Here we examine the recurrences for mergesort and quicksort.

**Mergesort**

Recall the mergesort algorithm: we divide the list of things to be sorted into two approximately equal size sublists, sort each of them, and then merge the result. Merging two sorted lists of size \( \frac{n}{2} \) takes time proportional to \( n \), since the merging requires iterating through the elements of each list. If \( n \) is even, then there are \( \frac{n}{2} + \frac{n}{2} = n \) elements in the two lists. If \( n \) is odd then one of the lists has size one greater than the other, but there are still \( n \) steps to the merge.

Let’s assume that \( n \) is a power of 2. This keeps the math simpler since we don’t have to deal with the case that the two sublists are not exactly the same length. In this case, the recurrence relation for mergesort is:

\[
t(n) = 2t\left(\frac{n}{2}\right) + cn.
\]

[ASIDE: If we were to consider a general \( n \), then the correct way to write the recurrence would be:

\[
t(n) = t\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + t\left(\left\lceil\frac{n}{2}\right\rceil\right) + cn
\]

where the \( \left\lfloor \frac{n}{2} \right\rfloor \) means floor\((n/2)\) and \( \left\lceil \frac{n}{2} \right\rceil \) means ceiling\((n/2)\), or ”round down” and ”round up”, respectively. That is, rather than treating \( n/2 \) as an integer division and ignoring the remainder (rounding down always), we would be treating it as \( n/2.0 \) and either rounding up or down. In the lecture, I gave the example of \( n = 13 \) so \( \left\lfloor \frac{n}{2} \right\rfloor = 6 \) and \( \left\lceil \frac{n}{2} \right\rceil = 7 \). In COMP 250, we don’t concern ourselves with this level of detail since nothing particularly interesting happens in the general case, and we would just be getting bogged down with notation. The interesting aspect of mergesort is most simply expressed by considering the case that \( n \) is a power of 2. ]

Another point to note about the mergesort recurrence before we solve it is that there is also a constant term, i.e. we could write the following

\[
t(n) = 2t\left(\frac{n}{2}\right) + c_1 + c_2n.
\]

However, since we are interested in the \( \mathcal{O}(\ ) \) behavior and since the \( c_2n \) term is going to dominate over the \( c_1 \) term, we don’t bother with the constant \( c_1 \) term.

Let’s solve the mergesort recurrent using backwards substitution:

\[
t(n) = 2t\left(\frac{n}{2}\right) + cn
\]
\[
= 2\left(2t\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn
\]
\[
= 4t\left(\frac{n}{4}\right) + cn + cn
\]
\[
= 4\left(2t\left(\frac{n}{8}\right) + c\frac{n}{4}\right) + cn + cn
\]
\[
= 8t\left(\frac{n}{8}\right) + cn + cn + cn
\]
\[
= 2^k t\left(\frac{n}{2^k}\right) + cnk
\]
\[
= n t(1) + c n \log n, \text{ when } n = 2^k
\]
which is $O(n \log_2 n)$ since the dominant term that depends on $n$ is $n \log_2 n$.

What is the intuition for why mergesort is $O(n \log_2 n)$? Think of the merge phases. The list of size $n$ is ultimately partitioned down into $n$ lists of size 1. If $n$ is a power of 2, then these $n$ lists are merged into $\frac{n}{2}$ lists of size 2, which are merged into $\frac{n}{4}$ lists of size 4, etc. So there are $\log_2 n$ "levels" of merging, and each require $O(n)$ work. Hence,

Notice that the bottleneck of the algorithm is the merging part, which takes a total of $cn$ operations at each "level" of the recursion. There are $\log n$ levels.

**Quicksort**

Let’s now turn to the recurrence for quicksort. Recall the main idea of quicksort. We remove some element called the pivot, and then partition the remaining elements based on whether they are smaller than or greater than the pivot, recursively quicksort these two lists, and then concatenate the two, putting the pivot in between.

In the best case, the partition produces two roughly equal sized lists. This is the best case because then one only needs about $\log n$ levels of the recursion and approximately the same recurrence as mergesort can be written and solved.

What about the worst case performance? If the element chosen as the pivot happens to be smaller than all the elements in the list, or larger than all the elements in the list, then the two lists are of size 0 and $n-1$. If this poor splitting happens at every level of the recursion, then performance degenerates to that of the $O(n^2)$ sorting algorithms we saw earlier, namely the recurrence becomes

$$t(n) = cn + t(n - 1).$$

Solving this by backstitution (see last lecture) gives

$$t(n) = c \frac{n(n + 1)}{2} + t(0)n$$

which is $O(n^2)$.

Why is quicksort called “quick” when its worst case is $O(n^2)$? In particular, it would seem that mergesort would be quicker since mergesort is $O(n \log n)$, regardless of best or worst case.

