the fact that they can be compared; in particular it uses the keys as array indices. Nevertheless, it is a perfectly correct algorithm — albeit one which assumes that the input is represented in a particular manner. Thus, unless we are willing to restrict the set of algorithms that solve the problem to only comparison-based ones, we will not be able to prove a lower bound of $\lceil \log_2 n \rceil$ for searching a sorted sequence.

In these notes, we shall restrict our attention to comparison-based algorithms. One of the advantages of doing so is that there is a very convenient way of representing such algorithms as trees of a special form, called *decision trees*, or *comparison trees*. It turns out that this representation of algorithms is more useful for proving lower bounds than, say, Turing programs.

It is true that the restriction to comparison-based algorithms is not meaningful for all problems. For example if our problem is to multiply two matrices or to find the shortest path between two nodes in a graph, it makes no sense to restrict our attention to comparison-based algorithms. However, the restriction is meaningful for a surprising range of problems. Here are some examples:

Searching a sorted list:

Input: A sequence $A_1, A_2, \ldots, A_n$ of keys in sorted order (i.e., $A_1 \leq A_2 \leq \ldots \leq A_n$) and a key K.

Output: If $K = A_i$ for some $1 \leq i \leq n$ then output i; else output 0. (If there is more than one i, $1 \leq i \leq n$, such that $K = A_i$ then we are free to output any such i. This is another example which illustrates why, in general, we should think of a problem as a relation, instead of a function, between inputs and outputs.)

Searching an unsorted list: The problem is the same as above except that it is *not* necessarily the case that $A_1 \leq A_2 \leq \ldots \leq A_n$.

Selection:

Input: A sequence $A_1, A_2, \ldots, A_n$ of keys and a number k, $1 \leq k \leq n$.

Output: The index of the k-th smallest key in the sequence.

Note that finding the maximum, the minimum and the median of a sequence of keys are special cases of this problem with $k = n$, $k = 1$ and $k = \lceil n/2 \rceil$, respectively.

Finding the mode:

Input: A sequence $A_1, A_2, \ldots, A_n$ of keys.

Output: The index of the key that appears most frequently in the input sequence. (If there are several keys which appear with the same, maximum, frequency, the index of any one of them may be output.)

Sorting:

Input: A sequence $A_1, A_2, \ldots, A_n$ of keys.

Output: A permutation π of $1, 2, \ldots, n$ such that $A_{\pi(1)} \leq A_{\pi(2)} \leq \ldots \leq A_{\pi(n)}$

Merging two sorted sequences:

Input: Two sorted sequences $A_1, A_2, \ldots, A_m$ and $A_{m+1}, A_{m+2}, \ldots, A_{m+n}$ of keys.

Output: A permutation π of $1, 2, \ldots, m+n$ such that $A_{\pi(1)} \leq A_{\pi(2)} \leq \ldots \leq A_{\pi(m+n)}$

In general, we shall be interested in problems where the input consists of a sequence of keys whose values can be compared by a *comparison operator*. (Sometimes the input may contain other elements, in addition to keys. For example, an input to the selection problem contains a number k in addition to the keys that can be compared.)

2. Decision trees

Before giving the general definition of a decision tree, let's look at an example of one so that we can have some idea about what it is we are trying to describe. This particular tree, shown in Figure 1, is a decision tree for the problem of searching a sorted list of 8 keys, assuming that we know we are searching for a key that is actually in the list (i.e., we do not allow unsuccessful searches). The tree corresponds to the binary search algorithm.

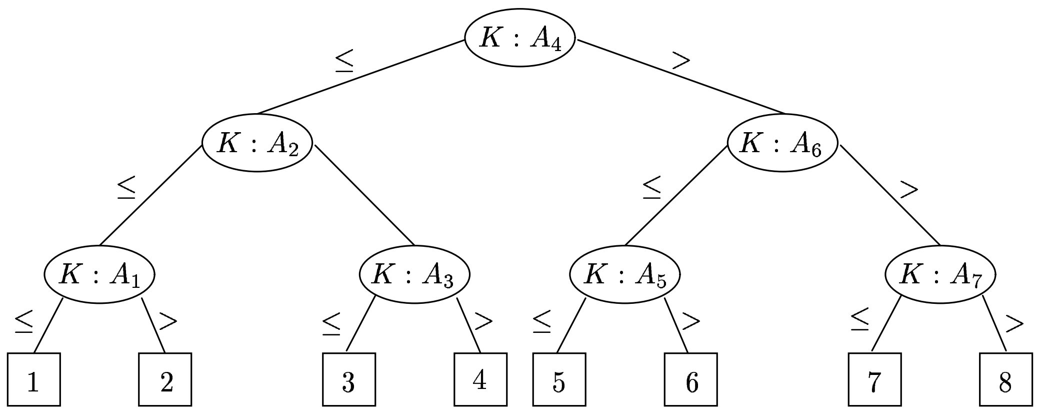

Figure 1

The internal nodes of this tree are labeled with comparisons between keys that appear in the input. For example, the root of the tree corresponds to a comparison between the key K we are searching for and the "middle" key in the sequence, A_4. Each internal node has two children, the edges to which are labeled $\leq$ and $>$. That is, the edges leading out of a node u are labeled with the possible outcomes of the comparison in u. The external nodes (leaves) of the tree are labeled with the possible outputs — in our case the indices of the 8 keys in the given sorted list. (Since we assume there are no unsuccessful searches, there are no leaves labeled '0'.)

