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:

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

[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.]

Let’s solve the mergesort recurrent using backwards substitution:

\[
t(n) \quad = \quad cn + 2t\left(\frac{n}{2}\right) \\
\quad = \quad cn + 2 \left( cn^2 + 2t\left(\frac{n}{4}\right) \right) \\
\quad = \quad cn + cn^2 + 4t\left(\frac{n}{4}\right) \\
\quad = \quad cn + cn^2 + 4 \left( cn^2 + 2t\left(\frac{n}{8}\right) \right) \\
\quad = \quad cn + cn + cn + 8\ t\left(\frac{n}{8}\right) \\
\quad = \quad cnk + 2^k \ t\left(\frac{n}{2^k}\right) \\
\quad = \quad cn \log_2 n + n \ t(1), \quad \text{when} \ n = 2^k
\]

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.
On the base case of mergesort

In the mergesort algorithm last week, we had a base case of \( n = 1 \). What if we had stopped the recursion at a larger base case? For example, suppose that when the list size has been reduced to 4 or less, we switch to running bubble sort instead of mergesort. Since bubble sort is \( O(n^2) \), one might ask whether this wouldcause the mergesort algorithm to increase from \( O(n \log_2 n) \) to \( O(n^2) \). Let’s solve the recurrence for mergesort by assuming \( t(n) = 2t\left(\frac{n}{2}\right) + c_1n \) when \( n > 4 \) and but that some other \( t(n) \) holds for \( n \leq 4 \).

Assume \( n \) is a power of 2 (to simplify the argument). We want to stop the backsubstitution at \( t(4) \) on the right side. So we let \( k \) be such that \( \frac{n}{2^k} = 4 \), that is, \( 2^k = \frac{n}{4} \).

\[
t(n) = c \cdot n + 2 \cdot t\left(\frac{n}{2}\right) \\
= \ldots \\
= c \cdot n \cdot k + 2^k \cdot t\left(\frac{n}{2^k}\right), \text{ and letting } 2^k = \frac{n}{4} \text{ gives...} \\
= c \cdot n \cdot (\log_2 n - 2) + \frac{n}{4} \cdot t(4) \\
= c \cdot n \cdot \log_2 n - 2cn + \frac{n}{4} \cdot t(4)
\]

which is still \( O(n \log_2 n) \) since the dominant term that depends on \( n \) is \( n \log_2 n \).

The subtle part part of this problem is that you may be thinking that if we are doing say bubblesort when \( n \leq 4 \) then we need to somehow put an \( n^2 \) dependence in somewhere. But you don’t need to do this! Since we are only switching to bubblesort when \( n \leq 4 \), the term \( t(4) \) is a constant. This isthe time it takes to solve a problem of size 4.

Note: while this might seem like a toy problem, it makes an important point. Sometimes when we write recursive methods, we find that the base case can be tricky to encode. If there is a slower method available that can be used for small instances of the problem, and this slower method is easy to encode, then use it!

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 then all the elements in the list, or larger then 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 \( \text{list}[0], \text{list}[\text{mid}], \text{list}[\text{size}-1] \). 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).

**Exercise**

In the lecture, I asked you to solve this one:
\[ t(n) = t\left(\frac{n}{2}\right) + n \]

This is similar to binary search, but now we have to do \( n \) operations during each call.

What do you predict? Is this \( O(\log_2 n) \) or \( O(n) \) or \( O(n^2) \) or what?

Here is the solution:
\[ t(n) = n + t\left(\frac{n}{2}\right) \]
\[ = n + \frac{n}{2} + t\left(\frac{n}{4}\right) \]
\[ = n + \frac{n}{2} + \frac{n}{4} + t\left(\frac{n}{8}\right) \]
\[ = n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{k-1}} + t\left(\frac{n}{2^k}\right) \]
\[ = n + \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2 + t(1), \text{ when } 2^k = n \]
\[ = n + \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2 + 1 - 1 + t(1) \]
\[ = n \sum_{i=0}^{\log_2 n} 2^i + t(1) - 1 \]
\[ = \frac{2^{(\log_2 n)+1} - 1}{(2 - 1)} + t(1) - 1 \]
\[ = 2n - 1 + t(1) - 1 \]
Tree traversal

Often we wish to iterate through or “traverse” all the nodes of the tree. We generally use the term tree traversal for this. There are two aspects to traversing a tree. One is that we need to follow references from parent to child, or child to its sibling. The second is that we may need to do something at each node. I will use the term “visit” for the latter. Visiting a node means doing some computation at that node. We will see some examples later.

