

Mergesort

In lecture 3, we saw the “insertion sort” algorithm for sorting n items. We saw that, in the worst case, this algorithm requires $\frac{n(n-1)}{2}$ or $\frac{n^2}{2} - \frac{n}{2}$ or operations. As discussed in the lecture slides, n^2 can be very prohibitively large if the number of items to be sorted is too large. e.g. if $n = 2^{20} \approx 10^6$, then $n^2 \approx 10^{12}$. Today’s machines run at about 10^9 operations per second (i.e. GHz), and so this means thousands of seconds to sort such a list (worst case).

We now consider an alternative sorting algorithm that runs much faster in the worst case. This algorithm is called *mergesort*. Here is the idea. If there is just one number to sort ($n = 1$), then do nothing. Otherwise, partition the list of n elements into two lists of size about $n/2$ elements each, 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

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

and sort these (by applying mergesort recursively):

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

Then, we merge these two lists

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

Algorithm: mergesort(list)

Input: list of elements that can be indexed by position

Output: Sorted list

```

if (list.length = 1) then
    return list
else
    mid ← (list.size - 1) / 2
    l1 ← list.getElements(0,mid)
    l2 ← list.getElements(mid+1,list.size-1)
    return merge( mergesort(l1),mergesort(l2))
end if

```

As we will see when we discuss recurrences a few lectures from now, most of the operations in mergesort are done in the **merge** stage. The partitioning of a list into two equal pieces mainly involves computing the midpoint mid to split the list, and the number of midpoints that are computed is

$$1 + 2 + 4 + \frac{n}{2} = n - 1.$$

Algorithm: `merge(l1, l2)`

Input: Sorted sequences `l1` and `l2`

Output: Sorted sequence `l` containing the elements from `l1` and `l2`

```

initialize empty list l
while l1 is not empty & l2 is not empty do
  if l1.first < l2.first then
    l.addlast( l1.remove(l1.first))
  else
    l.addlast( l2.remove(l2.first))
  end if
end while
while l1 is not empty do
  l.addlast( l1.remove(l1.first))
end while
while l2 is not empty do
  l.addlast( l2.remove(l2.first))
end while
return l

```

The n steps to find the *mid* values turns out to be much smaller than the number of operations required for the **merge** steps. As I sketched out in class and as I will discuss in detail later when we cover recurrences, the **merge** steps require about $cn \log n$ steps. The basic idea is that there are $\log n$ levels of the recursion (namely the number of times that you can divide n by 2 before you reach 1) and at each level there are a total of n elements that need to be merged. (*If you don't understand this yet, that is fine. I will return to it when we discuss recurrences. You do not need to understand it for the midterm.*)

To appreciate the difference between the worst case number of operations for insertion sort (say n^2) versus the worst case¹ (say $n \log n$), 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} |
| ... | ... | ... | ... |

¹mergesort always takes $cn \log n$ operations, i.e. best case = worst case