The execution of the algorithm represented by this decision tree on a specified input corresponds to the traversal of a path from the root to a leaf. In each internal node the comparison specified in that node is made. The outcome of that comparison, given the actual values of the input keys, determines which edge out of that node will be taken. This process (perform a comparison and follow the appropriate edge) is repeated until a leaf is reached. At this point the algorithm terminates, giving as output the label contained in that leaf. For example, if the input consists of the sequence $2, 3, 4, 7, 11, 13, 17, 19$ and the key being sought is 17, the execution of the algorithm represented by the decision tree in Figure 1 corresponds to the path from the root of the tree to the leaf labeled 7.

The decision tree in Figure 1 is correct because no matter what the values of the input keys are, the output obtained in the way just described is the correct one — provided, of course, they conform to the assumptions that the input must satisfy; in particular, that the keys $A_1, A_2, \ldots, A_8$ are sorted and that K is one of these eight keys. On the other hand, this decision tree is *not* a correct one for the problem of searching unsorted lists — even if we assume that there are only successful searches. For example, if the input list of 8 keys is 17, 19, 2, 5, 7, 3, 11, 13 and the key we are searching for is 17, the above tree will output 8, while the correct output would have been 1.

Notice that the decision tree in Figure 1 is a representation of binary search *for a sequence of eight keys.* We have said that a decision tree corresponds to a (comparison-based) algorithm. Usually we think of an algorithm as being capable of handling inputs of any size. When we model algorithms as decision trees, the situation is slightly different: We need a different tree for each different input size. This is not really a problem because our main interest in decision trees is as a tool for proving *lower bounds.* If we can prove that any decision tree that computes the output for inputs of size n requires at least $L(n)$ comparisons in the worst case, then it follows that any (comparison-based) algorithm which can handle all input sizes requires at least $L(n)$ time in the worst case. This is simply because we can view such an algorithm as defining an infinite sequence of comparison-based algorithms (decision trees), each one handling inputs of a specific size.

In Figure 1 we have assumed that each comparison has one of two outcomes: $\leq$ or $>$. This is not the only possibility. We can assume other comparison operators too. For example, we might wish to have a comparison operator which has three outcomes: $<$, $=$ and $>$. Or, if we know that the keys that are being compared are distinct, there may be two outcomes, $<$ and $>$. Or, if the keys are objects which it is meaningful to compare for equality but not for order (e.g., they are pointers) then the two outcomes of a comparison might be $=$ and $\neq$. Finally, we might wish to allow comparisons not only between keys directly, but between functions of keys. Then the labels of the nodes would be of the form $f(A_{i_1}, A_{i_2}, \ldots, A_{i_k}) : g(A_{j_1}, A_{j_2}, \ldots, A_{j_l})$, where f and g are a k-ary and an l-ary function of keys, respectively. In these notes, however, we will only allow direct comparisons of keys.

Now we can give the general definition of decision trees.

Consider a problem Π. Fix a comparison operator which we can apply to pairs of input keys so that each comparison has up to t possible outcomes (typically $t = 2$ or 3).

A *decision tree of order n for problem* Π is a t-ary tree T such that

(i) each internal node u is labeled with a comparison $A : B$, where A, B are input keys; the edges from u to its children are labeled with distinct outcomes of the comparison between A and B;

(ii) each leaf is labeled with an output of Π on some input of size n.

Let x be an input of size n for Π. That is, x specifies values for the input keys. Then, *path*(x) is a path $u_1, u_2, \ldots, u_k$ in T such that

(1) u_1 is the root of T;

(2) for every $1 \leq m < k$, if u_m is labeled with $A : B$ then u_{m+1} is the child of of u_m such that the edge (u_m, u_{m+1}) is labeled with the outcome of the comparison between the values of keys A and B in x; and

(3) u_k is a leaf of T.

It is easy to prove (by induction) that for a given input x, if $path(x)$ exists, it is unique.

A decision tree T of order n for problem Π is *correct* iff for every input x of size n,

(iii) $path(x)$ is defined; and

(iv) the label in the leaf node of $path(x)$ is an output of Π on input x.

