CS 624
Lecture 1: The Analysis of Algorithms

1 General stuff

1. Read the course home page carefully if you haven't already. You are responsible for everything on that page.
2. *Immediately* apply for a course account. Even though you will not use that account to pass in work, it will enable me to send email to the class. If you don't have an account, you won't get my emails. (And do check your email here!)
3. Assignments will be posted on the course home page.
4. As I explain on the course home page, I encourage you to talk with other students in the class and in fact with anyone else about this course and (in general) about problems in the course. You should state whom you discussed things with in your written work. This will not lower your grade.
5. However: all the work you hand in *must* be written entirely by you. **You may not copy all or even part of a solution from anyone else, or from the web, or from a book, or from anything else—even if you acknowledge the source.** I have absolutely no tolerance for this.

   And you may not get help from anyone in writing things up.

   With one exception: me. You may ask me for help (either in person or by email) in expressing things, and I'll be happy to give what help I think is appropriate. This will not lower your grade. And—please be careful about this—I expect you to be able to explain to me anything you have written.
6. There will be two exams in class, and there will be a final exam. All exams will be closed-book. You should bring nothing to the exams except something to write with.
7. Don't be afraid to ask questions. I like questions. Don't be afraid that you might appear "stupid". None of you are stupid.
8. Don't be afraid to send me email. I expect to get email. It doesn't offend me. It won't lower your grade.
9. You may typeset your homework if you want, but you really don't need to. You can just write it out neatly. And you must explain things clearly. I do need to be able to read it easily! This is very important. I'm not a mind reader. I don't automatically know what you were thinking when you wrote something. **If I can't understand it, then it's wrong.**
10. Finally, please staple your homework together. No paper clips or anything else. Just one staple.

2 A Word From Our Founder

Algorithms—including some very sophisticated algorithms—go back at least to the ancient Greeks several thousand years ago. The term "algorithm" was borrowed from Arabic sometime in the Middle Ages.

The term "analysis of algorithms" however, which is the title of this course, and which names one of the main branches of computer science, is actually not very old at all. In fact, it originated with Donald Knuth in the introduction to his famous series of books in 1967:

> The subject of these books might be called "nonnumerical analysis." Although computers have traditionally been associated with the solution of numerical problems such as the calculation of the roots of an equation, numerical interpolation and integration, etc., topics like this are not treated here except in passing. Numerical computer programming is a very interesting and rapidly expanding field, and many books have been written about it. In recent years, however, a good deal of interesting work has been done using computers for essentially nonnumerical problems, such as sorting, translating languages, theorem proving, the development of "software" (programs to facilitate the writing of other programs), and simulation of various processes from everyday life. ...
>
> Of course, "nonnumerical analysis" is a terribly negative name for this field of study, and it would be much better to have a positive, descriptive term which characterizes the subject. "Information processing" is too broad a designation for the material I am considering, and "programming techniques" is too narrow. Therefore I wish to propose *analysis of algorithms* as an appropriate name for the subject matter covered in these books; as explained more fully in the books themselves, this name is meant to imply "the theory of the properties of particular computer algorithms."
>
> Donald E. Knuth, from the preface to *The Art of Computer Programming, Vol. 1*

3 The Analysis of Algorithms

This course could really be thought of as a mathematics course. In particular, there is no programming in this course at all, although if you have not had some serious programming experience, you probably won't get the full benefit out of what we're doing here.

Nor is this course a "cookbook" course. It's not a course in "really neat algorithms you can use". Of course we will be discussing some interesting and important algorithms. But the course focuses entirely on two topics:

- Given an algorithm for computing something, we need to be able to prove that the algorithm is correct, in the sense that
  1. It actually halts on every input, and
  2. When it does halt, it has computed what it is supposed to compute; i.e., "the correct answer".
- Given an algorithm for computing something, we need to know how "good" or "efficient" that algorithm is. As I just stated it, this is a vague question, and there are several different ways

to ask it. For instance, we might ask how much space (memory or disk space) an algorithm needs. Or we might ask how much time it takes. And there are subtle but important variations of these questions that will be very important to us.

