Mergesort

In lecture 6, we saw three algorithms for sorting a list of n items. We saw that, in the worst case, all of these algorithm required $O(n^2)$ operations. Such algorithms will be unacceptably slow if n is large.

To make this claim more concrete, consider that if $n = 2^{20} \approx 10^6$ i.e one million, then $n^2 \approx 10^{12}$. How long would it take a program to run that many instructions?

Today's processors run at about 10^9 basic operations per second (i.e. GHz). So a problem that takes in the order of 10^{12} operations would require thousands of seconds of processing time. Having a multicore machine with say 4 processors only can speed things up by a factor of 4 – max! – which doesn't change the argument here.

Today we consider an alternative sorting algorithm that is much faster that these earlier $O(n^2)$ algorithms. The new algorithm is called *mergesort*. Here is the idea. If the list has just one number ($n = 1$), then do nothing. Otherwise, partition the list of n elements into two lists of size about $n/2$ elements each, sort the two individual lists (recursively, using mergesort), and then merge the two sorted lists.

For example, suppose we have a list

$$< 8, 10, 3, 11, 6, 1, 9, 7, 13, 2, 5, 4, 12 > .$$

We partition it into two lists

$$< 8, 10, 3, 11, 6, 1 > \quad < 9, 7, 13, 2, 5, 4, 12 > .$$

and sort these (by applying mergesort recursively):

$$< 1, 3, 6, 8, 10, 11 > \quad < 2, 4, 5, 7, 9, 12, 13 > .$$

Then, we merge these two lists

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13.$$

Here is pseudocode for the algorithm. Note that it uses a helper method `merge` which in fact does most of the work.

```
mergesort(list){
   if  list.length == 1
      return list
   else{
      mid = (list.size - 1) / 2
      list1 =  list.getElements(0,mid)
      list2 =  list.getElements(mid+1, list.size-1)
      list1 = mergesort(list1)
      list2 = mergesort(list2)
      return  merge( list1, list2 )
   }
}
```

Here is the merge algorithm. Note that it has two phases. The first phase initializes a new list (empty), steps through the two lists, (`list1`) and (`list2`), compares the front element of each list and removes the smaller of the two, and this removed element to to the back of the merged list. *See the detailed example in the slides for an illustration.*

The second phase of the algorithm starts after one of `list1` or `list2` becomes empty. In this case, the remaining elements from the non-empty list are moved to `list`. This second phase uses two `while` loops in the above pseudocode, and note that only one of these two loops will be used since we only reach phase two when one of `list1` or `list2` is already empty.

```
merge( list1,  list2){
   new empty list
   while  (list1 is not empty) & (list2 is not empty){
       if  (list1.first < list2.first)
          list.addlast( list1.removeFirst(list1) )
       else
          list.addlast( list2.removeFirst(list2) )
   }
   while list1 is not empty
       list.addlast( list1.removeFirst(list1) )
   while list2 is not empty
       list.addlast( list2.removeFirst(list2) )
   return  list
}
```

I have written the `mergesort` and `merge` algorithms using abstract list operations only, rather than specifying how exactly it is implemented (array list versus linked list). Staying at an abstract level has the advantage of getting us quickly to the main ideas of the algorithm: what is being computed and in which sequence? However, be aware that there are disadvantages of hiding the implementation details, i.e. the data structures. As we have seen, sometimes the choice of data structure can be important for performance.

For example, compare an array versus a linked list implementation. The call `getElements()` within `mergesort` would be different for these two data structures. For the array, `getElements()` might just compute the start and end indices for the two lists. These indices could be passed as parameters to the `mergesort` calls. For a linked list, it would be necessary to iterate through the list to find the location of the `mid` element, and one could then break up the list into the `list1` and `list2` with the `mid` list element being at the tail of `list1` and the `mid + 1` element being at the head of `list2`.