From now on, when we say "decision tree" we shall mean "correct decision tree", i.e., a tree that satisfies (i)–(iv).

Lemma 1. *The number of leaves in a decision tree of order n for problem Π is at least equal to the number of distinct outputs of Π on inputs of size n.*

PROOF: Suppose, by way of contradiction that there is some input x of size n such that there is no leaf labeled with an output of Π on x. Then, either $path(x)$ does not exist, violating (iii); or it exists but ends at a leaf labeled with something other than an output of Π on x, violating (iv). $\Box$

As we have seen, an execution of a decision tree algorithm corresponds to the traversal of a path from the root to a leaf of the tree. Under the reasonable assumption that each comparison takes a constant amount of time, the worst case time complexity of the algorithm is the length of the longest root-to-leaf path, i.e. the height of the tree.

We can also define the average complexity of the algorithm represented by a decision tree analogously, as the average length of a root-to-leaf path. More precisely, let the sample space Ω consist of all inputs of size n and define a random variable $L : \Omega \to \mathbf{N}$ by $L(x) = length(path(x))$. Then for a given probability distribution function P on Ω, the average complexity of the algorithm is $E(L) = \sum_{x \in \Omega} P(x) \cdot L(x)$.

Lemma 2. *Any t-ary tree with l leaves has height at least $\lceil \log_t l \rceil$ (for all $t \geq 2$).*

PROOF: Easy induction on l. $\Box$

Lemma 2, together with Lemma 1 and our characterisation of the complexity of a decision tree algorithm as the height of the tree, becomes a powerful tool for proving lower bounds on problems. The typical argument runs as follows:

> By Lemma 1, any decision tree that solves the problem for inputs of size n needs at least a certain number of leaves, say $l(n)$. Lemma 2 then implies that any such tree has height $\geq \lceil \log_t l(n) \rceil$. Thus, a lower bound on the problem (in the decision tree model) is $\Omega(\log_t l(n))$.

This is called an "information-theoretic argument" and lower bounds established in this fashion are called "information-theoretic lower bounds".

3. Applications of the information-theoretic argument

In all of the following we assume two-way comparisons between keys with outcomes $\leq$ and $>$ (or, in some cases $<$ and $>$).

A. Searching a sorted sequence

First, let us consider the variation of the problem where there are no unsuccessful searches. If we are searching a sequence of n keys, there are n possible outputs, so an immediate application of the information-theoretic argument yields a lower bound of $\lceil \log_2 n \rceil$ comparisons in the worst case.

In fact, binary search is an algorithm that solves the problem in exactly this number of comparisons. Thus, $\lceil \log_2 n \rceil$ is also an upper bound on the number of comparisons needed to search a sorted sequence. Since the upper and lower bounds match, we conclude that $\lceil \log_2 n \rceil$ is the worst-case complexity of searching a sorted sequence (assuming only successful searches).

Now consider the problem of searching a sorted sequence where we also allow unsuccessful searches. In this case there are $n+1$ possible outputs. Thus, a straightforward application of the information-theoretic argument yields a lower bound of $\lceil \log_2(n+1) \rceil$ comparisons. Unfortunately this cannot be matched by an upper bound. You can convince yourself of this by trying out all possible decision trees for solving this problem when $n=1$ or $n=2$. You will discover that for $n=1$ we need at least two comparisons (while the lower bound derived above yields only one comparison in this case); and for $n=2$ we need at least three comparisons (while the lower bound above yields only two comparisons).

The trouble is that for output 0 (i.e. unsuccessful search) we have counted only one leaf node in the tree while the experience you must have gained by trying out trees for $n=2$ suggests that more than one leaf labeled 0 is needed: We need one such node for each possible way that a search can be unsuccessful. More precisely, we must have one node labeled 0 for each of the $n+1$ between-key intervals defined by the given n keys. The following Lemma proves that this is, in fact, so.

Lemma 3. *If T is a decision tree for the problem of searching a sorted sequence of n keys (with possible unsuccessful searches) then T must contain $\geq n+1$ leaves labeled 0.*

Proof: Let E_0 denote the interval of keys strictly less than A_1, E_i denote the interval strictly between keys A_i and A_{i+1} for all $1 \leq i \leq n-1$, and E_n denote the interval of keys greater than A_n). Thus, an unsuccessful search is a search for some key in E_i, for some $0 \leq i \leq n$. Let $leaf(E_i)$ be the leaf in the path of T that corresponds to a search for some key in E_i.†

Now we claim that if $i \neq j$ then $leaf(E_i) \neq leaf(E_j)$. In other words, unsuccessful searches for keys in different intervals must necessarily end in different leaves. This will prove the lemma, since all of these leaves will be labeled 0 and there are $n+1$ between-key intervals.