For example, let us consider what computer scientists call a "dictionary":

A dictionary is a set of $\langle key, value \rangle$ pairs. A *value* is retrieved by looking up its associated *key*. Dictionaries come up all over the place in computer science.

1. Some dictionaries are built once and items are never deleted from them. An example would be a symbol table in a compiler. In such a case, a hash table is the best thing to use.
2. Some dictionaries need to handle both insertions and deletions. In such a case, some kind of binary search tree might be a better choice.

Note that I haven't proved either of these statements. And I'm not going to do it right now, either. We *will* be looking into some of this material later in the course, in some detail.

One important thing to notice, however—and it's a common phenomenon—is that many problems can be solved by more than one kind of algorithm. Often there is no "best" one—the one you want to use may depend on the situation.

We need to have a way of reasoning about these things so that the answers we come up with are not just things like "well, it seemed to work out best for me this way rather than that."

The "ways of reasoning" are all based on the mathematical analysis of algorithms. Mathematics provides us with a powerful tool that in many cases can tell us very accurately what an algorithm's strengths and weaknesses are.

Mathematical reasoning is also used in our first goal—proving that an algorithm actually does what we want it to do—i.e., that it is "correct". And such such proofs are important in themselves, because they usually give us some important insight into why the algorithm works. This is important for a computer scientist. (Suppose for example that we want to modify the algorithm slightly. Will it still work? Will it still be efficient?)

So we will be writing a lot of proofs in this class—both proofs of correctness and proofs of measures of efficiency. Some of the mathematics we will be using is pretty sophisticated. This is not a "plug-and-chug" course. You will have to think hard about a lot of this.

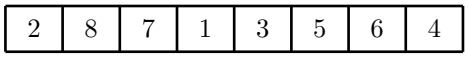
Don't be afraid of mathematics!

So now to get going, we're going to talk about a central problem in computer science—the problem of sorting—and two algorithms that can be used to solve it, and we'll say a little about how those two algorithms can be analyzed. Think of this as a general introduction to how this course will proceed. As we go on, our methods will use more far-reaching techniques.

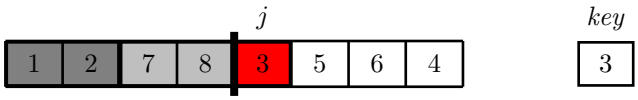
4 Insertion sort

Here is the code for insertion sort as given in the text. To make things simple to state, let us assume that we are sorting an array $A[1\,..\,n]$ of numbers. The same reasoning would apply if A were an array of any sort of elements that could be compared. For instance, we could have an array of words, and compare the words using alphabetical order. Also, to make things simple, let us assume that all the elements of A are distinct.

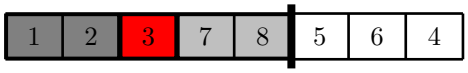
```
INSERTION-SORT(A)    // A is a 1-based array.
  for j ← 2 to A.length do
    key ← A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i ← j − 1
    while i > 0 and A[i] > key do
      A[i + 1] ← A[i]
      i ← i − 1
    A[i + 1] ← key
```



The initial unsorted array.



Beginning of step j. In this case, $j = 5$. The variable *key* holds the value 3. All the gray elements (to the left of the thick vertical line) are in order. The lightly shaded elements are the ones greater than the value 3 of the *key*.



End of step j. Now all the elements to the left of the thick line (which has been moved 1 element to the right) are in order. The next step will start with $j = 6$ and *key* will hold the value 5.

Figure 1: How insertion sort works.

There are two important questions we need to address:

Correctness. How do we know that this algorithm is correct? That is, can we prove it always gives the correct answer?

Efficiency. How efficient is it? What is the running time?

So now we address those two problems.

4.1 Proof of correctness of insertion sort.

This is essentially an inductive proof, where the induction is over the iterations of the **for** loop.

When you prove something by induction, there is always what we call the *inductive hypothesis*. It's a statement that should be pretty obviously true at the beginning of the process and that you have to prove continues to be true as the process goes on.

When you give a proof by induction, you have to state explicitly what the inductive hypothesis is. If you don't, you haven't given a proof by induction[1].