Depth first traversal

The first two traversals that we consider are called “depth first”. In these traversals, a node and all its descendents are visited before the next sibling is visited. There are two ways to do depth-first-traversal of a tree, depending on whether you visit a node before its descendents or after its descendents. In a pre-order traversal, you visit a node, and then visit all its children. In a post-order traversal, you visit the children of the node (and their children, recursively) and then visit the node.

```
def depthfirst_Preorder(root):
    if (root is not empty):
        visit root
        for each child of root
            depthfirst_Preorder(child)
    
```

An example is illustrated in the lectures slides. Suppose we have a file system. The directories and files define a tree whose internal nodes are directories and whose leaves are either empty directories or files. We first wish to print out the root directories, namely list the subdirectories and files in the root directory. For each subdirectory, we also print its subdirectory and files, and so on. This is all done using a pre-order traversal. The visit would be the print statement. Here is an example of what the output of the print might look like. (This is similar to what you get on Windows when browsing files in the Folders panel.)

```
My Documents  (directory)
    My Music    (directory)
        Raffi    (directory)
            Shake My Sillies Out (file)
            Baby Beluga (file)
        Eminem    (directory)
            Lose Yourself (file)
    My Videos  (directory)
        :  (file)
    Work       (directory)
        COMP250  (directory)
        :  
```

For a postorder traversal, one visits a node after having visited all the children of the node.

```java
depthfirst_Postorder(root){
    if (root is not empty){
        for each child of root
            depthfirst_Postorder(child){
                visit root
            }
    }
}
```

Let’s look at an example of post-order traversal. Suppose we want to calculate how many bytes are stored in all the files within some directory including all its sub-directories. This is post-order because in order to know the total bytes in some directory we first need to know the total number of bytes in all the subdirectories. Hence, we need to visit the subdirectories first. Here is an algorithm for computing the number of bytes. It traverses the tree in postorder in the sense that it computes the sum of bytes in each subdirectory by summing the bytes at each child node of that directory. (Frankly, in this example it is a big vague what I mean by “visit”, since computing \texttt{sum} doesn’t just happen after visiting the children but rather involves steps that occur before, during, and after visiting the children. The way I think of this is that if we were to store \texttt{sum} as a field in the node, then we could only do this after visiting the children.)

```java
numBytes(root){
    if root is a leaf
        return number of bytes at root
    else{
        sum = 0 // local variable
        for each child of root{
            sum += numBytes(child)
        }
        return sum
    }
}
```

**Depth first traversal without recursion**

As we have discussed already in this course, recursive algorithms are implemented using a call stack which keep track of information needed in each call. (Recall the stack lecture.) You can sometimes avoid recursion by using an explicit stack instead. Here is an algorithm for doing a depth first traversal which uses a stack rather than recursion. As you can see by running an example (see lecture slides), this algorithm visits the list of children of a node in the opposite order to that defined by the \texttt{for} loop.

Is this algorithm preorder or postorder? The \texttt{visit cur} statement occurs prior to the \texttt{for} loop. You might think this automatically makes it preorder. However, the situation is more subtle than that. If we were to move the \texttt{visit cur} statement to be after the \texttt{for} loop, the order of the visits of the nodes would be the same. Moreover, nodes would still be visited before their children. So, this would still be a preorder traversal.
treeTraversalUsingStack(root){
    s.push(root)
    while !s.isEmpty(){
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
        // 'visit cur' could be put here instead
    }
}

Breadth first traversal

What happens if we use a queue instead of a stack in the previous algorithm?

treeTraversalUsingQueue(root){
    q = empty queue
    q.enqueue(root)
    while !q.isEmpty() {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }

As shown in the example in the lecture, this algorithm visits all the nodes at each depth, before proceeding to the next depth. This is called breadth first traversal. The queue-based algorithm effectively does the following:

for i = 0 to height
    visit all nodes at level i

You should work through the example in the slides to make sure you understand why using queue here is different from using a stack.

A note about implementation

Recall first-child/next-sibling data structure for representing a tree, which we saw last lecture. Using this implementation, you can replace the line

for each child of cur
...

with the following

child = child.firstChild
while (child != null){
    ... // maybe do something at that child
    child = child.nextSibling
}
Here we are iterating through a (singly linked) list of children.