To prove the claim we shall assume that it is false and derive a contradiction. So, suppose that there are $i \neq j$ such that $leaf(E_i) = leaf(E_j)$. In other words, an unsuccessful search for a key in E_i and an unsuccessful search for a key in E_j will both end in the same leaf, call it l. Without loss of generality, assume $i < j$. Under these circumstances we'll show that the search for a key $K = A_m$ (for any $m = i+1, \ldots, j$) will also end up in l, which will be the desired contradiction, since a successful search (for A_m) and unsuccessful searches (for keys in E_i or E_j) will both end up in the same leaf l.

To see that the search for $K = A_m$ must end up in l consider the comparisons in the path from the root to l. First notice that there is no comparison of the form $K : A_t$ for any $t = i+1, \ldots, j$. (If there was, then the outcome of such a comparison when K is a key in interval E_i would be $<$, while the outcome of that comparison with a key in E_j would be $>$, meaning that a search for keys in these two intervals could not both end up in l). Thus all comparisons on that path fall in one of three categories: (a) $K : A_t$, for $t \leq i$; (b) $K : A_t$, for $t \geq j+1$; or (c) $A_t : A_r$ for some t, r. But any such comparison will have the same outcome when $K = A_m$ as when K is in the interval E_i

† This is well-defined in the sense that no matter what keys we are given as input, if the last input key K (the one we are searching for) is in the ith interval between keys then a search for that key will always end up in the same leaf. This can be proved by induction on the path followed in such a search, and depends on the assumption that the comparisons specified in the nodes of our decision trees are only between input keys, and not arbitrary functions of them.

or E_j. (For (a) the outcome will be $>$ in all cases; for (b) it will be $<$ in all cases; and for (c) the outcome is independent of K.) Therefore, the search for $K = A_m$ follows exactly the same path as the search for a key in E_i or E_j and therefore ends up in leaf l, which is labeled 0. This contradicts the assumption that T is correct, since the search for $K = A_m$ is successful and should not end in a leaf labeled 0. This contradiction implies that there can't be distinct intervals E_i, E_j such that $leaf(E_i) = leaf(E_j)$ and therefore we must have at least $n + 1$ leaves labeled 0, as wanted. $\square$

Given Lemma 3 we can now conclude that any decision tree that solves the problem of searching a sorted sequence (with possible unsuccessful searches) must contain $2n+1$ leaves. The information-theoretic argument now yields a lower bound of $\lceil \log_2(2n + 1) \rceil$ comparisons, which is actually matched by binary search, so that this is the exact complexity of the problem.

This example illustrates that certain refinements may be necessary before the information-theoretic argument can give the best possible lower bound. A more dramatic illustration of the same point is afforded by considering this variation of the problem: Given a sorted sequence of keys and a key K output "yes" if K is in the sequence and "no" if it is not. (That is, we don't care to know where the key is to be found — just that it is there.) A naive application of the information-theoretic argument is to say that a decision tree that solves this problem must have at least two leaves (since there are two possible outputs) and therefore there must be at least $\lceil \log_2 2 \rceil = 1$ comparison in any decision tree that solves the problem. This is true, but trivial! It *is* a lower bound but nowhere near the best possible one. A more careful analysis is needed to show that if there are n distinct keys, a correct decision tree must have at least n leaves labeled "yes" (one for each possible key) and, by an argument analogous to Lemma 3, at least $n+1$ leaves labeled "no". This gives us a lower bound of $\lceil \log_2(2n + 1) \rceil$ comparisons for this version of the problem too. This is best possible, because binary search achieves this bound.†

B. Searching an unsorted sequence

The lower bound arguments on searching a sorted sequence did not make use of the fact that the sequence is actually sorted, so they apply in this case too. However, binary search, the algorithm that matched those lower bounds, does not work with unsorted sequences. In fact, there is no (comparison-based) algorithm that can search an unsorted list in logarithmic time. As we shall prove in the next section, n comparisons are necessary (and, of course, sufficient) to search an unsorted sequence of n keys. This fact will be proved by using a different type of argument, called an "adversary argument". Thus, the information-theoretic argument, useful and powerful as it is, is not a panacea that takes care of all lower bounds!

† Here it is important to recall our assumption that *only* comparisons with outcomes $\leq$ and $>$ are allowed. If we also permitted direct comparisons for equality, these results do not quite hold, although something close enough does: In that case, it can be shown that at least $\lceil \log_2 2n \rceil = \lceil \log_2 n \rceil + 1$ comparisons are necessary (and, using binary search, sufficient). The proof of this is similar in spirit to the proof given below, but the details are somewhat more complex. It is based on the following fact: *There is a minimum height decision tree with at least n leaves labeled 0.* (Compare this to Lemma 3.) It is *not* the case that *every* decision tree must have at least n leaves labeled 0. For example, you can easily verify that the decision tree that corresponds to sequential search (and which uses only equality comparisons), has only one leaf labeled 0 — however, this is not a decision tree of minimum height. In fact, it is not even necessary that every *minimum* height (i.e., optimal) decision tree must have n leaves labeled 0. However, *some* optimal decision tree does! It is such intricacies that make this case technically more complicated.