In this case, the inductive hypothesis is what the text calls a *loop invariant*:

4.1 Lemma (The inductive hypothesis) *At the start of each iteration of the loop (each iteration being characterized by a value of j), the numbers in $A[1\,..\,(j-1)]$ are in sorted order. (In fact, they are the numbers that were originally in $A[1\,..\,(j-1)]$, but they are now in sorted order.)*

PROOF. When $j = 2$, this is trivially true. (Why?)

To pass from $j-1$ to j, we insert the j^{th} element (i.e., the *key*) just below the least element in $A[1\,..\,(j-1)]$ that is greater than it. So it is less than all the elements on its right, and greater than all the elements on its left. (Why is this?) And this just means that the elements in $A[1\,..\,j]$ are now in sorted order. □

Thus, when we are done (i.e., after the last iteration, which we can regard as the start of the nonexistent "next one"), all the elements in A are sorted.

And that's the proof.

4.1.1 What do we really mean by "the inductive hypothesis"?

People often get very confused about what the inductive hypothesis is. In fact, it might be better—and I encourage you to do this—to think of the inductive hypothesis as a *sequence* of statements. In this case, the statements are as follows:

- Statement 5 is: "At the start of iteration 5 of the loop, the numbers in $A[1\,..\,4]$ are in sorted order."
- Statement 6 is: "At the start of iteration 6 of the loop, the numbers in $A[1\,..\,5]$ are in sorted order."

and so on. You should be able to see that Statement 1 is vacuously true[2] And Statement 2 is trivially true. So we do have somewhere to start (as we already noted above). The proof then showed that if for some number j, Statement j was known to be true, then it follows (by the algorithm; i.e., the pseudo-code) that the next statement (i.e., Statement $j+1$) must be true. Since we know that Statement 2 is true, it thus follows that Statement 3 must be true. And the same reasoning shows us that we can continue in this manner and conclude that Statement j is true *for all* values of $j \geq 2$, which is what we needed to show.

[1] And quite likely, you haven't given a proof at all.

[2] Do you know what that means? If you don't, you need to look it up immediately. It's important for everything we do in this course.

4.1.2 A warning

For some reason, inductive proofs seem to be difficult at first. And I have seen many students who believe that you can get from "statement j" to "statement $j+1$" by simply substituting $j+1$ for j. **That can't be true.** That would only work if we knew from the beginning that the inductive hypothesis was true for all j. And we don't. (Because if we really knew that to begin with, we would be done, right?)

So we have to use something special to get from the presumed fact that statement j is true to deduce that statement $j+1$ is then true. And that "something special" **has** to be something based on how the algorithm actually works. In the case here, that "something special" was the red paragraph above.

You should take whatever time you need in order to understand what I have written here. I'm not going to go over this any more in class. You should all have had this in at least one previous class anyway, and I expect you to understand it and be able to use it.

4.2 Efficiency of insertion sort

What's the running time? Clearly, it depends on the input. For instance,

- A longer input will generally take longer.
- An already sorted input will take less time.

Let's express the running time in terms of the length n of the input. This takes care of the first item above. As for the second, there are three possibilities to consider:

the worst-case time For an input of size n, imagine finding the input $A[1\,..\,n]$ which takes the longest time to sort. That gives the worst-case time (as a function of n). The worst-case time is useful because it gives a guarantee: you know that no matter what the input is, you will certainly do at least that well.

the best-case time For an input of size n, imagine finding the input $A[1\,..\,n]$ which takes the shortest time to sort. That gives the best-case time. Leiserson says this is "bogus", because you almost never get the best-case time in practice. It's sort of "cheating", he thinks. But it does tell something—it tells you that using this algorithm, you will never do better than the best-case time.

the average-case time Average the times that the algorithm takes over all possible inputs of length n. To do this, we need some assumption of the statistical distribution of the inputs. (For instance, if we know that for our particular application certain inputs will never occur, we can ignore them in figuring out the average.)

Average-case analysis is the most difficult to figure out in general, but it is also the most useful. We will spend serious time on this in this course.

