Sorting

In the sorting problem we are given an unordered list (or array) of elements and we must produce a list (or array) with the elements rearranged into the correct order. This assumes that we have defined a total ordering on the possible data values; then the goal is that the output list be correctly ordered. For the purposes of illustration, we assume that the elements are integers and the comparison operation is less than or equal to.

Thus, if the input elements are

```
'(19 2 17 29 6 31 5 3)
```

the output elements should be

```
'(2 3 5 6 17 19 29 31).
```

There are many different sorting algorithms; we consider just two: insertion sort and merge sort.

Insertion sort

The basic operation in insertion sort is to insert a new element into a correctly sorted list of elements so that the resulting list is correctly sorted. Then insertion sort simply inserts the elements of the input list one by one into an initially empty list to produce the output list correctly sorted. Here is Racket code for the insertion operation.

```
(define (insert item lst)
  (cond
    [(null? lst)
     (list item)]
    [(<= item (car lst))
     (cons item lst)]
    [else
     (cons (car lst) (insert item (cdr lst)))]))
```

This code may be understood as follows. To insert an item into an empty list, just return a list consisting of the item. If the list is nonempty and the item is less than or equal to the first element of the list, then just add the item as the first element of the list and return it. If the list is nonempty, but the item is greater than the first element of the list, then recursively insert the item into the rest of the list, add the first element of the list back to the beginning, and return the result.

To see how this works, consider the tree of calls for `(insert 17 '(2 19))`.

```
insert 17 '(2 19) => '(2 17 19)
 /    |        \
cons  2      insert 17 '(19) => '(17 19)
```

First we observe that the running time of insert is $O(Q)$, where Q is the number of comparison operations `<=` that the procedure does. This is because the other operations (null?, car, cdr, cons) at each level of recursive call are $O(1)$, so the total time will be proportional to the number of recursive calls, and each recursive call

entails another comparison operation. The worst case number of comparisons is n, where n is the number of elements in the sorted list, because we may have to compare the item with every element of the list.

What about the running time of insertion sort? It will be proportional to the number of comparisons made, which in the worst case is 0 to insert an element into the empty list, then 1 to insert an element into a list of 1 element, then 2 to insert an element into a list of 2 elements, and so on, up to $n-1$ to insert the last element into a list of $n-1$ elements. Thus, the worst case total number of comparisons is

$$0 + 1 + 2 + \ldots + (n-1) = n(n-1)/2.$$

Thus the running time of insertion sort is quadratic, that is, $O(n^2)$. Depending on how you implement insertion sort, the worst case is likely to be either an already-sorted list, or the reverse of an already-sorted list, in which the element to be inserted must be compared with all the elements of the list it is being inserted into. This shows that the worst case running time of insertion sort is $\Theta(n^2)$.

Merge sort

Can any sorting algorithm do better than the $\Theta(n^2)$ of insertion sort? Yes, and we will see one of them: merge sort achieves worst case running time of $\Theta(n \log n)$. The fundamental operation for merge sort is to merge two correctly sorted lists into one list containing all the elements of both lists in correctly sorted order. For example, we want

```
(merge '(2 17 19 29) '(3 5 6 31)) => '(2 3 5 6 17 19 29 31).
```

The way we proceed is to compare the first elements of the lists to be merged. The smaller element cons'ed to the result of merging the rest of its list with the other list. For the base cases, if either one of the lists is empty, the result is just the other list. Here is a Racket procedure for merge.

```
(define (merge lst1 lst2)
  (cond
   [(null? lst1)
    lst2]
   [(null? lst2)
    lst1]
   [(<= (car lst1) (car lst2))
    (cons (car lst1) (merge (cdr lst1) lst2))]
   [else
    (cons (car lst2) (merge lst1 (cdr lst2)))]))
```

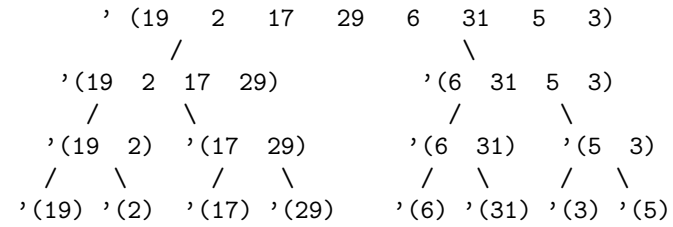
To see how it works, we consider the tree of calls of an example.

```
       merge '(2 19) '(17 29) => '(2 17 19 29)
        /    |     \
      cons   2     merge '(19) '(17 29) => '(17 19 29)
                    /     |     \
                 cons     17    merge '(19) '(29) => '(19 29)
                                 /    |      \
                              cons   19     merge '() '(29) => '(29)
```

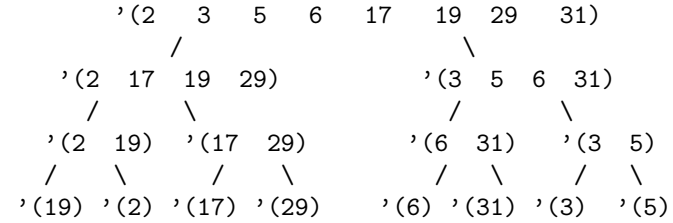
The running time of merge is $O(Q)$, where Q is the number of comparisons done. This is because the base cases simply return one of the two input lists (time $O(1)$), and each comparison results in one recursive

call, and cons'ing one element onto the result, both $O(1)$ operations. In the worst case, the number of comparisons is $n-1$, where n is the total number of elements in both lists. (The minus 1 comes from the fact that we always return a list containing the last element without paying an additional comparison; we only used 3 comparisons in the example above.)

Then merge sort operates by "divide and conquer" using merge. That is, it divides the input list into two (nearly) equal length lists, recursively merge sorts the two lists, and then merges them together to get a sorted list of all the elements. We consider the original list as an example; first we see the tree of arguments to recursive calls of merge sort to itself, assuming that it divides its input list into the first half and the second half.



The base cases are to sort an empty or one-element list, which can just be returned as sorted. The merges then happen as follows.



Once again, we can just bound the number of comparisons that are used; comparisons are only performed in the merge procedure. In every level of this merge tree, there are a total of at most $(n-1)$ comparisons performed, where n is the number of elements in the original input list. The number of levels of the tree is bounded by $(\log_2 n)+1$ so the total number of comparisons and the worst case running time of merge sort are $O(n \log n)$. (If the original list is not a power of 2 in length, we can imagine that it is "rounded up" to the next power of 2; the number of comparisons actually used will be bounded above by the bound for this fictitious larger list.)