C. Sorting a sequence

To simplify the discussion we shall assume that the n keys to be sorted are distinct. We can make this assumption without loss of generality since, any algorithm that sorts all sequences of n keys (distinct or not) will surely sort sequences of n distinct keys. Thus, if a lower bound holds when we assume that the keys are distinct, it will hold, *a fortiori*, if we don't make this assumption.

Recall that the output in our formulation of sorting is a permutation π of $1, 2, \ldots, n$ which tells us how to rearrange the n input keys to get them in sorted order. Since we have n distinct input keys there are exactly $n!$ permutations of the keys and therefore $n!$ outputs for sorting them. The information-theoretic argument says that we need at least $\lceil \log_2 n! \rceil$ comparisons. But $\log_2 n! = \sum_{i=1}^{n} \log_2 i$, and as we have seen the latter quantity is in $\Theta(n \log n)$. We conclude that $\Omega(n \log n)$ is an asymptotic lower bound on the worst-case complexity of comparison-based sorting. This is asymptotically tight since we know several (comparison-based) algorithms that can sort n keys in $O(n \log n)$ time (e.g., heap sort and merge sort). Thus, the complexity of sorting (in the decision tree model) is $\Theta(n \log n)$.

We have just proved that it is impossible to sort n keys in time asymptotically faster than $O(n \log n)$ in the worst-case (using comparison-based algorithms). Another interesting question is, can we sort faster than that on the average? We shall now prove that we cannot, if all permutations of the keys to be sorted are equally likely inputs.

Suppose we have n keys to sort and let T be a decision tree for sorting n keys. Let $d_T(\pi)$ be the depth in T of the leaf labeled with permutation π, and $P(\pi)$ be the probability that an execution of the algorithm will traverse the path leading to that leaf. Since all permutations of the keys are equally likely inputs, we have that for any π, $P(\pi) = 1/n!$. The expected number of comparisons to sort n keys using decision tree T is,

$$C_{avg}(T) = \sum_{\pi \text{ a leaf in } T} P(\pi) \cdot d_T(\pi) = EPL(T)/n!$$

where $EPL(T)$ is the external path length of T, i.e. the sum of the depths of all the leaves. Thus, to minimise the expected number of comparisons to sort n keys we must find a decision tree for sorting n keys which has minimal external path length.

Lemma 4. *A binary tree that has minimum external path length and ℓ leaves has 2^d nodes of depth d, for each $0 \leq d < \lceil \log_2 \ell \rceil - 1$.*

Proof sketch: If not, then the tree will have a leaf u at depth $\geq \lceil \log_2 \ell \rceil$ and some node v with at most one child at depth $< \lceil \log_2 \ell \rceil - 1$. Moving u and making it a child of v will reduce the external path length of the tree. $\square$

Hence, a tree with ℓ leaves that minimises the external path length has all its leaves at depth $\lceil \log_2 \ell \rceil - 1$ and $\lceil \log_2 \ell \rceil$. Therefore, a tree with ℓ leaves has external path length greater than $\ell\,(\lceil \log_2 \ell \rceil - 1)$. Thus, for any decision tree T that sorts n keys,

$$C_{avg}(T) > n!\,(\log_2 n! - 1)/n! = \log_2 n! - 1 \in \Omega(n \log n).$$

This means that, assuming that all key permutations are equally likely inputs, we cannot sort n keys faster than $O(n \log n)$ time, even on the average. This bound is tight since any algorithm that sorts n keys using $O(n \log n)$ comparisons in the worst case, will surely sort with $O(n \log n)$ comparisons on the average.

D. Merging two sorted sequences

In this problem the input consists of two sorted sequences $A_1 < A_2 < \ldots < A_m$ and $A_{m+1} < A_{m+2} < \ldots < A_{m+n}$ of keys. The output is a permutation π of $1, 2, \ldots, m+n$ such that $A_1 < A_2 < \ldots < A_{m+n}$. (Just as in sorting, we shall assume, without loss of generality, that all input keys are distinct.)