Note that there are a number of things we don't care about when we compute running time for an algorithm:

1. The speed of the machine.
2. The amount of memory it has.
3. The cache or caches it has.

(Well, this is not quite true. Sometimes we do care about these things, but not in this course.)

So what *do* we care about? Answer: the number of "operations". What is an operation? Answer: any arithmetic operation such as addition, or multiplication; any data structure operation such as assignment. We will generally assume in this course that these are all equally expensive. (For some very sophisticated scientific computations, this is not strictly speaking the right thing to do, but for most algorithms, and in particular, for the ones we will be considering here, it is fine.)

4.2.1 Some plausible but not rigorous estimates

So let's start looking at these efficiency questions. We can make reasonable guesses about the worst-case and best-case running times as follows:

1. It's reasonable to guess that the best-case time will occur when the input array is already sorted in the correct order. In this case, the **while** loop in the algorithm will have a constant cost for each value of j—in fact, it will not even be executed once, ever. The only cost will be the cost of the test at the top of the loop. So the cost for each iteration of the j loop is constant. Let's call it c. Then the running time is

$$T(n) = \sum_{j=1}^{n} c = (n-1)c$$

   so it's essentially *linear* in n.

2. It's also reasonable to guess that the worst-case time will occur when the input array is sorted, but in *reverse* order. In this case, the body of the **while** loop is executed $j-1$ times on the j^{th} iteration of the outer loop. And so if we say that a is the cost of the constant overhead of the instructions outside the **while** loop, and if the cost of the body of the **while** loop is c, we have

$$T(n) = \sum_{j=2}^{n} \big(a + (j-1)c\big)$$

   which is of the form $An^2 + Bn + C$ for some constants A, B, and C.

   Thus in this case, the cost is *quadratic* in n. This is of course considerably worse than linear.

4.2 Exercise *Prove the assertion made above: that if there are positive constants a and c such that*

$$T(n) = \sum_{j=2}^{n} \big(a + (j-1)c\big)$$

then there are constants A, B, and C such that

$$T(n) = An^2 + Bn + C$$

Of course A, B, and C depend on a and c, but do not *depend on n. You should show also that $A > 0$. That's an important fact.*

4.2.2 A careful and rigorous analysis

Well, actually this was just a little bit sloppy. We can do rather better by a careful analysis of the code. We can see that the key to determining the cost of the algorithm is the number of times the **while** loop is executed. And this may differ for different values of the **for** loop index j. So we will let t_j denote the number of times the test at the head of the **while** loop is executed. This is 1 more than the number of times the body of the **while** loop is executed.

Note that we have

$$1 \le t_j \le j$$

Then we can annotate the code like this:

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** $A.length$ **do**	c_1	n
$key \leftarrow A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.		
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$ **do**	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

The first thing we should do is write down an expression for the total running time $T(n)$ of this algorithm. It might at first appear that the expression $\sum_{j=2}^{n} t_j$ should be pulled out by itself. However, it turns out (as we'll see below) that it actually is more natural to pull out $\sum_{j=2}^{n}(t_j - 1)$.

When we do this, we get

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

Now we can be precise about our estimates:

worst-case time $T(n)$ will be as big as possible when $t_j = j$. This will happen precisely when A

is initially sorted in reverse order. Since

$$\sum_{j=2}^{n}(j-1) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$$

we see that $T(n)$ is of the form $an^2 + bn + c$—it is "quadratic in n".

best-case time $T(n)$ will be as small as possible when $t_j = 1$. This will happen precisely when A is initially sorted in the proper order. In this case we have

$$\sum_{j=2} n(1-1) = 0$$

and so $T(n)$ is of the form $an + b$—it is "linear in n".

Finding the average-case time is much more difficult, because it involves finding the *average* value of the sum $\sum_{j=2}^{n}(t_j - 1)$ over all possible permutations of the input. This seems at first to be impossible. However, it turns out that it can actually be done, and quite elegantly, in fact, as we will now show.

4.3 Permutations and inversions

Remember that we are taking the average over all permutations of the input data.

Now associated with each permutation is a set of *inversions*.