For the `merge`, if one were using an array, then one could use a second array (`list`) as a buffer for doing the merges. One could copy the elements from `list1` and `list2` to this second array. At the next level of the recursion, one could copy back to the first array, and go back and forth. So the space requirements would be double the size of the original list.

mergesort is $O(n \log n)$

There are $\log n$ levels of the recursion, namely the number of levels is the number of times that you can divide the list size n by 2 until you reach 1 element per list. The number of instructions that

must be executed at each level of the recursion is proportional to the number n of items in the list. Thus, the total number of instructions is proportional to $n * \log_2 n$, or as usually written $n \log_2 n$. I will discuss this again a few lectures from now, when we study *recurrences*.

To appreciate the difference between the number of operations for the earlier $O(n^2)$ sorting algorithms versus $O(n \log n)$ for mergesort, consider the following table.

n	$\log n$	$n \log n$	n^2
$10^3 \approx 2^{10}$	10	10^4	10^6
$10^6 \approx 2^{20}$	20	20×10^6	10^{12}
$10^9 \approx 2^{30}$	30	30×10^9	10^{18}
...	...	...	...

Thus, the time it takes to run mergesort is significantly less than the time it takes to run bubble/selection/insertion sort, when n becomes large. Very roughly speaking, on a computer that runs 10^9 operations per second running mergesort on a list of size $n = 10^9$ would take in the order of minutes, whereas running insertion sort would take centuries. (After class, one student asked me how I came up with such time estimates. That's easy: just consider there are 60 seconds/minute, 60 minutes/hour, 24 hours/day, etc.)

In the lectures slides, I went over an examples of how the various calls to `mergesort` and `merge` work. The tricky part is to see the order of the various recursive calls and exits. It is easy to understand this with pictures, so please see the slides. (We will many more examples of recursive algorithm later in the course when we look at trees and graphs, so if you don't yet get this, then please be patient with yourself – you will!)

Quicksort

Another well-known recursive algorithm for sorting a list is *quicksort*. This algorithm is quite similar to mergesort but there are important differences that I will highlight.

[ASIDE: I said in the lecture that I would not examine you on Quicksort because it is covered in COMP 251. However, I am going to renege on that. The last few offerings (and perhaps the next) of COMP 251 don't cover it, whereas the last few offerings of COMP 250 did cover it. So, you need to learn it.]

At each call of quicksort, one sorts a list as follows. An element `e` known as the *pivot* is removed from the current list. For example, `e` might be the first element. The remaining elements in the list are then partitioned into two lists: `list1` which contains those smaller than `e`, and `list2` which contains those elements that are greater than or equal to `e`. The two lists `list1` and `list2` are recursively sorted. Then the two sorted lists and the pivot are concatenated into a sorted list containing the original elements (specifically, `list1` followed by `e` followed by `list2`).

```
quicksort(list){
   if  list.length <= 1
      return list
   else{
      pivot = list.removeFirst()   // or some other element
      list1 = list.getElementsLessThan(e)
      list2 = list.getElementsNotLessThan(e) // i.e. the rest
      list1 = quicksort(list1)
      list2 = quicksort(list2)
      return  concatenate( list1, e, list2 )
   }
}
```

Unlike mergesort, most of the work in quicksort is done prior to the recursive calls. Given a pivot element `e` that has been removed from the list, the algorithm goes through the rest of the list and compares each element to `e`. This takes time proportional to the size of `list` in that recursive call, since one needs to examine each of the elements in the list and decide whether to put it into either `list1` or `list2`. The concatenation that is done after sorting `list1` or `list2` might also involve some work, depending on the data structure used.

In the lecture, I briefly discussed how the performance of quicksort can be bad in the worst case. I will fill in that discussion a few lectures from now when we discuss "recurrences".

There is lots more I could say about quicksort. For example, one common and simple implementation of quicksort uses a single array. That is, one can implement quicksort with no extra space required for copying the elements (unlike in mergesort, where it is very natural to use an extra array for the `merge` step). Sorting without using any extra space is called *in place* sorting.[1] If you would like to see how quicksort can be done in-place with an array, see `https://en.wikipedia.org/wiki/Quicksort`. I will not examine you on that, since I didn't present it in the lecture, but those of you who are Majoring in CS should give it 20-30 minutes so you get the basic idea.

[1] Bubble sort, selection sort, and insertion sort were also "in place" algorithms.