Even though we have a total of $m+n$ input keys, not all $(m+n)!$ permutations of the keys are possible outputs. This is because each of the two input sequences is sorted, so an output permutation cannot rearrange the keys within each of the two sequences. The number of possible outputs is, in fact, $\binom{m+n}{m}$. To see this think of the output as consisting of $m+n$ locations into which we must place the $m+n$ keys of the two sequences so that they appear in sorted order. If we choose m locations to put the keys of the first sequence $(A_1, \ldots, A_m)$ we have completely determined a possible output (because the keys of the first sequence will go into these m location in the order in which they appear in the input and the keys of the other sequence will go into the remaining locations in the order in which the appear in the input). Thus, the number of possible outputs is the number of ways in which we can select m locations out of the $m+n$; this is, by definition, $\binom{m+n}{m}$.[†]

The information-theoretic argument then, yields a lower bound of $\lceil \log_2 \binom{m+n}{m} \rceil$ comparisons to merge two sorted sequences of length m and n.

Now consider the special case where we are merging two sorted sequences of the same length, n. Then the above lower bound becomes $\log_2 \binom{2n}{n}$. Using Stirling's approximation it is easy to show that $\binom{2n}{n} \approx 2^{2n}/\sqrt{\pi n}$. Thus, in this case we get a lower bound of

$$\begin{aligned}\log_2 \binom{2n}{n} &\approx \log_2 2^{2n} - \frac{1}{2}\log_2 n - \frac{1}{2}\log_2 \pi \\ &\approx 2n - \frac{1}{2}\log_2 n - .826.\end{aligned}$$

comparisons to merge two sorted sequences of size n. The standard algorithm makes at most $2n-1$ comparisons in the worst case to merge two sorted sequences of size n. We see that there is a gap of about $\frac{1}{2}\log_2 n$ between the best known upper bound and the lower bound obtained using the information-theoretic argument. This is not as large as the gap of a *factor* of $n/(\log_2 n)$ that we saw between the upper bound and the information-theoretic lower bound obtained for searching an unsorted sequence, but it is still annoying. As we shall see in the next section, an adversary argument can help establish a better lower bound that closes this gap too!

4. The adversary argument and some applications

Consider a problem Π of the type we have been discussing, i.e. whose inputs are sequences of keys and the keys can be compared. Imagine two players, Alice and Bob, engaged in the following game: Alice submits to Bob questions of the form "what is the outcome of comparing input key A to input key B?" (abbreviated "$A : B$"). Bob responds to such a question by specifying the outcome, subject to the constraint that all his answers are consistent. That is, there must exist values for the input keys, so that his answers are true for that particular input. The goal of Alice is to ask as few questions as possible in order to determine the output. The goal of Bob is to force Alice to ask as many questions as possible before she can figure out what the output is.

[†] We could of course choose n locations to put the keys of the second sequence. This would yield $\binom{m+n}{n}$ possible outputs. This is OK, since $\binom{m+n}{m} = \binom{m+n}{n}$.

It is helpful to think of Alice as a decision tree algorithm and Bob as an input x for the algorithm. Alice's questions correspond to the comparisons which the algorithm stipulates must be performed and Bob's answers correspond to the outcomes of those comparisons when the input is x. Since Bob's goal is to force Alice to ask as many questions as possible, we can view Bob as an adversary who is trying to cause an algorithm to perform many comparisons.

We say that Bob's answers are consistent with an input x if his answer to every question "$A : B$" asked by Alice is correct for the values of keys A and B in x. Notice that there may be several inputs consistent with Bob's answers; or, there may be no input consistent with Bob's answers, in which case we say that his answers are inconsistent. A *strategy* for Bob is a prescription for answering Alice's questions in a consistent manner.

The lower bound technique we are about to present is based on the following lemma.

Adversary Argument Lemma. *Suppose that for any algorithm (decision tree) used by Alice to ask questions, Bob has a strategy for answering Alice's questions so that for any sequence of fewer than q questions that she asks, the answers given by Bob (according to his strategy) are consistent with two inputs x and x' such that the outputs of Π on x and x' are different. Then q is a lower bound on the worst case number of comparisons needed to solve Π.*

Informally, this is true because, on the basis of Bob's answers to fewer than q questions, Alice can't distinguish between the possibility of the input being x and it being x'. Since these two inputs yield distinct outputs, she can't possibly determine the output after fewer than q questions.

The proof of the Adversary Argument Lemma formalises this idea.

PROOF: Suppose, by way of contradiction that there is a decision tree T for Π which has height $< q$. Consider the root-to-leaf path of T obtained as follows:

a. Visit the root.
b. Let Alice ask Bob the question in the node of T that is presently being visited and let Bob answer the question using the strategy mentioned in the Lemma. Follow the edge from the node being visited that corresponds to Bob's answer and visist the node to which that edge leads.
c. Repeat (b) until the node being visited is a leaf. Let that leaf be l.