Definition *An* inversion *of a sequence of numbers* $(a_1, a_2, \ldots, a_n)$ *is an ordered pair* (i, j) *such that*

- $i < j$, *and*
- $a_i > a_j$

In other words, it is the set of ordered pairs of **indices** which are "in order" but whose corresponding **values** are "out of order".

Suppose, for instance we consider the sequence $(5, 2, 3, 7, 1)$, which we denote as follows:

n	a_n
1	5
2	2
3	3
4	7
5	1

So $a_3 = 3$, and $a_4 = 7$. This sequence has 6 inversions:

inversion (i, j)	elements (a_i, a_j)
$(1, 2)$	$(5, 2)$
$(2, 5)$	$(2, 1)$
$(3, 5)$	$(3, 1)$
$(4, 5)$	$(7, 1)$
$(1, 3)$	$(5, 3)$
$(1, 5)$	$(5, 1)$

Note that if I tell you I have a permutation of 11 elements and that $(2, 4)$ is an inversion in that permutation, that tells you nothing about what the actual elements are—2 and 4 are just the *indexes* of the elements, not the *values* of the elements.

In the above example, for instance, if I did not tell you that the permutation was the ordered set $(5, 2, 3, 7, 1)$, but I only told you that the permutation I had in mind had the inversion $(4, 5)$, you would have no way of knowing that the two elements in question were 7 and 1.

The way inversions come into our problem is this: (I'll write it as a lemma as a way of isolating it as a separate result.)

4.3 Lemma *The number of inversions of the input data is the number of times the* ***while*** *loop is executed.*

PROOF. When we are at the top of the **while** loop, we have to decide whether or not $A[i]$ is bigger than the key (which was the original $A[j]$). Now the index j of the element $A[j]$ is the original index in the input data, because until we get to the end of the j^{th} iteration of the **for** loop we have not moved the element $A[j]$. The element $A[i]$ may have a different index than it did originally, but in any case, its original index as well as i (which is its current index) are both $< j$. Therefore while we are executing the j^{th} iteration of the **for** loop, we know that the number of inversions (i, j) with $i < j$ in the current array is the same as in the original array—that is, $A[j]$ is still in its original position, and each of the elements $A[i]$ may have moved from its original position $A[i_{\text{orig}}]$, but in any case, both i and i_{orig} are $< j$.

Now there are two possibilities:

- $A[i] < A[j]$. In this case, the pair (i, j) is not an inversion of the current sequence. And in this case, we do not execute the body of the **while** loop—we exit the loop and continue on to the last statement of the **for** loop.
- $A[i] > A[j]$. In this case, the pair (i, j) *is* an inversion of the current sequence. And in this case, we *do* execute the body of the **while** loop, which has the effect of getting rid of that inversion by moving $A[i]$ up one, freeing up a place below it for $A[j]$ to be inserted.

Further, because we know that at the beginning of the j^{th} iteration of the **for** loop the elements $A[1\,..\,(j-1)]$ are in increasing order, we know that the number of iterations of the **while** loop on the j^{th} iteration of the **for** loop is exactly the number of inversions (i, j) with $i < j$. And by what we have seen above, this is exactly the same as the number of inversions (i, j) with $i < j$ in the original sequence.

As the **for** loop continues, therefore, the total number of times the **while** loop executes is just the total number of inversions in the entire original sequence. □

Thus, we have shown that the total number of inversions in the original sequence is just the expression $\sum_{j=2}^{n}(t_j - 1)$.

4.4 How many inversions are there?

Remember that our problem was to find the average value of the sum $\sum_{j=2}^{n}(t_j - 1)$ over all permutations. We now have just seen that this is just the total number of inversions in all permutations, divided by the number of permutations (which, as we know, is $n!$).

So we have reduced the problem to finding the total number of inversions in all permutations on n elements.

At first, this may seem not to be of much help. But in fact, there is a nice way of looking at it which gives us the answer very quickly:

To each permutation $(a_1, a_2, \ldots, a_n)$ there is a *reverse permutation*, namely $(a_n, \ldots, a_2, a_1)$, which consists of the same numbers in the reverse order. Just to establish some notation, we can name the elements of the reverse permutation like this: $(b_1, b_2, \ldots, b_n)$. So we have $b_1 = a_n$, $b_2 = a_{n-1}$, and so on.