There are two basic reasons why quicksort is "quick". One reason is that the first case is easy to avoid in practice. For example, if one is a bit more clever about choosing the pivot, then one can make the worst case situation happen with very low probability. One idea for choosing a good pivot is to examine three particular elements in the list: the first element, the middle element, and the last element (list[0], list[mid], list[size-1]. For the pivot, one sorts these three elements and takes the middle value (the median) as the pivot. The idea is that it is very unlikely for all three of these elements to be among the three smallest (or three largest). In particular, if the list happens to be close to sorted (or sorted in the wrong direction) then the "median of three" will tend to be close the median of the entire list. Note that the best split occurs if we take the pivot to be the median of the whole list. In practice, such a simple idea works very well, and partitions have close to even size.

The second reason that quicksort is quick is that, if one uses an array list to represent the list, then it is possible to do the partition in place, that is, without using extra space. One needs to be a bit clever to set this up, but it is straightforward to implement. I’ve decided not to
cover the details this year, but if you feel like you are missing out, then do check it out: [https://en.wikipedia.org/wiki/Quicksort](https://en.wikipedia.org/wiki/Quicksort)

The "in place" property of quicksort is a big advantage, since it reduces the number of copies one needs to do. By contrast, the straightforward implementation of mergesort requires that we use a second array and merge the elements into it. This leads to lots of copying, which tends to be slow. There are clever ways to implement mergesort which make it run faster, but the details are way beyond the scope of the course. Besides, experimental results have shown that, in practice, quicksort tends to be faster than mergesort even if the 'in place' mergesort method is used. Quicksort truly is very quick (if done in place).

**Logarithms: some review, some new**

When you signed up for COMP 250, you probably didn’t expect that you would need to think so much about logarithms. Surprise, logarithms are very important in understanding algorithms! Because you may be rusty on logarithms, here I will review some basic and important properties that you will need to know or at least be familiar with in COMP 250.

Let’s begin with the definition. The logarithm is the inverse function of the exponential: if we take the exponential function for some base $b$, namely $b^x$, then the inverse function is called the *logarithm* of that base $b$. To be concrete, for any positive base $b$ and any $x > 0$, by definition:

$$ \log_b(b^x) = x $$

$$ b^{\log_b x} = x. $$

Note that in computer science we often use bases $b = 2, 10$ but other bases arise naturally too.

Here are a few basic properties of logs that *follow from* the definition. Here we assume $a, b, c > 0$.

**Claim 1:**

$$ \log_b(a^c) = c \log_b a. $$

**Proof:** let $x$ be the number such that $b^x = a$, that is, $x = \log_b a$. Then,

$$ \log_b(a^c) = \log_b((b^x)^c) = \log_b(b^{xc}) = xc = c \log_b a. $$

Note that $(b^x)^c = b^{xc}$ is obvious when $c$ is an integer. The relation is also easy to see when $c$ is a rational number ($c = \frac{p}{q}$ when $p, q$ are integers). The case of irrational $c$ is less obvious, but it can be shown to hold too (details omitted).

**Claim 2:**

$$ \log_b(ac) = \log_b a + \log_b c. $$

**Proof:** let $a = b^x$ and $c = b^y$, that is, $x = \log_b a$ and $y = \log_b c$. Then,

$$ \log_b(ac) = \log_b(b^x b^y) = \log_b b^{x+y} = x + y \text{ from Claim 1.} $$

which proves the claim.

The next two properties compare logarithms of two different bases, that is, $\log_b x$ versus $\log_a x$. Intuitively, if $b > a$, then $\log_b x < \log_a x$ since it should take a smaller power of $b$ to be equal to some given power of $a$. 

last updated: 13\textsuperscript{th} Oct, 2016 3  lecture notes ©Michael Langer
Claim 3:

\[ \log_b x = (\log_b a)(\log_a x) \]

Proof: From the definition of logarithms as the inverse of exponential,

\[ a^{\log_a x} = x. \]

Taking the \( \log_b \) of both sides:

\[ \log_b(a^{\log_a x}) = \log_b x \]

and so from Property 1:

\[ (\log_a x)(\log_b a) = \log_b x. \]

Claim 4

\[ a^{\log_b c} = c^{\log_b a} \]

Proof: I’ll let \( x \) be the left side, do some manipulations, and show that \( x \) is also equal to the right side.

\[ x = a^{\log_b c} \]

Thus, \[ \log_b x = \log_b a^{\log_b c} = (\log_b a)(\log_b c) = \log_b c^{(\log_b a)} \]

Thus, \[ x = c^{\log_b a}. \]

The last line follows from taking the inverse of \( \log_b \) on both sides of the previous line, i.e. the inverse of \( \log_b \) is the exponential with base \( b \).