Since T has height $< q$ this process will terminate before Alice has asked q questions. By assumption, the answers given by Bob using his strategy, are consistent with two inputs x and x'. This means that $path(x)$ and $path(x')$ are both the path from the root of T to l. Since T is a correct decision tree for Π, the label of l must be an output of Π on both x and x' which is impossible since these inputs yield distinct outputs. □

To apply the adversary argument for a particular problem Π, we must prove that Bob has a strategy with the properties mentioned in the Lemma. In general, Bob can fix his strategy in full knowledge of Alice's algorithm for asking questions. Recalling our analogy that Alice corresponds to a decision tree algorithm and Bob to an input, the adversary argument lemma can be paraphrased as follows: For any algorithm that solves Π there is a "bad" input x such that the algorithm cannot determine the output of Π on x in fewer than q comparisons. In all our applications of the adversary argument it so happens that Bob can fix his strategy independent of Alice's algorithm. That is, what we actually have in these cases in that there is a "bad" input x such that no algorithm that solves Π can determine the output on input x in fewer than q comparisons. The *same* "bad" input works for all algorithms). However, there are other, more complicated, situations where to apply the adversary argument effectively, we must allow Bob to vary his strategy depending on Alice's algorithm.

We shall now give a few applications of the adversary argument.

A. Searching an unsorted sequence

Recall that the input consists of a sequence $A_1, A_2, \ldots, A_n$ of keys and a key K; the output is i if $K = A_i$ for some $1 \leq i \leq n$, and 0 otherwise. Bob's strategy in answering questions is this: If asked "$A_i : A_j$" he answers "$<$" if $i < j$ and he answer "$>$" if $i > j$; if asked "$A_i : K$" he answers "$<$"; and if asked "$K : A_i$" he answers "$>$". That is, Bob's strategy is to answer Alice's questions as if the input keys satisfy $A_1 < A_2 < \ldots < A_n < K$.

Consider *any* set if fewer than n questions. Given this set construct an undirected graph whose nodes are the $n+1$ input keys (the n keys in the sequence plus the key K being sought). Put an edge between two keys iff there is a question comparing them. Since there are fewer than n questions there are fewer than n edges. Since the graph has $n+1$ nodes and fewer than n edges it must be disconnected.† Let X be the set of keys (= nodes) in the graph that are in the same connected component as K (i.e., there is a path from these nodes to K); and let $\overline{X}$ be the set of the remaining keys.

Let $x = \alpha_1, \alpha_2, \ldots, \alpha_n, \kappa$ be an input for the $n+1$ keys such that $\alpha_1 < \alpha_2 < \ldots < \alpha_n < \kappa$. Obviously, x is consistent with Bob's answers to the questions and the output in this case is 0.

Now let $x' = \alpha'_1, \alpha'_2, \ldots, \alpha'_n, \kappa'$ be an input obtained as follows: The keys among $A_1, \ldots, A_n$ that are in X have increasing values according to their index and K has a value greater than all of them. More precisely, if $A_i, A_j \in X$ and $i < j$ then $\alpha'_i < \alpha'_j$, and $\alpha'_i < \kappa'$. Similarly, the keys in $\overline{X}$ have increasing values according to their index but the key in $\overline{X}$ with maximum index has value equal to κ'. That is, if $A_i, A_j \in X$ and $i < j$ then $\alpha'_i < \alpha'_j$; and $\alpha'_m = \kappa'$, where $m = \max\{i : A_i \in \overline{X}\}$. The output for x' is *not* 0 because in x', $K = A_m$.

Since there are no edges between nodes in X and $\overline{X}$ in the graph, there are no questions comparing keys in X to keys in $\overline{X}$. Since the relative order of the keys in X and $\overline{X}$ is the same in x as in x' and Bob's answers are consistent with x, they are also consistent with x'. But as we have seen the outputs for x and x' are different.

By the adversary argument then, any decision tree that solves the problem of searching an unsorted sequence of n keys requires at least n comparisons in the worst case. This lower bound is best possible because sequential search is an algorithm that solves the problem using exactly n comparisons in the worst case.

B. Finding the maximum

The input is a sequence of n keys $A_1, A_2, \ldots, A_n$, not necessarily sorted. To simplify matters we assume that the keys are distinct. This assumption can be made without loss of generality. The output is the index of the maximum key.

Let Bob's strategy to answer questions be as follows: If Alice asks "$A_i : A_j$", he answers "$<$" if $i < j$ and "$>$" if $i > j$. That is, Bob answers questions as if the input satisfies $A_1 < A_2 < \ldots < A_n$. If the outcome of a comparison $A_i : A_j$ is $<$ we call A_i the *loser* and A_j the *winner* of the comparison.

Suppose that Alice asked fewer than $n-1$ questions. Since each comparison yields exactly one loser, there are fewer than $n-1$ losers in all of her questions. Thus there are at least two distinct keys which never lost a comparison. One of them must, of course, by A_n. Let A_m be another.

† It is an elementary result of graph theory that a graph with k nodes and fewer than $k-1$ edges is disconnected.