For example, the permutation we considered above

$$(5, 2, 3, 7, 1)$$

corresponds to the reverse permutation

$$(1, 7, 3, 2, 5)$$

Here we have

$$a_1 = 5 = b_5$$
$$a_2 = 2 = b_4$$
$$a_3 = 3 = b_3$$
$$a_4 = 7 = b_2$$
$$a_5 = 1 = b_1$$

We can use this to make some correspondences, like this:

	pair of indices	corresponding pair of elements
original permutation	$(2,3)$	(a_2, a_3)
reverse permutation	$(3,4)$	(b_3, b_4)

Note that

- We have the ordered pair of indices $(2,3)$ in the original permutation corresponding to $(3,4)$ (rather than $(4,3)$) in the reverse permutation, because we consider pairs of indices where the first is less than the second.
- $(2,3)$ is *not* an inversion in the original permutation, but its corresponding pair of indices $(3,4)$ *is* an inversion in the reversed permutation.

In fact, we could write this more systematically:

	pair of indices	corresponding pair of elements
original permutation	(i,j)	(a_i, a_j)
reverse permutation	(j',i')	$(b_{j'}, b_{i'})$

Again, note that we have written (j',i'), because (just as we have $i < j$) we want to have $j' < i'$.

4.4 Exercise *Assume that the number of elements in the permutation is n. What is the formula for i' as a function of i (and n, of course)?*

Now here's the key idea: as we noticed above, the pair of indices (i,j) is an inversion in the original permutation iff the pair of indices (j',i') is not an inversion in the reversed permutation.

Now how many pairs (i,j) with $i < j$ are there in the original sequence? Think of it this way: There are $\binom{n}{2}$ 2-element subsets of the original sequence. Each 2-element subset corresponds to two ordered pairs, of which one has its indices "in the right order". (For example, the 2-element subset $\{3,4\}$ corresponds to the two ordered pairs $(3,4)$ and $(4,3)$. One of these—$(3,4)$—is in the right order.)

Therefore the number of pairs (i,j) in the original sequence with $i < j$ is just $\binom{n}{2}$. And each one of these pairs is either an inversion in the original sequence or corresponds to one (i.e, to an inversion (j',i') in the reverse sequence. Therefore the number of inversions in both those sequences together must be exactly $\binom{n}{2}$. That's the key.

In the example we've been using, there are 5 elements, and so there are $\binom{5}{2}$ pairs:

pairs of indices in the original permutation				pairs of elements in the original permutation			
$(1,2)$	$(1,3)$	$(1,4)$	$(1,5)$	$(5,2)$	$(5,3)$	$(5,7)$	$(5,1)$
	$(2,3)$	$(2,4)$	$(2,5)$		$(2,3)$	$(2,7)$	$(2,1)$
		$(3,4)$	$(3,5)$			$(3,7)$	$(3,1)$
			$(4,5)$				$(7,1)$

The red pairs of indices (on the left) are inversions in the original permutation, and the blue pairs correspond to inversions in the reverse permutation.

We're almost done.

The total number of permutations is $n!$. Since each permutation can be paired with its reverse permutation, the total number of such pairs of permutations is $n!/2$. Each such pair of permutations contains a total of $\binom{n}{2}$ inversions. Therefore the total number of inversions in all the permutations together is

$$\frac{n!}{2}\binom{n}{2}$$

The average number of inversions in a permutation then, is this divided by the number of permutations (which is $n!$). That is, the average number of inversions in a permutation is

$$\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4}$$

4.5 The Average-case analysis: putting it all together

We thus find that the average value of $T(n)$ (averaged over all permutations of the input data) is

$$(c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) + (c_5 + c_6 + c_7)\frac{n(n-1)}{4}$$

And without dragging this out any more, it is clear that this is of the form $an^2 + bn + c$. So the average running time of insertion sort is quadratic in the size of the input data.

5 Merge sort

Merge sort (which I think is due to von Neumann) is an example of a "divide-and-conquer" algorithm. Such algorithms are generally very powerful. Here's the basic idea:

- Divide the problem into two smaller problems.
- Solve each of the smaller problems. (This amounts to a recursive call.)
- Combine the two solutions into a solution of the original problem.

For this to work, of course, there must be a way of dividing the original problem into subproblems in such a way that the solutions of the two sub problems are related in some easy-to-understand way to the solution of the original problem. And—related to this—the costs of steps 1 and 3 must be relatively cheap.

Merge sort works in this way. We start as before with an array $A[1\,..\,n]$ of numbers to be sorted. The top-level call is

$$\textsc{Merge-Sort}(A, 1, n)$$

which sorts all the elements in array A in positions $1 \,..\, n$. And in general, we have calls MERGE-SORT(A, p, r) which (called recursively) sort all the elements in array A in positions $p \,..\, r$. It works like this:

MERGE-SORT(A, p, r)
  if $p < r$ **then**
    $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT$(A, q + 1, r)$
    MERGE(A, p, q, r)

So the whole algorithm (in this case) really depends on step 3—the combining step, which is implemented by the auxiliary function MERGE. Here's how MERGE works:

We start with two lists (from the two previous calls to MERGE-SORT. By induction they are already sorted.

5.1 Exercise *What does "by induction" really mean here?*

Let us denote the two lists by L and R (for "left" and "right"), and say we are looking at them "from the bottom up":

L	R
20	12
13	11
7	9
1	2

At each step we pick the least of the two visible numbers. So the first number we pick is 1.

1

L	R
20	12
13	11
7	9
	2

1 2

L	R
20	12
13	11
7	9

1 2 7

20 12
13 11
9

1 2 7 9

20 12
13 11

1 2 7 9 11

20 12
13

1 2 7 9 11 12

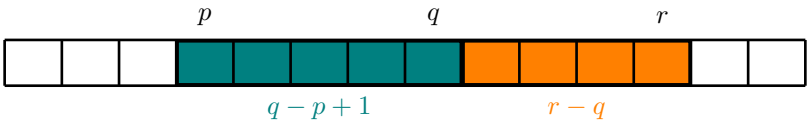
20
13

1 2 7 9 11 12 13

20

1 2 7 9 11 12 13 20

Figure 2 shows the code for MERGE(A, p, q, r). This picture helps explain some of the code in that figure:



PROOF OF CORRECTNESS. Here's our inductive hypothesis, which again in this case has the form of a loop invariant:

5.2 Lemma (The inductive hypothesis) *At the start of each iteration of the* ***for*** *loop on k, the subarray $A[p\,..\,k-1]$ contains the $k-p$ smallest elements of $L[1\,..\,n_1+1]$ and $R[1\,..\,n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.*

PROOF. When $k = p$, this is vacuously true. (Why?)

To go from $k-1$ to k, there are two possibilities: either $L[i] \le R[j]$ or $L[i] > R[j]$.

In the first case, $L[i]$ is the smallest element not yet copied back into A. (Why is this?) So we copy it into A (at the right position), and the loop invariant is maintained.

The second case is similar. □

```
MERGE(A, p, q, r)
    n_1 ← q − p + 1
    n_2 ← r − q
    // Create arrays L[1 .. n_1 + 1] and R[1 .. n_2 + 1]
    for i ← 1 to n_1 do
        L[i] ← A[p + i − 1]
    for j ← 1 to n_2 do
        R[j] ← A[q + j]
    L[n_1 + 1] ← ∞
    R[n_2 + 1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r do
        if L[i] ≤ R[j] then
            A[k] ← L[i]
            i ← i + 1
        else
            A[k] ← R[j]
            j ← j + 1
```

Figure 2: Pseudo-code for MERGE(A, p, q, r).

And that finishes the proof of correctness, because when we're done, k is $r + 1$. □

5.3 Exercise *Rewrite the inductive hypothesis as a sequence of statements, as we did before. Then rewrite the proof using that sequence of statements.*

We note that MERGE(A, p, q, r) takes time "linear" in $n = r - p + 1$. (By this I simply mean that the running time is of the form $an + b$).