Let $x = \alpha_1, \alpha_2, \ldots, \alpha_n$ be an input such that $\alpha_1 < \alpha_2 < \ldots < \alpha_n$. Let $x' = \alpha'_1, \alpha'_2, \ldots, \alpha'_n$ be the input where $\alpha'_i = \alpha_i$ for all $i \neq m$, and α'_m is a value greater than α'_n. We claim that both these inputs are consistent with Bob's answers. This is obvious for x. Regarding x' note that all comparisons *not* involving A_m will have the same outcome as in x, since the keys compared are identical. Comparisons involving A_m will also have the same outcome since A_m, by assumption, did not lose any comparison in x and it cannot lose any comparison in x' because it is the maximum key in that input! Therefore, Bob's answers are consistent with both x and x'. But the outputs for these two keys are different: n in the case of x, and m in the case of x'.

By the adversary argument then, any decision tree algorithm that solves the problem of finding the maximum of n keys requires at least $n - 1$ comparisons in the worst case. This lower bound is tight since the straightforward algorithm for finding the maximum of n keys requires $n - 1$ comparisons.

C. Merging two sorted sequences of equal length

Let $A_1, A_2, \ldots, A_n$ and $B_1, B_2, \ldots, B_n$ be the two input sorted sequences (for ease of exposition we use $B_1, B_2, \ldots, B_n$ instead of $A_{n+1}, A_{n+2}, \ldots, A_{2n}$ here). The output is a rearrangement of the $2n$ input keys so that the entire sequence is in sorted order.

Let Bob answer questions as follows: If Alice asks "$A_i : B_j$, answer "$<$" if $i \leq j$; answer "$>$" if $i > j$. That is, Bob answers as if the input satisfies $A_1 < B_1 < A_2 < B_2 < \ldots < A_n < B_n$. (We can assume, without loss of generality, that Alice does not waste her breath asking questions like "$A_i : A_j$" or "$B_i : B_j$" since the two input lists are already sorted.)

Suppose Alice asks fewer than $2n - 1$ questions. Then she must *not* have asked the question "$A_i : B_i$" for some $1 \leq i \leq n$, or the question "$B_i : A_{i+1}$" for some $1 \leq i < n$. (This is because there are $2n - 1$ such questions and she has asked fewer than that number.) Let a question she has not asked be "$A_m : B_m$" for some m. Let $x = \alpha_1, \alpha_2, \ldots, \alpha_n, \beta_1, \beta_2, \ldots, \beta_n$ be an input satisfying $\alpha_1 < \beta_1 < \alpha_2 < \beta_2 < \ldots < \alpha_n < \beta_n$. Let $x' = \alpha'_1, \alpha'_2, \ldots, \alpha'_n, \beta'_1, \beta'_2, \ldots, \beta'_n$ be the input obtained from x by swapping the values of A_m and B_m — i.e., $\alpha'_i = \alpha_i$ and $\beta'_i = \beta_i$ for all $i \neq m$, and $\alpha'_m = \beta_m$ and $\beta'_m = \alpha_m$. Thus, x' satisfies $\alpha'_1 < \beta'_1 < \alpha'_2 < \beta'_2 < \ldots < \beta'_{m-1} < \beta'_m < \alpha'_m < \alpha'_{m+1} < \cdots < \alpha'_n < \beta'_n$.

Since the outcome of any comparison between keys, other than $A_m : B_m$, is the same in x and in x' and since that comparison is not one in Alice's questions, Bob's answers are consistent with both x and x'. Furthermore, the two inputs result in distinct outputs (because the rearrangements to get a sorted sequence differ in the order which they assign to A_m and B_m).

In case the question not asked by Alice was of the form "$B_m : A_{m+1}$" we get a similar situation with x' obtained from x by swapping the values of B_m and A_{m+1} (instead of A_m and B_m).

Thus, if Alice asks fewer than $2n - 1$ questions we have two inputs that yield distinct outputs, both of which are consistent with Bob's answers. By the adversary argument then, any decision tree algorithm that merges two sorted sequences of length n requires at least $2n - 1$ comparisons in the worst case. This lower bound is best possible since the standard algorithm for merging two sorted sequences uses no more than $2n - 1$ comparisons in the worst case.

Exercise: Using an adversary argument, prove a lower bound of $\lceil \log_2 n \rceil$ comparisons for searching a sorted sequence of n keys.

This exercise is an example of a lower bound proof based on the adversary argument in which Bob's strategy for answering questions depends on Alice's algorithm for asking questions. That is, there is no single input x that is "bad" for *all* algorithms, but for every algorithm there is *some* "bad" input. In this case, an input is "bad" if it causes the algorithm to perform at least $\lceil \log_2 n \rceil$ comparisons.