To analyze the running time of MERGE-SORT, we look at the steps involved. Say the time to perform MERGE-SORT on an array of n elements is $T(n)$. Then $T(n)$ must be the sum of three terms:

Step 1. ("Divide") This just finds the middle of the array. This can be done in constant time. Say the constant is c.

Step 2. ("Conquer") This consists of the two recursive calls, each on an array of half the size. So the cost of this step is $2T(n/2)$.

Step 3. ("Merge") We have just seen that this is linear in n. Say it is cn. (Note that we're using the same constant c here as in Step 1. This is really OK; we could use any sufficiently large c and get a slightly worse upper bound on the running time; for our purposes that would be fine.)

And finally, there is a "base case": when A contains only 1 element. In this case, certainly the cost is simply some constant. So—since we don't really care what these constants are—we can see that

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

Note how the recursion in the algorithm is mirrored in the recursive nature of this identity.

Now identities like this come up frequently in algorithmic analysis. It's important to have ways of solving them. We'll see a couple. One basic way is to form a *recursion tree*. Figure 3 shows how we do it:

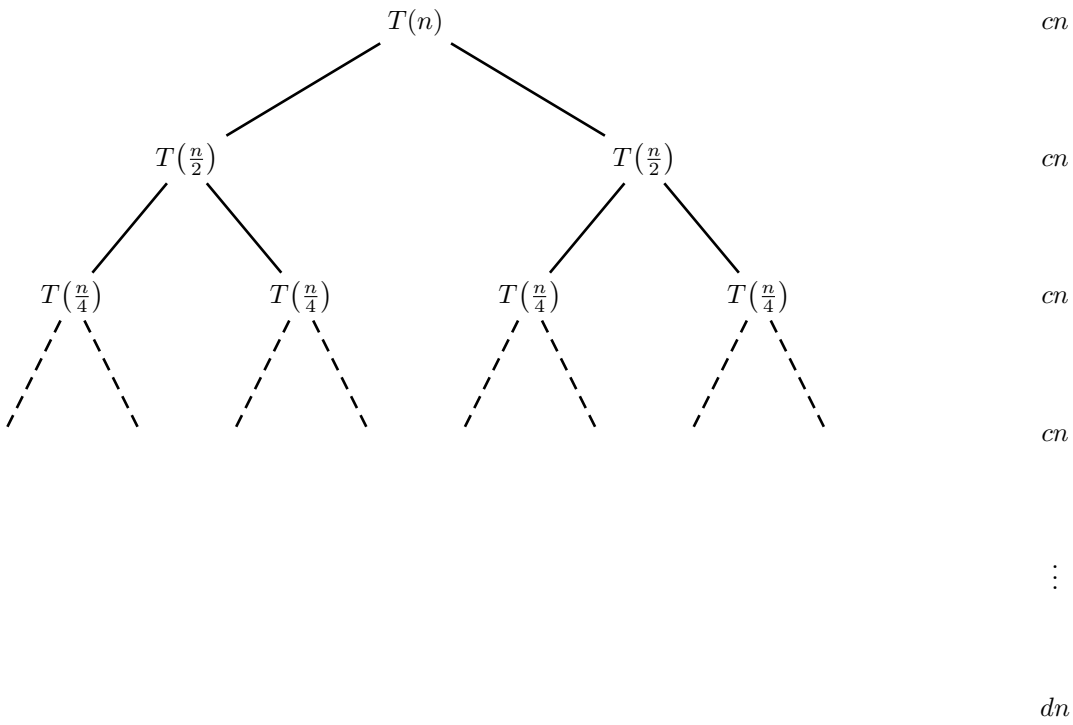


Figure 3: Recursion tree for MERGE-SORT in the simplest case where n is a power of 2.

If $n = 2^p$ then there are p rows with cn on the right, and one last row with dn on the right. Since $p = \log_2 n$, this means that the total cost is

$$cn \log_2 n + dn$$

In other words, this is what we call an "$n \log n$" algorithm.